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.[1] 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, polygon, Bézier curve, and so on—and inherently possesses any attributes we might care about, such as color, size, and location.

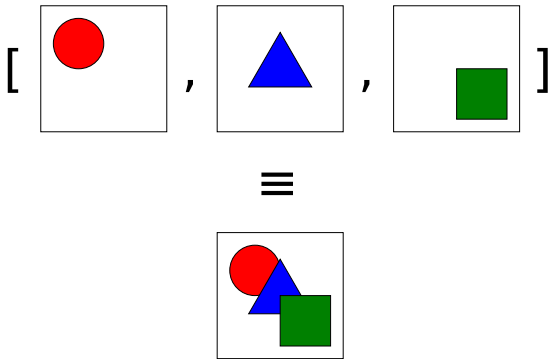


Figure 1. Superimposing a list of primitives

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:

[1] `http://projects.haskell.org/diagrams/`

Haskell '12, September 13, 2012, Copenhagen, Denmark.

- 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 $\diamond :: S \to S \to S$ and a distinguished element $\varepsilon :: S$, subject to the laws

$$\varepsilon \diamond x = x \diamond \varepsilon = x \tag{M1}$$

$$x \diamond (y \diamond z) = (x \diamond y) \diamond z. \tag{M2}$$

where x, y, and z are arbitrary elements of S. That is, ε 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
  (◇) :: a → a → a
  mconcat :: [a] → a
  mconcat = foldr (◇) ε
```

The actual Monoid methods are named *mempty* and *mappend*, but I will use ε 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 = λa → f1 a ◇ f2 a
```

  In fact, if you squint and think of the function type $a \to m$ 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 $a \to a$ 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 $(,)$, $((\to)\ e)$, 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

$$f\ \varepsilon = \varepsilon \tag{H1}$$

$$f\ (x \diamond y) = f\ x \diamond f\ y \tag{H2}$$

For example, $length\ [\,] = 0$ and $length\ (xs +\!\!+ ys) = length\ xs + length\ ys$, 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 $f :: a \to m$, where m is a monoid, extends uniquely to a monoid homomorphism from $[a]$ to m—namely, $mconcat \circ map\ f$. It will be useful later to give this construction a name:

```
hom :: Monoid m ⇒ (a → m) → ([a] → m)
hom f = mconcat ∘ map f
```

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[2]:

```
class Semigroup a where
  (◇) :: a → a → a
```

(The Semigroup class also declares two other methods with default implementations in terms of (◇); 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 ◇ 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 $(\mathbb{N}, min)$ can be thought of as "positive infinity": $min(+\infty, n) = min(n, +\infty) = n$ 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 ◇ b       = b
  a       ◇ Nothing = a
  (Just a) ◇ (Just b) = Just (a ◇ 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 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 ε
  (Diagram ps1) ◇ (Diagram ps2) = Diagram (ps2 ◇ 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[3]). 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 (◇) 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

[2] `http://hackage.haskell.org/package/semigroups`

[3] `http://hackage.haskell.org/package/newtype`

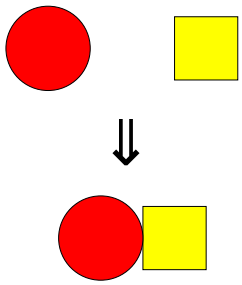


Figure 2. Placing two diagrams beside one another

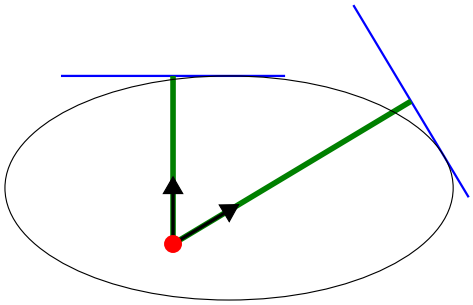


Figure 3. Envelope for an ellipse

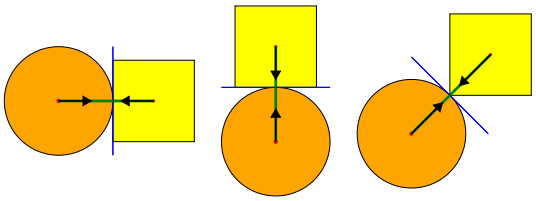


Figure 4. Using envelopes to place diagrams beside one another

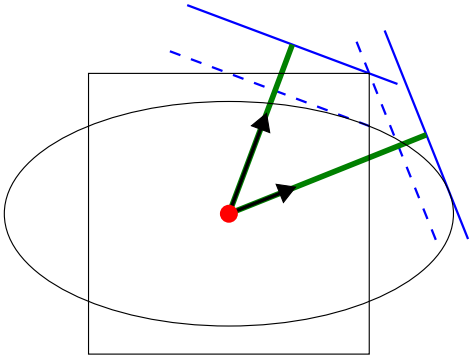


Figure 5. Composing envelopes

get as a result?), and they introduce an inherent left-right-up-down bias—which, though it may be appropriate for something like TEX, is best avoided in a general-purpose drawing library.

An elegant functional solution is something I term an *envelope*.[4] 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 $\mathsf{V}_2 \to \mathbb{R}$.[5] 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 $s \leqslant t$.

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*.

[4] The initial idea for envelopes is due to Sebastian Setzer. See `http://byorgey.wordpress.com/2009/10/28/collecting-attributes/#comment-2030`.

[5] 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.

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 $((\to)\ \mathsf{V}_2)$. 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 $((\to)\ \mathsf{V}_2)$ 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

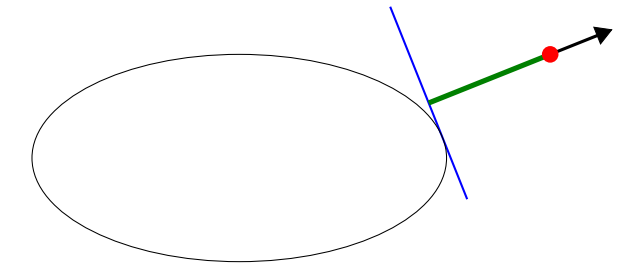


Figure 6. Negative distance as output of an envelope

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 ε.

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 $(const\ 0)$ 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

```
        V2 → Option (Max ℝ)
Option (V2 →        Max ℝ)
```

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 $hom\ f = mconcat \circ map\ f$ expresses the lifting of a function $a \to m$ to a monoid homomorphism $[a] \to m$.

If we assume that there is a function

```
translateP :: V2 → Prim → 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 (Dual [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

$$e = hom\ envelopeP\ ps.$$

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,

$$hom\ envelopeP\ [\,] = \varepsilon.$$

The interesting case is $(\diamond)$. Suppose we have two diagram values Diagram (Dual ps_1, e_1) and Diagram (Dual ps_2, e_2) for which the invariant holds, and we combine them with $(\diamond)$, resulting in Diagram (Dual $(ps_2 +\!\!+ ps_1), e_1 \diamond e_2$). We must show that the invariant is preserved, that is,

$$e_1 \diamond e_2 = hom\ envelopeP\ (ps_2 +\!\!+ ps_1).$$

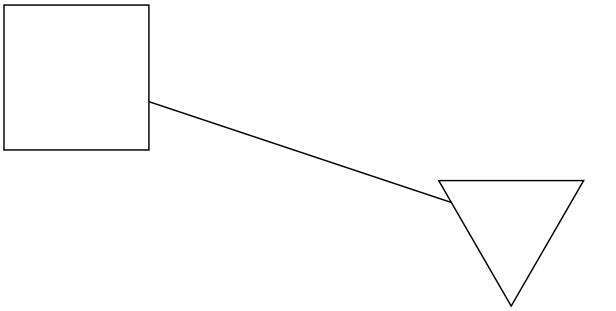


Figure 7. Drawing a line between two shapes

Again, since *hom envelopeP* is a monoid homomorphism,

$$\begin{aligned} &\mathit{hom\ envelopeP}\ (\mathit{ps}_2 +\!\!+ \mathit{ps}_1) \\ &\qquad = \mathit{hom\ envelopeP}\ \mathit{ps}_2 \diamond \mathit{hom\ envelopeP}\ \mathit{ps}_1, \end{aligned}$$

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 $\mathsf{Max}\,\mathbb{R}$ is a commutative semigroup, and $((\to)\ \mathsf{V}_2)$ and Option both preserve commutativity. □

Intuitively, it is precisely the fact that the old version of *envelope* (defined in terms of *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.

$$\textbf{newtype}\ \mathsf{Trace} = \mathsf{Trace}\ (\mathsf{P}_2 \to \mathsf{V}_2 \to \mathbb{R})$$

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

$$\begin{aligned} &\textbf{newtype}\ \mathsf{Trace} = \mathsf{Trace}\ (\mathsf{P}_2 \to \mathsf{V}_2 \to \mathsf{Min}\ \mathbb{R}) \\ &\quad \textbf{deriving}\ (\mathsf{Semigroup}) \end{aligned}$$

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

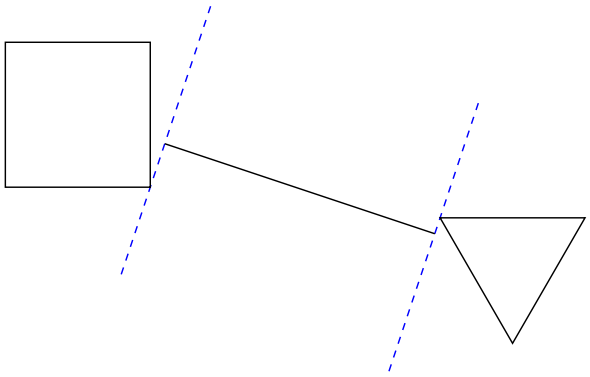


Figure 8. Envelopes are not useful for drawing connecting lines!

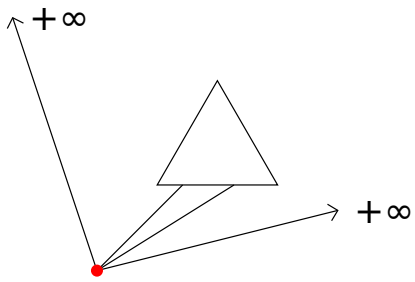


Figure 9. Returning $+\infty$ from a trace

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):

$$\begin{aligned} &\mathsf{P}_2 \to &&\mathsf{V}_2 \to \mathsf{Option}\ (\mathsf{Min}\ \mathbb{R}) \\ &\mathsf{P}_2 \to \mathsf{Option}\ (\mathsf{V}_2 \to &&\mathsf{Min}\ \mathbb{R}) \\ \mathsf{Option}\ (\mathsf{P}_2 \to &&\mathsf{V}_2 \to &\mathsf{Min}\ \mathbb{R}) \end{aligned}$$

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:

$$\begin{aligned} &\textbf{newtype}\ \mathsf{Trace} = \mathsf{Trace}\ (\mathsf{P}_2 \to \mathsf{V}_2 \to \mathsf{Option}\ (\mathsf{Min}\ \mathbb{R})) \\ &\quad \textbf{deriving}\ (\mathsf{Semigroup}, \mathsf{Monoid}) \end{aligned}$$

Assuming there is a function $\mathit{traceP} :: \mathsf{Prim} \to \mathsf{Trace}$ to compute the trace of any primitive, we could define

$$\begin{aligned} &\mathit{trace} :: \mathsf{Diagram} \to \mathsf{Trace} \\ &\mathit{trace} = \mathit{hom\ traceP} \circ \mathit{unD} \end{aligned}$$

However, this is a monoid homomorphism since Trace is also a commutative monoid, so we can cache the trace of each diagram as well.

$$\begin{aligned} &\textbf{newtype}\ \mathsf{Diagram} \\ &\quad = \mathsf{Diagram}\ (\mathsf{Dual}\ [\mathsf{Prim}], \mathsf{Envelope}, \mathsf{Trace}) \\ &\quad \textbf{deriving}\ (\mathsf{Semigroup}, \mathsf{Monoid}) \end{aligned}$$

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 ε is the identity transformation. Given these intuitive semantics, we expect

$$transformP\ \varepsilon\ p = p \tag{1}$$

that is, transforming by the identity transformation has no effect, and

$$transformP\ (t_1 \diamond t_2)\ p = transformP\ t_1\ (transformP\ t_2\ p) \tag{2}$$

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, η-reducing (1) and (2) yields

$$transformP\ \varepsilon = id \tag{1'}$$

$$transformP\ (t_1 \diamond t_2) = transformP\ t_1 \circ transformP\ t_2 \tag{2'}$$

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 $m \Rightarrow$ Action $m\ a$ **where**
  $act :: m \to a \to a$

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 $m_1, m_2 :: m$,

$$act\ \varepsilon = id \tag{MA1}$$

$$act\ (m_1 \diamond m_2) = act\ m_1 \circ act\ m_2 \tag{MA2}$$

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, $map\ id = id$ and $map\ (f \circ g) = map\ f \circ map\ g$.

Proof.

$$\begin{aligned}
&transformD\ \varepsilon \\
=\ &\{\quad \text{definition of } transformD \quad\} \\
&map\ (act\ \varepsilon) \\
=\ &\{\quad \text{Transformation / Prim} \quad\} \\
&map\ id \\
=\ &\{\quad \text{list functor} \quad\} \\
&id
\end{aligned}$$

$$\begin{aligned}
&transformD\ (t_1 \diamond t_2) \\
=\ &\{\quad \text{definition} \quad\} \\
&map\ (act\ (t_1 \diamond t_2)) \\
=\ &\{\quad \text{Transformation / Prim} \quad\} \\
&map\ (act\ t_1 \circ act\ t_2) \\
=\ &\{\quad \text{list functor} \quad\} \\
&map\ (act\ t_1) \circ map\ (act\ t_2) \\
=\ &\{\quad \text{definition} \quad\} \\
&transformD\ t_1 \circ transformD\ t_2
\end{aligned}$$

□

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

$$act\ t\ \varepsilon = \varepsilon, \tag{3}$$

since mapping t over the empty list of primitives results in the empty list again. We also have

$$act\ t\ (d_1 \diamond d_2) = (act\ t\ d_1) \diamond (act\ t\ d_2), \tag{4}$$

since

$$\begin{aligned}
&act\ t\ (d_1 \diamond d_2) \\
=\ &\{\quad \text{definitions of } act \text{ and } (\diamond) \quad\} \\
&map\ (act\ t)\ (d_1 +\!\!+ d_2) \\
=\ &\{\quad \text{naturality of } (+\!\!+) \quad\} \\
&map\ (act\ t)\ d_1 +\!\!+ map\ (act\ t)\ d_2 \\
=\ &\{\quad \text{definition} \quad\} \\
&act\ t\ d_1 \diamond act\ t\ d_2
\end{aligned}$$

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:

$$act\ m\ \varepsilon = \varepsilon \tag{MA3}$$

$$act\ m\ (n_1 \diamond n_2) = act\ m\ n_1 \diamond act\ m\ n_2 \tag{MA4}$$

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

$$act\ t \circ envelopeP = envelopeP \circ act\ t \tag{TE}$$

$$act\ t \circ traceP = traceP \circ act\ t \tag{TT}$$

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
  act t (Diagram (Dual ps, e, tr))
    = Diagram (Dual (map (act t) ps), act t e, 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

$$e = hom\ envelopeP\ ps.$$

We must therefore prove that

$$act\ t\ (hom\ envelopeP\ ps) = hom\ envelopeP\ (map\ (act\ t)\ ps),$$

or, in point-free form,

$$act\ t \circ hom\ envelopeP = hom\ envelopeP \circ map\ (act\ t).$$

Proof. We reason as follows:

$$\begin{aligned}
&act\ t \circ hom\ envelope\ \mathsf{P} \\
=\ &\{\quad \text{definition} \quad\} \\
&act\ t \circ mconcat \circ map\ envelopeP \\
=\ &\{\quad \text{lemma proved below} \quad\} \\
&mconcat \circ map\ (act\ t) \circ map\ envelopeP \\
=\ &\{\quad \text{list functor, (TE)} \quad\} \\
&mconcat \circ map\ envelopeP \circ map\ (act\ t) \\
=\ &\{\quad \text{definition} \quad\} \\
&hom\ envelopeP \circ map\ (act\ t)
\end{aligned}$$

□

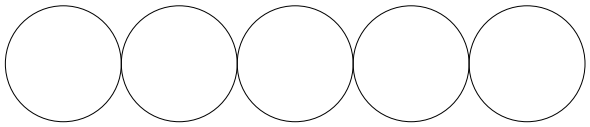


Figure 10. Laying out a line of circles with *beside*

It remains only to prove that $act\ t \circ mconcat = mconcat \circ map\ (act\ t)$. 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 $act\ t \circ mconcat$. If l is the empty list,

$$\begin{aligned}
&act\ t\ (mconcat\ [\,]) \\
=\ &\{\quad mconcat \quad\} \\
&act\ t\ \varepsilon \\
=\ &\{\quad \text{monoid action (MA3)} \quad\} \\
&\varepsilon \\
=\ &\{\quad mconcat\text{, definition of } map \quad\} \\
&mconcat\ (map\ (act\ t)\ [\,])
\end{aligned}$$

In the case that $l = x : xs$,

$$\begin{aligned}
&act\ t\ (mconcat\ (x : xs)) \\
=\ &\{\quad mconcat \quad\} \\
&act\ t\ (x \diamond mconcat\ xs) \\
=\ &\{\quad \text{monoid action (MA4)} \quad\} \\
&act\ t\ x \diamond act\ t\ (mconcat\ xs) \\
=\ &\{\quad \text{induction hypothesis} \quad\} \\
&act\ t\ x \diamond mconcat\ (map\ (act\ t)\ xs) \\
=\ &\{\quad mconcat \quad\} \\
&mconcat\ (act\ t\ x : map\ (act\ t)\ xs) \\
=\ &\{\quad \text{definition of } map \quad\} \\
&mconcat\ (map\ (act\ t)\ (x : xs))
\end{aligned}$$

□

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):

```
unit_x :: V2   — unit vector along the positive x-axis
hcat = foldr (beside unit_x) ε
lineOfCircles n = hcat (replicate n circle)
```

Fully evaluating *lineOfCircles n* takes $O(n^2)$ time, because the kth call to *beside* must map over k primitives, resulting in $1+2+3+\cdots+n$ 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

$$act\ (t_1 \diamond t_2) = act\ t_1 \circ act\ t_2.$$

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' ◇ 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)) ◇ (Diagram (ps2, t2, e2, tr2))
    = Diagram (act t1 ps1 ◇ act t2 ps2,
               ε,
               e1 ◇ e2,
               tr1 ◇ 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 → Diagram
ε    :: Diagram
(◇)  :: Diagram → Diagram → Diagram
act  :: Transformation → Diagram → 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):

```
envelope :: Diagram →                 Envelope
envelope (Prim p)                   = envelopeP p
envelope Empty                      = ε
envelope (Compose (e, _) _ _)       = e
envelope (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 ◇ d = d
  d ◇ Empty = d
  d1 ◇ d2
    = Compose
      (envelope d1 ◇ envelope d2
      , trace      d1 ◇ trace      d2
      )
      d1 d2
instance Monoid Diagram where
  ε = 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 $unD :: \text{Diagram} \to [\text{Prim}]$ "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 ◇ 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:

```
unD :: Diagram →                 [Prim]
unD (Prim p)                   = [p]
unD Empty                      = ε
unD (Compose _ d1 d2)          = unD d2 ◇ unD d1
unD (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,

$$act\ t\ (unD\ d) = go\ t\ d.$$

From this it will follow, by (MA1), that

$$unD\ d = act\ \varepsilon\ (unD\ d) = go\ \varepsilon\ d = unD'\ d.$$

Proof. By induction on d.

- If $d = \mathsf{Prim}\ p$, then $act\ t\ (unD\ (\mathsf{Prim}\ p)) = act\ t\ [p] = [act\ t\ p] = go\ t\ (\mathsf{Prim}\ p)$.
- If $d = \mathsf{Empty}$, then $act\ t\ (unD\ \mathsf{Empty}) = act\ t\ \varepsilon = \varepsilon = go\ t\ \mathsf{Empty}$, where the central equality is (MA3).
- If $d = \mathsf{Compose}\ c\ d_1\ d_2$, then

  $act\ t\ (unD\ (\mathsf{Compose}\ c\ d_1\ d_2))$
  $= \{$ definition $\}$
  $act\ t\ (unD\ d_2 \diamond unD\ d_1)$
  $= \{$ monoid action (MA4) $\}$
  $act\ t\ (unD\ d_2) \diamond act\ t\ (unD\ d_1)$
  $= \{$ induction hypothesis $\}$
  $go\ t\ d_2 \diamond go\ t\ d_1$
  $= \{$ definition $\}$
  $go\ t\ (\mathsf{Compose}\ c\ d_1\ d_2)$

- Finally, if $d = \mathsf{Act}\ t'\ d'$, then

  $act\ t\ (unD\ (\mathsf{Act}\ t'\ d'))$
  $= \{$ definition $\}$
  $act\ t\ (act\ t'\ (unD\ d'))$
  $= \{$ monoid action (MA2) $\}$
  $act\ (t \diamond t')\ (unD\ d')$
  $= \{$ induction hypothesis $\}$
  $go\ (t \diamond t')\ d'$
  $= \{$ definition $\}$
  $go\ t\ (\mathsf{Act}\ t'\ d')$

□

Variation VIII: Difference lists

Actually, *unD′* still suffers from another performance problem hinted at in the previous variation. A right-nested expression like $d_1 \diamond (d_2 \diamond (d_3 \diamond d_4))$ still takes quadratic time to compile, because it results in left-nested calls to $(+\!\!+)$. This can be solved using *difference lists* [Hughes 1986]: the idea is to represent a list $xs :: [a]$ using the function $(xs+\!\!+) :: [a] \to [a]$. 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:

$$(((xs+\!\!+) \circ (ys+\!\!+)) \circ (zs+\!\!+))\ [\,] = xs +\!\!+ (ys +\!\!+ (zs +\!\!+ [\,])).$$

In fact, difference lists arise from viewing

$$(+\!\!+) :: [a] \to ([a] \to [a])$$

itself as a monoid homomorphism, from the list monoid to the monoid of endomorphisms on $[a]$. (H1) states that $(+\!\!+)\ \varepsilon = \varepsilon$, which expands to $(+\!\!+)\ [\,] = id$, that is, $[\,] +\!\!+ xs = xs$, which is true by definition. (H2) states that $(+\!\!+)\ (xs \diamond ys) = (+\!\!+)\ xs \diamond (+\!\!+)\ ys$, which can be rewritten as

$$((xs +\!\!+ ys)+\!\!+) = (xs+\!\!+) \circ (ys+\!\!+).$$

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,

$$(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$$

and it resolves itself into the familiar associativity of $(+\!\!+)$.

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 $(l \to u)$ 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*:

```
getU :: (Action d u, Monoid u) ⇒ MTree d u l → u
getU (Leaf u _)          = u
getU Empty               = ε
getU (Compose u _ _) = u
getU (Act d t)           = act d (getU t)
envelope = fst  ∘ getU
trace    = snd ∘ getU
```

The Semigroup and Action instances are straightforward generalizations of the instances from Variation VII.

```
instance (Action d u, Monoid u)
    ⇒ Semigroup (MTree d u l) where
  Empty ◇ t = t
  t ◇ Empty = t
  t1 ◇ t2     = Compose (getU t1 ◇ getU t2) t1 t2
instance Semigroup d ⇒ Action d (MTree d u l) where
  act _ Empty    = Empty
  act d (Act d' t) = Act (d ◇ d') t
  act d t          = Act d t
```

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.

```
foldMTree :: (Monoid d, Monoid r, Action d r)
          ⇒ (l → r) → MTree d u l → r
foldMTree leaf t = appEndo (go ε t) ε where
  go d (Leaf _ l)          = Endo (act d (leaf l)◇)
  go _ Empty               = ε
  go d (Compose _ t1 t2) = go d t1 ◇ go d t2
  go d (Act d' t)          = go (d ◇ d') t
unD :: Diagram → [Prim]
unD = getDual ∘ foldMTree (Dual ∘ (:[]))
```

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, {Fill Purple, Stroke Red} :: Style might indicate a diagram drawn in red and filled with purple. Style is then an instance of Monoid, with ε corresponding to the Style containing no attributes, and $(\diamond)$ corresponding to right-biased union. For example,

```
{Fill Purple, Stroke Red} ◇ {Stroke Green, Alpha 0.3}
                  = {Fill Purple, Stroke Green, Alpha 0.3}
```

where Stroke Green overrides Stroke Red. We would also expect to have a function

```
applyStyle :: Style → Diagram → 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:

```
type Diagram
  = MTree Transformation (Envelope, Trace) Prim
```

Can we simply replace Transformation with the product monoid (Transformation, 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:

```
instance Action Style Envelope where
  act _ = id
```

In fact, the only other thing missing is an Action instance defining the action of a product monoid. One obvious instance is:

```
instance (Action m1 a, Action m2 a)
    ⇒ Action (m1, m2) a where
  act (m1, m2) = act m1 ∘ act m2
```

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

$$act\ ((m_{11}, m_{21}) \diamond (m_{12}, m_{22})) = act\ (m_{11}, m_{21}) \circ act\ (m_{12}, m_{22}).$$

Beginning with the left-hand side,

$$\begin{aligned} & act\ ((m_{11}, m_{21}) \diamond (m_{12}, m_{22})) \\ = \ & \{ \quad \text{product monoid} \quad \} \\ & act\ (m_{11} \diamond m_{12}, m_{21} \diamond m_{22}) \\ = \ & \{ \quad \text{proposed definition of } act \text{ for pairs} \quad \} \\ & act\ (m_{11} \diamond m_{12}) \circ act\ (m_{21} \diamond m_{22}) \\ = \ & \{ \quad m_1 \,/\, a, m_2 \,/\, a \ \text{(MA2)} \quad \} \\ & act\ m_{11} \circ act\ m_{12} \circ act\ m_{21} \circ act\ m_{22} \end{aligned}$$

But the right-hand side yields

$$\begin{aligned} & act\ (m_{11}, m_{21}) \circ act\ (m_{12}, m_{22}) \\ = \ & \{ \quad \text{proposed definition of } act \quad \} \\ & act\ m_{11} \circ act\ m_{21} \circ act\ m_{12} \circ act\ m_{22} \end{aligned}$$

In general, these will be equal only when $act\ m_{12} \circ act\ m_{21} = act\ m_{21} \circ act\ m_{12}$—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

```
type Diagram = MTree (Transformation, Style)
                     (Envelope, Trace)
                     Prim
```

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

$$\begin{aligned} & hom :: \mathsf{Monoid}\ m \Rightarrow (a \to m) \to ([a] \to m) \\ & hom\ f = mconcat \circ map\ f \end{aligned}$$

We first note that

$$\begin{aligned} & hom\ f\ (x : xs) \\ = \ & \{ \quad \text{definition of } hom \text{ and } map \quad \} \\ & mconcat\ (f\ x : map\ f\ xs) \\ = \ & \{ \quad mconcat \quad \} \\ & f\ x \diamond mconcat\ (map\ f\ xs) \\ = \ & \{ \quad \text{definition of } hom \quad \} \\ & f\ x \diamond hom\ f\ xs \end{aligned}$$

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 $xs = x : xs'$:

$$\begin{aligned} & hom\ f\ ((x : xs') +\!\!+ ys) \\ & \qquad \{ \quad \text{definition of } (+\!\!+) \quad \} \\ = \ & hom\ f\ (x : (xs' +\!\!+ ys)) \\ & \qquad \{ \quad hom \text{ of } (:), \text{ proved above} \quad \} \\ = \ & f\ x \diamond hom\ f\ (xs' +\!\!+ ys) \\ & \qquad \{ \quad \text{induction hypothesis} \quad \} \\ = \ & f\ x \diamond hom\ f\ xs' \diamond hom\ f\ ys \\ & \qquad \{ \quad \text{associativity of } (\diamond) \text{ and } hom \text{ of } (:) \quad \} \\ = \ & (hom\ f\ (x : xs')) \diamond hom\ f\ ys \end{aligned}$$

□

As a corollary, $mconcat\ (xs +\!\!+ ys) = mconcat\ xs +\!\!+ mconcat\ ys$, since $hom\ id = mconcat \circ map\ id = mconcat$.

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:

$$\begin{array}{l} halve :: [\alpha] \to [\alpha] \\ halve\ as = take\ (length\ as\ `div`\ 2)\ as \end{array}$$

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, 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:

$$\begin{array}{l} put_1 :: [\alpha] \to [\alpha] \to [\alpha] \\ put_1\ as\ as' \mid length\ as' == n \\ \qquad\qquad = as' +\!\!+ drop\ n\ as \\ \qquad\qquad\ \ \mathbf{where}\ n = length\ as\ `div`\ 2 \end{array}$$

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:

$$\mathbf{data}\ \mathsf{Tree}\ \alpha = \mathsf{Leaf}\ \alpha \mid \mathsf{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha)$$

$$\begin{array}{l} flatten :: \mathsf{Tree}\ \alpha \to [\alpha] \\ flatten\ (\mathsf{Leaf}\ a) \quad\ = [a] \\ flatten\ (\mathsf{Node}\ t_1\ t_2) = flatten\ t_1 +\!\!+ flatten\ t_2 \end{array}$$

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:

$$\begin{array}{l} put_2 :: \mathsf{Tree}\ \alpha \to [\alpha] \to \mathsf{Tree}\ \alpha \\ put_2\ s\ v = \mathbf{case}\ go\ s\ v\ \mathbf{of}\ (t, []) \to t \\ \quad \mathbf{where}\ go\ (\mathsf{Leaf}\ a) \quad\ (b : bs) = (\mathsf{Leaf}\ b, bs) \\ \qquad\qquad go\ (\mathsf{Node}\ s_1\ s_2)\ bs \quad\ = (\mathsf{Node}\ t_1\ t_2, ds) \\ \qquad\qquad\ \ \mathbf{where}\ (t_1, cs) = go\ s_1\ bs \\ \qquad\qquad\qquad\ \ (t_2, ds) = go\ s_2\ cs \end{array}$$

Finally, consider a function that removes duplicate occurrences of elements from a list, with implementation taken over from a standard library:

$$\begin{array}{l} rmdups :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \\ rmdups = \mathsf{List}.nub \end{array}$$

An appropriate backwards propagation function looks as follows:

$$\begin{array}{l} put_3 :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha] \\ put_3\ s\ v \mid v == \mathsf{List}.nub\ v\ \&\&\ length\ v == length\ s' \\ \qquad\qquad = map\ (fromJust \circ flip\ lookup\ (zip\ s'\ v))\ s \\ \qquad\qquad\ \ \mathbf{where}\ s' = \mathsf{List}.nub\ s \end{array}$$

For example, in a Haskell interpreter:

```
> put_3 "abcbabcbaccba" "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

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

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,

$$\begin{array}{ll} \mathit{bff}\ \mathit{halve} & \equiv \mathit{put}_1 \\ \mathit{bff}\ \mathit{flatten} & \equiv \mathit{put}_2 \\ \mathit{bff}\ \mathit{rmdups} & \equiv \mathit{put}_3\,. \end{array}$$

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 Spyratos 1981) that a forward/backward pair of functions $\mathit{get}/\mathit{put}$ should satisfy the laws

$$\mathit{put}\ s\ (\mathit{get}\ s) \equiv s$$

and, if $\mathit{put}\ s\ v$ is defined,

$$\mathit{get}\ (\mathit{put}\ s\ v) \equiv v\,,$$

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 "abcbabcbaccba" "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 $\mathit{put}\ s\ v$ is not equal to $\mathit{get}\ s$ 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*, $\mathit{bff}_{\mathsf{Eq}}$, and $\mathit{bff}_{\mathsf{Ord}}$, 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 $[\alpha] \to [\alpha]$ without an Eq-constraint. So $\mathit{bff}_{\mathsf{Eq}}$ will be used for *rmdups* and its like, and $\mathit{bff}_{\mathsf{Ord}}$ for functions like the following one:

$$\begin{array}{l} \mathit{top3} :: \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha] \\ \mathit{top3} = \mathit{take}\ 3 \circ \mathsf{List}.\mathit{sort} \circ \mathsf{List}.\mathit{nub} \end{array}$$

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 $\mathit{bff}_{\mathsf{Eq}}$ and $\mathit{bff}_{\mathsf{Ord}}$, 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, $\mathit{bff}_{\mathsf{Eq}}\ \mathit{get}$, or $\mathit{bff}_{\mathsf{Ord}}\ \mathit{get}$ 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 Voigtländer (2004). This simplification is done solely for the sake of exposition, not because of any fundamental problems with doing elsewise.

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

$$\mathit{bff} :: (\forall\alpha.[\alpha] \to [\alpha]) \to (\forall\alpha.[\alpha] \to [\alpha] \to [\alpha])\,.$$

Note that the local universal quantifications over α 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 α. 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* $\mathit{tail} \equiv \mathit{put}$) 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 $tail$ 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 *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 IntMap.*fromAscList* expects a list with integer keys in ascending order and that 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 :: (∀α.[α] → [α]) → (∀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 *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 *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 *sieve* "abcdefg" ≡ "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 *bff* $(\lambda s \to s +\!\!+ s)$ "a" $((\lambda s \to s +\!\!+ s)$ "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 *bff halve* "" "a" is defined, *halve* (*bff halve* "" "a") ≡ "a", but it is not the case that *halve* "" ≡ "a".)

On the other hand, apart from this empty list weirdness we truly have *bff halve* $\equiv put_1$. So it seems we have made a good start, on which to extend in the next section.

3. Correct Bidirectionalization

In order to fix *bff* to adhere to the GetPut law, we need to deal with duplication of list elements. Consider again the source-to-view function $\lambda s \to s +\!\!+ s$. Applied to a template $[0..n]$, it will deliver the template view $[0, \ldots, n, 0, \ldots, n]$. 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 *bff* as follows:

$$bff :: (\forall\alpha.[\alpha] \to [\alpha]) \to (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])$$

That is, the get-function given to *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 *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 $assoc$ 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 Eq gives a definition for $==$ that makes it reflexive, symmetric, and transitive. Then, the following two lemmas hold.

Lemma 1. *For every* $is :: [\mathsf{Int}]$*, type* τ *that is an instance of* Eq*, and* $f :: \mathsf{Int} \to \tau$*, we have*

$$assoc\ is\ (map\ f\ is) \equiv \mathsf{Right}\ h$$

for some $h :: \mathsf{IntMap}\ \tau$ *with*

$$\mathsf{IntMap}.lookup\ i\ h \equiv \mathbf{if}\ elem\ i\ is\ \mathbf{then}\ \mathsf{Just}\ (f\ i)\ \mathbf{else}\ \mathsf{Nothing}$$

for every $i :: \mathsf{Int}$.

Lemma 2. *Let* $is :: [\mathsf{Int}]$*, let* τ *be a type that is an instance of* Eq*, and let* $v :: [\tau]$ *and* $h :: \mathsf{IntMap}\ \tau$*. We have that if*

$$\mathsf{Right}\ h \equiv assoc\ is\ v\,,$$

then

$$map\ (flip\ \mathsf{IntMap}.lookup\ h)\ is == map\ \mathsf{Just}\ v\,.$$

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 $==$ by $\equiv$, because the instance of Eq for τ may very well give $x == y$ for some $x \not\equiv y$. We will continue to be careful about this distinction in what follows. Of course, the instances of Eq used in practice will often have $==$ 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:

$$\begin{aligned}
&bff :: (\forall\alpha.[\alpha] \to [\alpha]) \to (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])\\
&bff\ get = \lambda s\ v \to\\
&\ \ \mathbf{let}\ s' = [0..length\ s - 1]\\
&\qquad\ g = \mathsf{IntMap}.fromAscList\ (zip\ s'\ s)\\
&\qquad\ h = either\ error\ id\ (assoc\ (get\ s')\ v)\\
&\qquad\ h' = \mathsf{IntMap}.union\ h\ g\\
&\ \ \mathbf{in}\ \ seq\ h\ (map\ (fromJust \circ flip\ \mathsf{IntMap}.lookup\ h')\ s')
\end{aligned}$$

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* $get :: \forall\alpha.[\alpha] \to [\alpha]$*, type* τ *that is an instance of* Eq*, and* $s :: [\tau]$*, we have*

$$bff\ get\ s\ (get\ s) \equiv s\,.$$

Proof. By the function definition for *bff* we have

$$\begin{aligned}
&bff\ get\ s\ (get\ s)\\
&\qquad\equiv\\
&seq\ h\ (map\ (fromJust \circ flip\ \mathsf{IntMap}.lookup\ h')\ s')\,,
\end{aligned} \tag{1}$$

where:

$$s' \equiv [0..length\ s - 1] \tag{2}$$

$$g \equiv \mathsf{IntMap}.fromAscList\ (zip\ s'\ s) \tag{3}$$

$$h \equiv either\ error\ id\ (assoc\ (get\ s')\ (get\ s)) \tag{4}$$

$$h' \equiv \mathsf{IntMap}.union\ h\ g\,. \tag{5}$$

Clearly, (2) implies

$$s \equiv map\ (s\,!!)\ s'\,, \tag{6}$$

where the operator !! is used for extracting a list element at a given index position. Thus,

$$get\ s \equiv get\ (map\ (s\,!!)\ s')\,.$$

By a free theorem of Wadler (1989), every $get :: \forall\alpha.[\alpha] \to [\alpha]$ satisfies

$$get \circ map\ f \equiv map\ f \circ get$$

for every choice of f. Thus, in particular,

$$get\ s \equiv map\ (s\,!!)\ (get\ s')\,.$$

Together with (4) and Lemma 1, this gives that h is defined (i.e., not a runtime error) and that for every $i :: \mathsf{Int}$,

$$\begin{aligned}
\mathsf{IntMap}.lookup\ i\ h \equiv \mathbf{if}\ elem\ i\ (get\ s')\ \mathbf{then}\ \mathsf{Just}\ (s\,!!\,i)\\
\mathbf{else}\ \ \mathsf{Nothing}\,.
\end{aligned}$$

Since by (2), (3), and the specification of $\mathsf{IntMap}.fromAscList$, for every $i :: \mathsf{Int}$,

$$\begin{aligned}
\mathsf{IntMap}.lookup\ i\ g \equiv \mathbf{if}\ elem\ i\ s'\ \mathbf{then}\ \mathsf{Just}\ (s\,!!\,i)\\
\mathbf{else}\ \ \mathsf{Nothing}\,,
\end{aligned}$$

we have by (5) and the specification of $\mathsf{IntMap}.union$ that for every $i :: \mathsf{Int}$,

$$\begin{aligned}
\mathsf{IntMap}.lookup\ i\ h' \equiv\ &\mathbf{if}\ elem\ i\ (get\ s')\\
&\mathbf{then}\ \mathsf{Just}\ (s\,!!\,i)\\
&\mathbf{else}\ \ \mathbf{if}\ elem\ i\ s'\ \mathbf{then}\ \mathsf{Just}\ (s\,!!\,i)\\
&\qquad\mathbf{else}\ \ \mathsf{Nothing}\,.
\end{aligned}$$

Together with (1), the definedness of h, and (6), this gives the claim.

Theorem 2. *Let* $get :: \forall\alpha.[\alpha] \to [\alpha]$*, let* τ *be a type that is an instance of* Eq*, and let* $v, s :: [\tau]$*. We have that if* $bff\ get\ s\ v$ *is defined, then*

$$get\ (bff\ get\ s\ v) == v\,.$$

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 $==$, rather than for true semantic equivalence. As mentioned earlier, in practice $==$ 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 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

$$\begin{aligned}
bff :: &(\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha])\\
&\to (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])\,,
\end{aligned}$$

so that it would also accept *get*-functions like the *rmdups* :: Eq $\alpha \Rightarrow [\alpha] \to [\alpha]$ from the introduction, this would be inviting disaster:

```
> bff rmdups "abcbabcbaccba" "aBc"
"*** Exception: Update changes the length.
> bff rmdups "abcbabcbaccba" "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* "abcbabcbaccba" ≡ "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 "abcbabcbaccba". 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 get :: Eq $\alpha \Rightarrow [\alpha] \to [\alpha]$ 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 $get :: [\alpha] \to [\alpha]$ 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 "abcbabcbaccba" 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 $number_{\mathsf{Eq}}$ 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.

```
template_Eq :: Eq α ⇒ [α] → ([Int], IntMapEq α)
template_Eq s = case runState (go s) (IntMapEq.empty, 0)
                  of   (s', (g, _)) → (s', g)
 where go []       = return []
       go (a : as) = do i ← number_Eq a
                        is ← go as
                        return (i : is)

number_Eq :: Eq α ⇒ α → State (IntMapEq α, Int) Int
number_Eq 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
```

```
empty      :: IntMapEq α
insert     :: Int → α → IntMapEq α → IntMapEq α
checkInsert :: Eq α ⇒ Int → α → IntMapEq α
                     → Either String (IntMapEq α)
union      :: Eq α ⇒ IntMapEq α → IntMapEq α
                     → Either String (IntMapEq α)
lookup     :: Int → IntMapEq α → Maybe α
lookupR    :: Eq α ⇒ α → IntMapEq α → Maybe Int
```

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 τ be a type that is an instance of* Eq *and let $s :: [\tau]$, $s' ::$ [Int], and $g ::$ IntMapEq τ. We have that if*

$$(s', g) \equiv template_{\mathsf{Eq}}\ s\,,$$

then

- map (*flip* IntMapEq.$lookup\ g$) $s' == map$ Just s,
- *for every* $i ::$ Int *not in* s', IntMapEq.$lookup\ i\ g \equiv$ Nothing,
- *for every* $i /= j$ *in* s',

$$\mathsf{IntMapEq}.lookup\ i\ g /= \mathsf{IntMapEq}.lookup\ j\ g\,.$$

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 τ) values by g is very essential. The proofs of both Theorems 1 and 2 use the free theorem $get \circ map\ f \equiv map\ f \circ get$. But that was for $get :: [\alpha] \to [\alpha]$. For the get :: Eq $\alpha \Rightarrow [\alpha] \to [\alpha]$ 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 IntMapEqs under use. This is why both IntMapEq.*checkInsert* and IntMapEq.*union* have Either String (IntMapEq α) 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 $number_{\mathsf{Eq}}$, on the other hand, has no such safeguards. But Lemma 3 tells us that everything is still okay with $template_{\mathsf{Eq}}$.

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" ≡ "aaa" disaster with its violation of the PutGet law). The variant of *assoc* to use is then as follows:

```
assoc_Eq :: Eq α ⇒ [Int] → [α] → Either String (IntMapEq α)
assoc_Eq []       []       = Right IntMapEq.empty
assoc_Eq (i : is) (b : bs) = either Left
                                    (IntMapEq.checkInsert i b)
                                    (assoc_Eq is bs)
assoc_Eq _        _        = Left "Update changes the length."
```

For it, we claim the following two lemmas. The notion of a function $f :: \mathsf{Int} \to \tau$, for a type τ that is an instance of Eq, being injective on a list $is :: [\mathsf{Int}]$ is defined as "for every $i /= j$ in is, also $f\ i /= f\ j$".

Lemma 4. *Let $is :: [\mathsf{Int}]$, let τ be a type that is an instance of Eq, and let $f :: \mathsf{Int} \to \tau$ and $v :: [\tau]$. We have that if $map\ f\ is == v$ and f is injective on is, then*

$$assoc_{\mathsf{Eq}}\ is\ v \equiv \mathsf{Right}\ h$$

for some $h :: \mathsf{IntMapEq}\ \tau$ with

$$\mathsf{IntMapEq}.lookup\ i\ h == \mathbf{if}\ elem\ i\ is\ \mathbf{then}\ \mathsf{Just}\ (f\ i)\ \mathbf{else}\ \mathsf{Nothing}$$

for every $i :: \mathsf{Int}$.

Lemma 5. *Let $is :: [\mathsf{Int}]$, let τ be a type that is an instance of Eq, and let $v :: [\tau]$ and $h :: \mathsf{IntMapEq}\ \tau$. We have that if*

$$\mathsf{Right}\ h \equiv assoc_{\mathsf{Eq}}\ is\ v\,,$$

then

- $map\ (flip\ \mathsf{IntMapEq}.lookup\ h)\ is == map\ \mathsf{Just}\ v$,
- *for every $i :: \mathsf{Int}$ not in is,* $\mathsf{IntMapEq}.lookup\ i\ h \equiv \mathsf{Nothing}$,
- *flip* $\mathsf{IntMapEq}.lookup\ h$ *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:

$$\begin{aligned}
&bff_{\mathsf{Eq}} :: (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]) \\
&\qquad \to (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha]) \\
&bff_{\mathsf{Eq}}\ get = \lambda s\ v \to \\
&\ \mathbf{let}\ (s', g) = template_{\mathsf{Eq}}\ s \\
&\qquad h\ \ = either\ error\ id\ (assoc_{\mathsf{Eq}}\ (get\ s')\ v) \\
&\qquad h' = either\ error\ id\ (\mathsf{IntMapEq}.union\ h\ g) \\
&\ \mathbf{in}\ \ seq\ h'\ (map\ (fromJust \circ flip\ \mathsf{IntMapEq}.lookup\ h')\ s')
\end{aligned}$$

Let us do some sanity checks:

```
> bff_Eq rmdups "abcbabcbaccba" "aBc"
"aBcBaBcBaccBa"
> bff_Eq rmdups "abcbabcbaccba" "abc"
"abcbabcbaccba"
> 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 $bff_{\mathsf{Eq}}\ rmdups \equiv put_3$ for put_3 as given in the introduction (except that $bff_{\mathsf{Eq}}\ rmdups$ gives more meaningful error messages).

A small, but important, detail in the definition of bff_{Eq} is that the computation of h' via $\mathsf{IntMapEq}.union$ 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) "abcbabcbaccba" "ba"
"*** Exception: Update violates differentness.
```

Note that the view obtained from "abcbabcbaccba" by applying $tail \circ rmdups$ 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 "abababababaaaba". And this would contradict the PutGet law, because $tail \circ rmdups$ applied to "abababababaaaba" 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 $\mathsf{IntMapEq}.union$ 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 $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$. 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 τ_1 and τ_2 that are instances of Eq, relation $\mathcal{R}$ between them that respects 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 $get\ l_1$ and $get\ l_2$ are also of the same length and element-wise related by $\mathcal{R}$. The notion of $\mathcal{R}$ respecting 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 $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$, let τ be a type that is an instance of Eq, and let $f :: \mathsf{Int} \to \tau$, $s' :: [\mathsf{Int}]$, and $s :: [\tau]$. We have that if $map\ f\ s' == s$ and f is injective on s', then $map\ f\ (get\ s') == get\ s$ and every i in $get\ s'$ is also in s'.*

The relation $\mathcal{R}$ used for this specialization is the one which contains exactly all pairs (i, a) with $i :: \mathsf{Int}$, $a :: \tau$, i in s', and $f\ i == a$.

Now, we can go about proving the GetPut and PutGet laws for bff_{Eq}. The former is now also established only up to $==$.

Theorem 3. *For every $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$, type τ that is an instance of Eq, and $s :: [\tau]$, we have*

$$bff_{\mathsf{Eq}}\ get\ s\ (get\ s) == s\,.$$

Theorem 4. *Let $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$, let τ be a type that is an instance of Eq, and let $v, s :: [\tau]$. We have that if $bff_{\mathsf{Eq}}\ get\ s\ v$ is defined, then*

$$get\ (bff_{\mathsf{Eq}}\ get\ s\ v) == v\,.$$

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

$$\begin{aligned}
&bff_{\mathsf{Ord}} :: (\forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]) \\
&\qquad \to (\forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])\ ?
\end{aligned}$$

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 $template_{\mathsf{Eq}}$ 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:

$$\begin{aligned}&\textbf{class}\ \mathsf{Functor}\ \phi \Rightarrow \mathsf{Applicative}\ \phi\ \textbf{where}\\ &\ \mathit{pure} :: \alpha \to \phi\ \alpha\\ &\ (<\!*\!>) :: \phi\ (\alpha \to \beta) \to \phi\ \alpha \to \phi\ \beta\end{aligned}$$

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):

$$\begin{aligned}&\mathit{collect} :: \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to \mathsf{Const}\ (\mathsf{Set}\ \alpha)\ [\beta]\\ &\mathit{collect}\ s = \mathit{traverse}\ (\lambda a \to \mathsf{Const}\ (\mathsf{Set}.\mathit{singleton}\ a))\ s\\ \\ &\mathit{traverse} :: \mathsf{Applicative}\ \phi \Rightarrow (\alpha \to \phi\ \beta) \to [\alpha] \to \phi\ [\beta]\\ &\mathit{traverse}\ f\ [\,] \quad\ = \mathit{pure}\ [\,]\\ &\mathit{traverse}\ f\ (a : as) = \mathit{pure}\ (:)\ <\!*\!>\ f\ a\ <\!*\!>\ \mathit{traverse}\ f\ as\end{aligned}$$

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 $\mathit{fromAscPairList}$ expects a list with both keys and values in ascending order. Together with the function $\mathsf{Set}.\mathit{toAscList}$ that transforms a set into a sorted list, we can define

$$\begin{aligned}&\mathit{set2map} :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Set}\ \alpha \to \mathsf{IntMapOrd}\ \alpha\\ &\mathit{set2map}\ \mathit{as} =\\ &\ \mathsf{IntMapOrd}.\mathit{fromAscPairList}\ (\mathit{zip}\ [0..]\ (\mathsf{Set}.\mathit{toAscList}\ \mathit{as}))\end{aligned}$$

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:[1]

$$\begin{aligned}&\mathit{propagate} :: \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to ((\to)\ (\mathsf{IntMapOrd}\ \alpha))\ [\mathsf{Int}]\\ &\mathit{propagate}\ s =\\ &\ \mathit{traverse}\ (\lambda a \to \mathit{fromJust} \circ \mathsf{IntMapOrd}.\mathit{lookupR}\ a)\ s\end{aligned}$$

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:

$$\begin{aligned}&\mathit{template}_{\mathsf{Ord}} :: \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to ([\mathsf{Int}], \mathsf{IntMapOrd}\ \alpha)\\ &\mathit{template}_{\mathsf{Ord}}\ s = \textbf{case}\ \mathit{traverse}\ \mathit{number}_{\mathsf{Ord}}\ s\ \textbf{of}\\ &\qquad\qquad \mathsf{Lift}\ (\mathsf{Const}\ \mathit{as}, f) \to \textbf{let}\ m = \mathit{set2map}\ \mathit{as}\\ &\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ \ (f\ m, m)\\ \\ &\mathit{number}_{\mathsf{Ord}} :: \mathsf{Ord}\ \alpha \Rightarrow \alpha \to \mathsf{Lift}\ (,)\ (\mathsf{Const}\ (\mathsf{Set}\ \alpha))\\ &\qquad\qquad\qquad\qquad\qquad\qquad ((\to)\ (\mathsf{IntMapOrd}\ \alpha))\ \mathsf{Int}\\ &\mathit{number}_{\mathsf{Ord}}\ a = \mathsf{Lift}\ (\mathsf{Const}\ (\mathsf{Set}.\mathit{singleton}\ a),\\ &\qquad\qquad\qquad\qquad \mathit{fromJust} \circ \mathsf{IntMapOrd}.\mathit{lookupR}\ a)\end{aligned}$$

Note that $\mathit{number}_{\mathsf{Ord}}$, which serves as argument to $\mathit{traverse}$ in the definition of $\mathit{template}_{\mathsf{Ord}}$, is essentially obtained as a "split"

[1] Note that the type of $\mathit{propagate}$ could equivalently be written as follows: $\mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to \mathsf{IntMapOrd}\ \alpha \to [\mathsf{Int}]$.

$$\begin{aligned}&\mathit{fromAscPairList} :: \mathsf{Ord}\ \alpha \Rightarrow [(\mathsf{Int}, \alpha)] \to \mathsf{IntMapOrd}\ \alpha\\ &\mathit{empty} \qquad\quad :: \mathsf{IntMapOrd}\ \alpha\\ &\mathit{checkInsert} \quad :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Int} \to \alpha \to \mathsf{IntMapOrd}\ \alpha\\ &\qquad\qquad\qquad\qquad \to \mathsf{Either\ String}\ (\mathsf{IntMapOrd}\ \alpha)\\ &\mathit{union} \qquad\quad :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{IntMapOrd}\ \alpha \to \mathsf{IntMapOrd}\ \alpha\\ &\qquad\qquad\qquad\qquad \to \mathsf{Either\ String}\ (\mathsf{IntMapOrd}\ \alpha)\\ &\mathit{lookup} \qquad\quad :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Int} \to \mathsf{IntMapOrd}\ \alpha \to \mathsf{Maybe}\ \alpha\\ &\mathit{lookupR} \qquad :: \mathsf{Ord}\ \alpha \Rightarrow \alpha \to \mathsf{IntMapOrd}\ \alpha \to \mathsf{Maybe\ Int}\end{aligned}$$

Figure 3. Functions from module IntMapOrd.

of the corresponding arguments in the definitions of $\mathit{collect}$ and $\mathit{propagate}$ 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 $f :: \mathsf{Int} \to \tau$, for a type τ that is an instance of Ord, being order-preserving on a list $s' :: [\mathsf{Int}]$ is defined as "for every $i < j$ in s', also $f\ i < f\ j$".

Lemma 7. *Let τ be a type that is an instance of* Ord *and let $s :: [\tau]$, $s' :: [\mathsf{Int}]$, and $g :: \mathsf{IntMapOrd}\ \tau$. We have that if*

$$(s', g) \equiv \mathit{template}_{\mathsf{Ord}}\ s\,,$$

then

- $\mathit{map}\ (\mathit{flip}\ \mathsf{IntMapOrd}.\mathit{lookup}\ g)\ s' == \mathit{map}\ \mathsf{Just}\ s$,
- *for every* $i :: \mathsf{Int}$ *not in* s', $\mathsf{IntMapOrd}.\mathit{lookup}\ i\ g \equiv \mathsf{Nothing}$,
- $\mathit{flip}\ \mathsf{IntMapOrd}.\mathit{lookup}\ g$ *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 $\mathit{assoc}_{\mathsf{Eq}}$ to $\mathit{assoc}_{\mathsf{Ord}}$ are almost trivial:

$$\begin{aligned}&\mathit{assoc}_{\mathsf{Ord}} :: \mathsf{Ord}\ \alpha \Rightarrow [\mathsf{Int}] \to [\alpha] \to \mathsf{Either\ String}\ (\mathsf{IntMapOrd}\ \alpha)\\ &\mathit{assoc}_{\mathsf{Ord}}\ [\,] \quad\ [\,] \quad\ = \mathsf{Right}\ \mathsf{IntMapOrd}.\mathit{empty}\\ &\mathit{assoc}_{\mathsf{Ord}}\ (i : is)\ (b : bs) = \mathit{either}\ \mathsf{Left}\\ &\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{IntMapOrd}.\mathit{checkInsert}\ i\ b)\\ &\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{assoc}_{\mathsf{Ord}}\ \mathit{is}\ \mathit{bs})\\ &\mathit{assoc}_{\mathsf{Ord}}\ \_ \quad\ \_ \quad\ = \mathsf{Left}\ \text{"Update changes the length."}\end{aligned}$$

and analogues of Lemmas 4 and 5 for $\mathit{assoc}_{\mathsf{Ord}}$ instead of $\mathit{assoc}_{\mathsf{Eq}}$ 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:

$$\begin{aligned}&\mathit{bff}_{\mathsf{Ord}} :: (\forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha])\\ &\qquad \to (\forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])\\ &\mathit{bff}_{\mathsf{Ord}}\ \mathit{get} = \lambda s\ v \to\\ &\ \textbf{let}\ (s', g) = \mathit{template}_{\mathsf{Ord}}\ s\\ &\qquad h \quad\ = \mathit{either}\ \mathit{error}\ \mathit{id}\ (\mathit{assoc}_{\mathsf{Ord}}\ (\mathit{get}\ s')\ v)\\ &\qquad h' \quad\ = \mathit{either}\ \mathit{error}\ \mathit{id}\ (\mathsf{IntMapOrd}.\mathit{union}\ h\ g)\\ &\ \textbf{in}\ \ \mathit{seq}\ h'\ (\mathit{map}\ (\mathit{fromJust} \circ \mathit{flip}\ \mathsf{IntMapOrd}.\mathit{lookup}\ h')\ s')\end{aligned}$$

Showing off its power, for the function $\mathit{top3}$ 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_{Ord}, we need an appropriate free theorem that holds for every function of type $get :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]$. Again consulting the online free theorems generator `http://linux.tcs.inf.tu-dresden.de/~voigt/ft`, we obtain that for every choice of types τ_1 and τ_2 that are instances of Ord, relation $\mathcal{R}$ between them that respects 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 $get\ l_1$ and $get\ l_2$ are also of the same length and element-wise related by $\mathcal{R}$. The notion of $\mathcal{R}$ respecting 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 Ord type class relate to the definitions for $==$ and $<$ in the natural way). Setting $\tau_1 = \mathsf{Int}$, $\tau_2 = \tau$, $\mathcal{R} = \{(i, a) \mid elem\ i\ s'\ \&\&\ f\ i == a\}$, $l_1 = s'$, and $l_2 = s$, we obtain the following specialized version.

Lemma 8. *Let $get :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]$, let τ be a type that is an instance of* Ord*, and let $f :: \mathsf{Int} \to \tau$, $s' :: [\mathsf{Int}]$, and $s :: [\tau]$. We have that if $map\ f\ s' == s$ and f is order-preserving on s', then $map\ f\ (get\ s') == get\ s$ and every i in $get\ s'$ 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 $get :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]$, type τ that is an instance of* Ord*, and $s :: [\tau]$, we have*

$$bff_{\mathsf{Ord}}\ get\ s\ (get\ s) == s\,.$$

Theorem 6. *Let $get :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]$, let τ be a type that is an instance of* Ord*, and let $v, s :: [\tau]$. We have that if $bff_{\mathsf{Ord}}\ get\ s\ v$ is defined, then*

$$get\ (bff_{\mathsf{Ord}}\ get\ s\ v) == v\,.$$

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 get-functions. Let us start with the input side, and with bff_{Ord}.

Where in the definition of bff_{Ord} is the fact important that the input data structure is a list? The answer is: at two places; once in the definition of $traverse$ as used in $template_{\mathsf{Ord}}$ and thus in bff_{Ord}, and once when using map inside bff_{Ord} itself. But note that even though we have provided our own definition of $traverse$ 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:

$$\begin{aligned} traverse :: &(\mathsf{Applicative}\ \phi, \mathsf{Traversable}\ \kappa) \\ &\Rightarrow (\alpha \to \phi\ \beta) \to \kappa\ \alpha \to \phi\ (\kappa\ \beta)\,. \end{aligned}$$

Moreover, there is also a predefined instance of Traversable for the type of lists, and the definition for $traverse$ 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 $template_{\mathsf{Ord}}$ the following more general type, without changing anything about its definition:

$$\begin{aligned} template_{\mathsf{Ord}} :: &(\mathsf{Traversable}\ \kappa, \mathsf{Ord}\ \alpha) \\ &\Rightarrow \kappa\ \alpha \to (\kappa\ \mathsf{Int}, \mathsf{IntMapOrd}\ \alpha)\,. \end{aligned}$$

Providing an instance definition for the data type Tree from the introduction as follows:

$$\begin{aligned} &\mathbf{instance}\ \mathsf{Traversable\ Tree}\ \mathbf{where} \\ &\quad traverse\ f\ (\mathsf{Leaf}\ a) \quad\ = pure\ \mathsf{Leaf}\ <\!\!*\!\!>\ f\ a \\ &\quad traverse\ f\ (\mathsf{Node}\ t_1\ t_2) = pure\ \mathsf{Node}\ <\!\!*\!\!>\ traverse\ f\ t_1 \\ &\qquad\qquad\qquad\qquad\qquad\quad <\!\!*\!\!>\ traverse\ f\ t_2 \end{aligned}$$

we then have, for example:

```
> template_Ord (Node (Leaf 'a') (Leaf 'b'))
(Node (Leaf 0) (Leaf 1),fromList [(0,'a'),(1,'b')])
```

Actually, for dependency reasons, we also need to add the following two instance definitions:

$$\begin{aligned} &\mathbf{instance}\ \mathsf{Foldable\ Tree}\ \mathbf{where} \\ &\quad foldMap = foldMapDefault \\ \\ &\mathbf{instance}\ \mathsf{Functor\ Tree}\ \mathbf{where} \\ &\quad fmap = fmapDefault \end{aligned}$$

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

$$\$(\ derive\ makeTraversable\ ''\mathsf{Tree}\ )$$

Back to bff_{Ord} itself. Since every Traversable is also a Functor, it now suffices to replace map by

$$fmap :: \mathsf{Functor}\ \kappa \Rightarrow (\beta \to \alpha) \to \kappa\ \beta \to \kappa\ \alpha$$

in bff_{Ord}'s definition to allow a generalization of its type as well:

$$\begin{aligned} bff_{\mathsf{Ord}} :: &\mathsf{Traversable}\ \kappa \Rightarrow (\forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow \kappa\ \alpha \to [\alpha]) \\ &\to (\forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow \kappa\ \alpha \to [\alpha] \to \kappa\ \alpha)\,. \end{aligned}$$

This means that we can now bidirectionalize functions of type $get :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow \kappa\ \alpha \to [\alpha]$ for any instance κ of Traversable, not just for lists. For example, we can use bff_{Ord} on functions $get :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Tree}\ \alpha \to [\alpha]$ just as well.

Can we profit from the same kind of genericity also for bff_{Eq} and bff? Concentrating on bff_{Eq} first, it seems that we cannot readily use the generic $traverse$, because $template_{\mathsf{Eq}}$ 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 $number_{\mathsf{Eq}}$ we can rewrite $template_{\mathsf{Eq}}$ as follows:

$$\begin{aligned} &template_{\mathsf{Eq}} :: (\mathsf{Traversable}\ \kappa, \mathsf{Eq}\ \alpha) \\ &\qquad\qquad \Rightarrow \kappa\ \alpha \to (\kappa\ \mathsf{Int}, \mathsf{IntMapEq}\ \alpha) \\ &template_{\mathsf{Eq}}\ s = \mathbf{case}\ runState\ (go\ s)\ (\mathsf{IntMapEq}.empty, 0) \\ &\qquad\qquad\quad \mathbf{of}\quad (s', (g, \_)) \to (s', g) \\ &\ \mathbf{where}\ go = unwrapMonad \\ &\qquad\qquad\quad \circ traverse\ (\mathsf{WrapMonad} \circ number_{\mathsf{Eq}}) \end{aligned}$$

and then obtain a generic bidirectionalizer

$$\begin{aligned} bff_{\mathsf{Eq}} :: &\mathsf{Traversable}\ \kappa \Rightarrow (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow \kappa\ \alpha \to [\alpha]) \\ &\to (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow \kappa\ \alpha \to [\alpha] \to \kappa\ \alpha) \end{aligned}$$

simply by replacing map by $fmap$ in the definition of bff_{Eq}.

And even for *bff* we can replace the template generation via $s' = [0..length\ s - 1]$ and $g = \mathsf{IntMap}.fromAscList\ (zip\ s'\ s)$ by a more streamlined one amenable to Traversable. Again we use a state monad, wrapped up as an applicative functor. In full:[2]

```
bff :: Traversable κ ⇒ (∀α. κ α → [α])
                     → (∀α. Eq α ⇒ κ α → [α] → κ α)
bff get = λs v →
  let (s', g) = template s
      h       = either error id (assoc (get s') v)
      h'      = IntMap.union h g
  in  seq h (fmap (fromJust ∘ flip IntMap.lookup h') s')

template :: Traversable κ ⇒ κ α → (κ Int, IntMap α)
template s =
  case runState (go s) ([], 0)
  of   (s', (l, _)) → (s', IntMap.fromAscList (reverse l))
  where go = unwrapMonad
                ∘ traverse (WrapMonad ∘ number)

number :: α → State ([(Int, α)], Int) Int
number a = do (l, i) ← State.get
              State.put ((i, a) : l, i + 1)
              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, $bff\ flatten \equiv put_2$.

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*, $assoc_{\mathsf{Eq}}$, and $assoc_{\mathsf{Ord}}$ 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 κ where
  tryZip :: κ α → κ β → Either String (κ (α, β))
```

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."
```

[2] 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 κ ⇒ (α → β → β) → β
                                  → κ α → β
```

Using it, we get for example:

```
assoc :: (Zippable κ, Foldable κ, Eq α)
      ⇒ κ Int → κ α → Either String (IntMap α)
assoc = makeAssoc checkInsert IntMap.empty

makeAssoc checkInsert empty s'' v =
  either Left f (tryZip s'' v)
  where f = Data.Foldable.foldr
              (either Left ∘ uncurry checkInsert)
              (Right empty)
```

Then we can change the type of *bff* into

```
bff :: (Traversable κ, Zippable κ', Foldable κ')
    ⇒ (∀α. κ α → κ' α)
      → (∀α. Eq α ⇒ κ α → κ' α → κ α)
```

without having to change anything at all about the function's current definition. Analogously, with

```
assocEq :: (Zippable κ, Foldable κ, Eq α)
        ⇒ κ Int → κ α → Either String (IntMapEq α)
assocEq = makeAssoc IntMapEq.checkInsert
                    IntMapEq.empty
```

and

```
assocOrd :: (Zippable κ, Foldable κ, Ord α)
         ⇒ κ Int → κ α → Either String (IntMapOrd α)
assocOrd = makeAssoc IntMapOrd.checkInsert
                     IntMapOrd.empty
```

and without any changes to the current function definitions of bff_{Eq} and bff_{Ord}, 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 κ' 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*, bff_{Eq}, and bff_{Ord} 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.[3]

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

[3] 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 $put \equiv bff\ get$, $put \equiv bff_{\mathsf{Eq}}\ get$, or $put \equiv bff_{\mathsf{Ord}}\ get$, we have that if $put\ s\ v$ and $put\ (put\ s\ v)\ v'$ are defined, then

$$put\ (put\ s\ v)\ v' == put\ s\ v'\,.$$

And undoability is also a given; i.e., if $put\ s\ v$ is defined, then

$$put\ (put\ s\ v)\ (get\ s) == s\,.$$

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 put_1 would be possible by weakening the condition

$$length\ as' == n$$

to

$$length\ as' == n\ ||\ odd\ (length\ as)\ \&\&\ length\ as' == n+1\,.$$

Our technique does not detect this, i.e., $bff\ halve$ is semantically equivalent to the original version of 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.

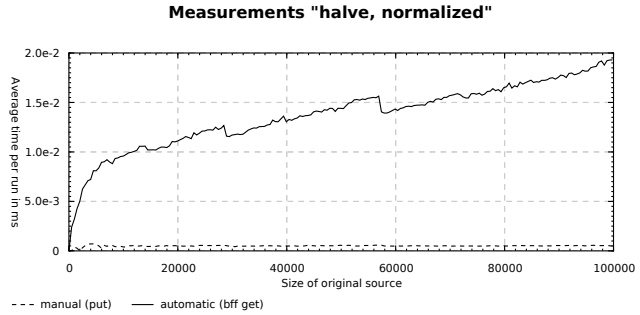


Figure 4. Evaluation of put_1 vs. *bff halve*.

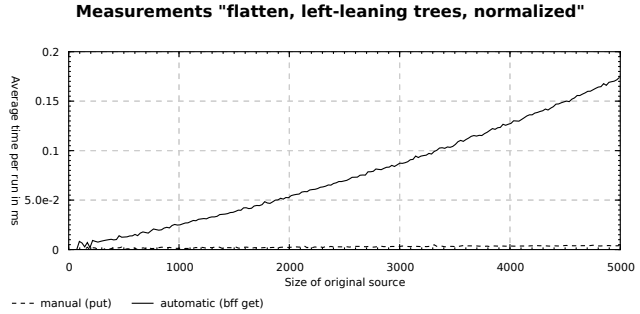


Figure 5. Evaluation of put_2 vs. *bff flatten*, on nasty input.

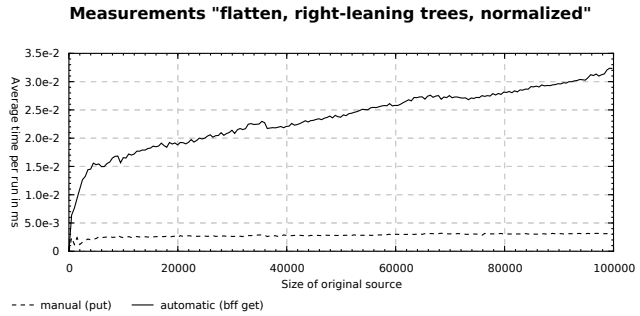


Figure 6. Evaluation of put_2 vs. *bff flatten*, on nice input.

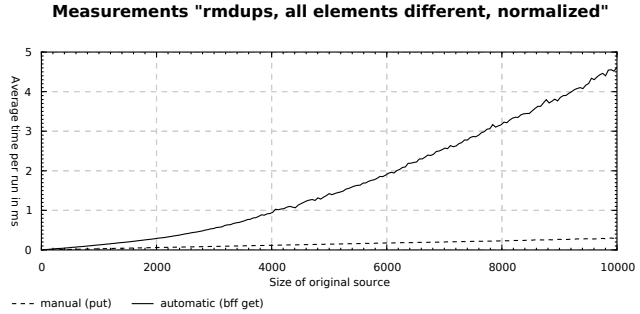


Figure 7. Evaluation of put_3 vs. bff_{Eq} *rmdups*.

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. Spyratos. 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 $get :: \forall\alpha.[\alpha] \to [\alpha]$, let τ be a type that is an instance of Eq, and let $v, s :: [\tau]$. If $\mathit{bff}\ get\ s\ v$ is defined, then we necessarily have

$$\mathit{bff}\ get\ s\ v \equiv \mathit{map}\ f\ s'\,,$$

where

$$\mathsf{Right}\ h \equiv \mathit{assoc}\ (\mathit{get}\ s')\ v \tag{7}$$

$$h' \equiv \mathsf{IntMap}.\mathit{union}\ h\ g \tag{8}$$

$$f \equiv \mathit{fromJust} \circ \mathit{flip}\ \mathsf{IntMap}.\mathit{lookup}\ h' \tag{9}$$

(and the values of s' and g are unimportant in what follows). Thus, by the free theorem mentioned in the proof of Theorem 1,

$$\mathit{get}\ (\mathit{bff}\ get\ s\ v) \equiv \mathit{map}\ f\ (\mathit{get}\ s')\,. \tag{10}$$

By (7) and Lemma 2 we have

$$\mathit{map}\ (\mathit{flip}\ \mathsf{IntMap}.\mathit{lookup}\ h)\ (\mathit{get}\ s') == \mathit{map}\ \mathsf{Just}\ v\,. \tag{11}$$

In particular, for every i in $get\ s'$, we have $\mathsf{IntMap}.\mathit{lookup}\ i\ h \equiv \mathsf{Just}\ b$ for some $b :: \tau$. But then by (8) and the specifications of $\mathsf{IntMap}.\mathit{union}$ and $\mathsf{IntMap}.\mathit{lookup}$,

$$\begin{gathered}\mathit{map}\ (\mathit{flip}\ \mathsf{IntMap}.\mathit{lookup}\ h')\ (\mathit{get}\ s') \\ \equiv \\ \mathit{map}\ (\mathit{flip}\ \mathsf{IntMap}.\mathit{lookup}\ h)\ (\mathit{get}\ s')\,.\end{gathered}$$

Together with (9), the well-known anti-fusion law $\mathit{map}\ (f_1 \circ f_2) \equiv \mathit{map}\ f_1 \circ \mathit{map}\ f_2$, and (11), this implies

$$\mathit{map}\ f\ (\mathit{get}\ s') == \mathit{map}\ \mathit{fromJust}\ (\mathit{map}\ \mathsf{Just}\ v)\,,$$

which gives

$$\mathit{get}\ (\mathit{bff}\ get\ s\ v) == v$$

by (10).

B. Proof of Theorem 3

Let $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$, let τ be a type that is an instance of Eq, and let $s :: [\tau]$. By the function definition for $\mathit{bff}_{\mathsf{Eq}}$ we have

$$\begin{gathered}\mathit{bff}_{\mathsf{Eq}}\ get\ s\ (get\ s) \\ \equiv \\ \mathit{seq}\ h'\ (\mathit{map}\ (\mathit{fromJust} \circ \mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ h')\ s')\,,\end{gathered} \tag{12}$$

where:

$$(s', g) \equiv \mathit{template}_{\mathsf{Eq}}\ s \tag{13}$$

$$h \equiv \mathit{either}\ \mathit{error}\ \mathit{id}\ (\mathit{assoc}_{\mathsf{Eq}}\ (\mathit{get}\ s')\ (\mathit{get}\ s)) \tag{14}$$

$$h' \equiv \mathit{either}\ \mathit{error}\ \mathit{id}\ (\mathsf{IntMapEq}.\mathit{union}\ h\ g)\,. \tag{15}$$

By (13) and Lemma 3, we have

$$\mathit{map}\ (\mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ g)\ s' == \mathit{map}\ \mathsf{Just}\ s\,, \tag{16}$$

as well as that

- for every $i :: \mathsf{Int}$ not in s', $\mathsf{IntMapEq}.\mathit{lookup}\ i\ g \equiv \mathsf{Nothing}$, and that
- $\mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ g$ is injective on s'.

Consequently, setting

$$f \equiv \mathit{fromJust} \circ \mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ g\,, \tag{17}$$

we have

$$\mathit{map}\ f\ s' == s \tag{18}$$

and that f is injective on s'. By Lemma 6, this gives

$$\mathit{map}\ f\ (\mathit{get}\ s') == \mathit{get}\ s$$

and that every i in $get\ s'$ is also in s'. Together with (14) and Lemma 4, we can conclude that h is defined and that for every $i :: \mathsf{Int}$,

$$\begin{aligned}\mathsf{IntMapEq}.\mathit{lookup}\ i\ h == \mathbf{if}\ \mathit{elem}\ i\ (\mathit{get}\ s')\ \mathbf{then}\ \mathsf{Just}\ (f\ i) \\ \mathbf{else}\ \ \mathsf{Nothing}\,.\end{aligned}$$

On the other hand, we have by (16), (17), and the fact (derived above) that for every $i :: \mathsf{Int}$ not in s', $\mathsf{IntMapEq}.\mathit{lookup}\ i\ g \equiv \mathsf{Nothing}$, that for every $i :: \mathsf{Int}$,

$$\begin{aligned}\mathsf{IntMapEq}.\mathit{lookup}\ i\ g \equiv \mathbf{if}\ \mathit{elem}\ i\ s'\ \mathbf{then}\ \mathsf{Just}\ (f\ i) \\ \mathbf{else}\ \ \mathsf{Nothing}\,.\end{aligned}$$

Hence, by (15), the injectivity of $\mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ g$ on s' (derived above), the fact (also derived above) that every i in $get\ s'$ is also in s', and the specification of $\mathsf{IntMapEq}.\mathit{union}$, we have that h' is defined and that for every i in s',

$$\mathsf{IntMapEq}.\mathit{lookup}\ i\ h' == \mathsf{Just}\ (f\ i)\,.$$

Together with (12) and (18), this gives

$$\mathit{bff}_{\mathsf{Eq}}\ get\ s\ (get\ s) == s\,.$$

C. Proof of Theorem 4

Let $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$, let τ be a type that is an instance of Eq, and let $v, s :: [\tau]$. If $\mathit{bff}_{\mathsf{Eq}}\ get\ s\ v$ is defined, then we necessarily have

$$\mathit{bff}_{\mathsf{Eq}}\ get\ s\ v \equiv \mathit{map}\ f\ s'\,, \tag{19}$$

where

$$(s', g) \equiv \mathit{template}_{\mathsf{Eq}}\ s \tag{20}$$

$$\mathsf{Right}\ h \equiv \mathit{assoc}_{\mathsf{Eq}}\ (\mathit{get}\ s')\ v \tag{21}$$

$$\mathsf{Right}\ h' \equiv \mathsf{IntMapEq}.\mathit{union}\ h\ g \tag{22}$$

$$f \equiv \mathit{fromJust} \circ \mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ h'\,. \tag{23}$$

By (20) and Lemma 3 we have that for every i in s', it holds $\mathsf{IntMapEq}.\mathit{lookup}\ i\ g \equiv \mathsf{Just}\ a$ for some $a :: \tau$. Moreover, by (21) and Lemma 5 we have

$$\mathit{map}\ (\mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ h)\ (\mathit{get}\ s') == \mathit{map}\ \mathsf{Just}\ v\,, \tag{24}$$

as well as that

- for every $i :: \mathsf{Int}$ not in $\mathit{get}\ s'$, $\mathsf{IntMapEq}.\mathit{lookup}\ i\ h \equiv \mathsf{Nothing}$, and that
- $\mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ h$ is injective on $\mathit{get}\ s'$.

Putting all these facts together with (22), the specification of $\mathsf{IntMapEq}.\mathit{union}$, and (23), we get that f is injective on s'. Thus, by (19) and Lemma 6,

$$\mathit{get}\ (\mathit{bff}_{\mathsf{Eq}}\ get\ s\ v) == \mathit{map}\ f\ (\mathit{get}\ s')\,. \tag{25}$$

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

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 irresistable. 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[1].

```
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.

```
repeat :: a → [a]
repeat x = x : repeat x
```

```
zapp :: [a → b] → [a] → [b]
zapp (f : fs) (x : xs) = f x : zapp fs xs
zapp _        _        = []
```

The general scheme is as follows:

$$\mathsf{zipWith}_n \;::\; (a_1 \to \cdots \to a_n \to b) \to [a_1] \to \cdots \to [a_n] \to [b]$$
$$\mathsf{zipWith}_n\ f\ xs_1\ \ldots\ xs_n = \mathsf{repeat}\ f\ \text{`}\mathsf{zapp}\text{`}\ xs_1\ \text{`}\mathsf{zapp}\text{`}\ \ldots\ \text{`}\mathsf{zapp}\text{`}\ xs_n$$

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

[1] 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 Env v is some notion of environment and 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)   = 𝕂 i
eval (Add p q) = 𝕂 (+) `𝕊` eval p `𝕊` eval q
```

where

$$\mathbb{K} :: a \to env \to a \qquad \mathbb{S} :: (env \to a \to b) \to (env \to a) \to (env \to b)$$
$$\mathbb{K}\ x\ \gamma = x \qquad \mathbb{S}\ ef\ es\ \gamma = (ef\ \gamma)\ (es\ \gamma)$$

Except for the noise of the $\mathbb{K}$ and $\mathbb{S}$ combinators[2], 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:

identity	pure id ⊛ u	=	u
composition	pure $(\cdot)$ ⊛ u ⊛ v ⊛ w	=	u ⊛ (v ⊛ w)
homomorphism	pure f ⊛ pure x	=	pure (f x)
interchange	u ⊛ pure x	=	pure ($\lambda f \to f\ x$) ⊛ u

[2] 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:

$$(\langle\$\rangle) :: \mathsf{Applicative}\ f \Rightarrow (a \to b) \to f\ a \to f\ b$$
$$f \langle\$\rangle\ u = \mathsf{pure}\ f \circledast u$$

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:

$$\mathsf{pure}\ f \circledast u_1 \circledast \ldots \circledast u_n$$

This canonical form captures the essence of Applicative programming: computations have a fixed structure, given by the pure function, and a sequence of subcomputations, given by the effectful arguments. We therefore find it convenient, at least within this paper, to write this form using a special bracket notation,

$$[\![f\ u_1\ \ldots\ u_n]\!]$$

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 $\mathbb{i}$ and $\mathbb{I}$ with the right behaviour[3].

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 $(\to)\ env$:

```
instance Applicative ((→) env) where
  pure x    = λγ → x                -- 𝕂
  ef ⊛ ex = λγ → (ef γ) (ex γ)      -- 𝕊
```

With these instances, sequence and eval become:

```
sequence :: [IO a] → IO [a]
sequence []     = ⟦[]⟧
sequence (c : cs) = ⟦(:) c (sequence cs)⟧

eval :: Exp v → Env v → 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

[3] *Hint:* Define an overloaded function applicative $u\ v1\ \ldots\ vn\ \mathbb{i} = u \circledast v1 \circledast \ldots \circledast vn$

library's 'list of successes' (Wadler, 1985) monad and take instead an instance Applicative [] that supports 'vectorisation', where pure = repeat and (⊛) = 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 $a \to$ 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 $[\![\cdots]\!]$, 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 $\mathsf{an} :: \mathsf{Id}\ a \to a$ at the same time as the constructor $\mathsf{An} :: a \to \mathsf{Id}\ a$. 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

$$\mathsf{fmap}\ f = \mathsf{an} \cdot \mathsf{traverse}\ (\mathsf{An} \cdot f)$$

Meertens (1998) defined generic dist-like operators, families of functions of type $t\ (f\ a) \to f\ (t\ a)$ 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 $[\![\cdots]\!]$ 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          = ⟦Leaf⟧
  traverse f (Node l x r) = ⟦Node (traverse f l) (f x) (traverse f r)⟧
```

This construction even works for non-regular types. However, not every Functor is Traversable. For example, the functor $(\to)\ env$ cannot in general be Traversable. To see why, take $env = \mathsf{Integer}$ 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
  ∅   :: o
  (⊕) :: o → o → 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 ⇒ Applicative (Accy o) where
  pure _              = Acc ∅
  Acc o1 ⊛ Acc o2 = Acc (o1 ⊕ 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) ⇒ (a → o) → t a → o
accumulate f = acc · traverse (Acc · f)

reduce :: (Traversable t, Monoid o) ⇒ t o → o
reduce = accumulate id
```

Operations like flattening and concatenation become straightforward:

```
flatten :: Tree a → [a]          concat :: [[a]] → [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
  ∅                       = Might False
  Might x ⊕ Might y = Might (x ∨ y)

any :: Traversable t ⇒ (a → Bool) → t a → Bool
any p = might · accumulate (Might · p)
```

Now any · (≡) 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

```
(>>=) :: Accy o a → (a → Accy o b) → 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 $\circledast$ 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 $\circledast$ keeps

the structure of a computation fixed, just sequencing the effects. For example, one may write

$$\begin{aligned}&\mathsf{miffy} :: \mathsf{Monad}\ m \Rightarrow m\ \mathsf{Bool} \to m\ a \to m\ a \to m\ a\\&\mathsf{miffy}\ mb\ mt\ me = \mathbf{do}\\&\quad b \leftarrow mb\\&\quad \mathbf{if}\ b\ \mathbf{then}\ mt\ \mathbf{else}\ me\end{aligned}$$

so that the value of mb will choose between the *computations* mt and me, performing only one, whilst

$$\begin{aligned}&\mathsf{iffy} :: \mathsf{Applicative}\ f \Rightarrow f\ \mathsf{Bool} \to f\ a \to f\ a \to f\ a\\&\mathsf{iffy}\ fb\ ft\ fe = [\![\ cond\ fb\ ft\ fe\ ]\!]\ \mathbf{where}\\&\quad cond\ b\ t\ e = \mathbf{if}\ b\ \mathbf{then}\ t\ \mathbf{else}\ e\end{aligned}$$

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,

$$\mathsf{iffy}\ [\![\ \mathsf{True}\ ]\!]\ [\![\ t\ ]\!]\ \mathsf{Nothing} = \mathsf{Nothing}$$

because the 'else' computation fails, even though its value is not needed, but

$$\mathsf{miffy}\ [\![\ \mathsf{True}\ ]\!]\ [\![\ t\ ]\!]\ \mathsf{Nothing} = [\![\ t\ ]\!]$$

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:

$$\mathsf{miffy}\ [\![\ (\leqslant)\ \mathit{getSpeed}\ \mathit{getSpeedLimit}\ ]\!]\ \mathit{stepOnIt}\ \mathit{checkMirror}$$

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 '⊛' 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*,

$$\mathbf{newtype}\ (f \circ g)\ a = \mathsf{Comp}\{\mathsf{comp} :: (f\ (g\ a))\}$$

just by lifting the inner Applicative operations to the outer layer:

$$\begin{aligned}&\mathbf{instance}\ (\mathsf{Applicative}\ f, \mathsf{Applicative}\ g) \Rightarrow \mathsf{Applicative}\ (f \circ g)\ \mathbf{where}\\&\quad \mathsf{pure}\ x = \mathsf{Comp}\ [\![\ (\mathsf{pure}\ x)\ ]\!]\\&\quad \mathsf{Comp}\ fs \circledast \mathsf{Comp}\ xs = \mathsf{Comp}\ [\![\ (\circledast)\ fs\ xs\ ]\!]\end{aligned}$$

As a consequence, the composition of two monads may not be a monad, but it is certainly applicative. For example, both Maybe ∘ IO and IO ∘ Maybe are applicative: IO ∘ 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 '⪢='. However with the Applicative interface we can continue in the face of errors:

```
instance Monoid err ⇒ Applicative (Except err) where
  pure                          = OK
  OK f ⊛ OK x                   = OK (f x)
  OK f ⊛ Failed err             = Failed err
  Failed err ⊛ OK x             = Failed err
  Failed err1 ⊛ Failed err2     = Failed (err1 ⊕ 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 (⇝) where
  arr   :: (a → b) → (a ⇝ b)
  (⋙)   :: (a ⇝ b) → (b ⇝ c) → (a ⇝ c)
  first :: (a ⇝ b) → ((a, c) ⇝ (b, c))
```

Examples include ordinary '→', 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 '⊛', the '⋙' 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 (⇝) env a = Env (env ⇝ a)

instance Arrow (⇝) ⇒ Applicative (EnvArrow (⇝) env) where
  pure x           = Env (arr (const x))
  Env u ⊛ Env v = Env (u△v ⋙ arr (λ(f, x) → f x))
    where u△v = arr dup ⋙ first u ⋙ arr swap ⋙ first v ⋙ 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 (⇝) a b = Static (f (a ⇝ b))

instance (Applicative f, Arrow (⇝)) ⇒ Arrow (StaticArrow f (⇝)) where
  arr f                 = Static ⟦(arr f)⟧
  Static f ⋙ Static g = Static ⟦(⋙) 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 '⊛', but there is an equivalent symmetrical definition.

```
class Functor f ⇒ Monoidal f where
  unit :: f ()
  (⋆)  :: f a → f b → f (a, b)
```

These operations are clearly definable for any Applicative functor:

```
unit    :: Applicative f ⇒ f ()
unit    = pure ()
(⋆)     :: Applicative f ⇒ f a → f b → f (a, b)
fa ⋆ fb = ⟦(,) fa fb⟧
```

Moreover, we can recover the Applicative interface from Monoidal as follows:

```
pure      :: Monoidal f ⇒ a → f a
pure x    = fmap (λ_ → x)        unit
(⊛)       :: Monoidal f ⇒ f (a → b) → f a → f b
mf ⊛ mx = fmap (λ(f, x) → f x) (mf ⋆ mx)
```

The laws of Applicative given in Section 2 are equivalent to the usual Functor laws, plus the following laws of Monoidal:

naturality of $\star$	$\mathsf{fmap}\ (f \times g)\ (u \star v)$	$=$	$\mathsf{fmap}\ f\ u \star \mathsf{fmap}\ g\ v$
left identity	$\mathsf{fmap}\ \mathsf{snd}\ (\mathsf{unit} \star v)$	$=$	v
right identity	$\mathsf{fmap}\ \mathsf{fst}\ (u \star \mathsf{unit})$	$=$	u
associativity	$\mathsf{fmap}\ \mathsf{assoc}\ (u \star (v \star w))$	$=$	$(u \star v) \star w$

for the functions

$$(\times) :: (a \to b) \to (c \to d) \to (a, c) \to (b, d)$$
$$(f \times g)\ (x, y) = (f\ x, g\ y)$$
$$\mathsf{assoc} :: (a, (b, c)) \to ((a, b), c)$$
$$\mathsf{assoc}\ (a, (b, c)) = ((a, b), c)$$

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:

$$\begin{array}{ccc}
(A \times B) \times (F\,C \times F\,D) & \cong & (A \times F\,C) \times (B \times F\,D) \\
{\scriptstyle (A \times B) \times m}\downarrow & & \downarrow{\scriptstyle t \times t} \\
(A \times B) \times F\,(C \times D) & & F\,(A \times C) \times F\,(B \times D) \\
{\scriptstyle t}\downarrow & & \downarrow{\scriptstyle m} \\
F\,((A \times B) \times (C \times D)) & \cong & F\,((A \times C) \times (B \times D))
\end{array}$$

Note that B and $F\,C$ 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:

$$\mathsf{fmap}\ \mathsf{swap}\ (\mathsf{pure}\ x \star u) = u \star \mathsf{pure}\ x$$

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 $[\![f\ u_1\ \ldots\ u_n]\!]$ 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:

$$\begin{array}{c} {}^{1}\!/\!_{1}\ {}^{2}\!/\!_{1}\ {}^{3}\!/\!_{1}\cdots {}^{m}\!/\!_{1}\cdots \\ {}^{1}\!/\!_{2}\ {}^{2}\!/\!_{2}\ {}^{3}\!/\!_{2}\cdots {}^{m}\!/\!_{2}\cdots \\ \vdots \\ {}^{1}\!/\!_{n}\ {}^{2}\!/\!_{n}\ {}^{3}\!/\!_{n}\cdots {}^{m}\!/\!_{n}\cdots \\ \vdots \end{array}$$

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:

$$\begin{array}{l} rats_1 :: [Rational] \\ rats_1 = concat\ (diags\ [[{}^{m}\!/\!_{n} \mid m \leftarrow [1..]] \mid n \leftarrow [1..]]) \\ diags = diags'\ [\,] \\ \qquad \textbf{where}\ diags'\ xss\ (ys : yss) = map\ head\ xss : diags'\ (ys : map\ tail\ xss)\ yss \end{array}$$

Equivalently, one can deforest the matrix altogether, and generate the diagonals directly:

$$\begin{array}{l} rats_2 :: [Rational] \\ rats_2 = concat\ [[{}^{m}\!/\!_{d-m} \mid m \leftarrow [1..d-1]] \mid d \leftarrow [2..]] \end{array}$$

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 coprime 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:

$$\begin{aligned} &gcd && :: (Integer, Integer) \to Integer \\ &gcd\ (m,n) &&= \textbf{if } m < n \textbf{ then } gcd\ (m, n-m) \textbf{ else} \\ &&&\quad \textbf{if } m > n \textbf{ then } gcd\ (m-n, n)\ \textbf{ else } m \end{aligned}$$

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:

$$\begin{aligned} &igcd && :: (Integer, Integer) \to (Integer, [Bool]) \\ &igcd\ (m,n) &&= \textbf{if } m < n \textbf{ then } step\ False\ (igcd\ (m, n-m)) \textbf{ else} \\ &&&\quad \textbf{if } m > n \textbf{ then } step\ True\ \ (igcd\ (m-n, n))\ \textbf{ else } (m, [\,]) \\ &&&\quad\quad \textbf{where } step\ b\ (d, bs) = (d, b : bs) \end{aligned}$$

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:

$$\begin{aligned} &ungcd :: (Integer, [Bool]) \to (Integer, Integer) \\ &ungcd\ (d, bs) = foldr\ undo\ (d,d)\ bs \\ &\quad\quad \textbf{where } undo\ False\ (m,n) = (m, n+m) \\ &\quad\quad\quad\quad\ \ undo\ True\ \ (m,n)\ \ = (m+n, n) \end{aligned}$$

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 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*:

$$m/n = m'/n' \iff pgcd\ (m,n) = pgcd\ (m', n')$$

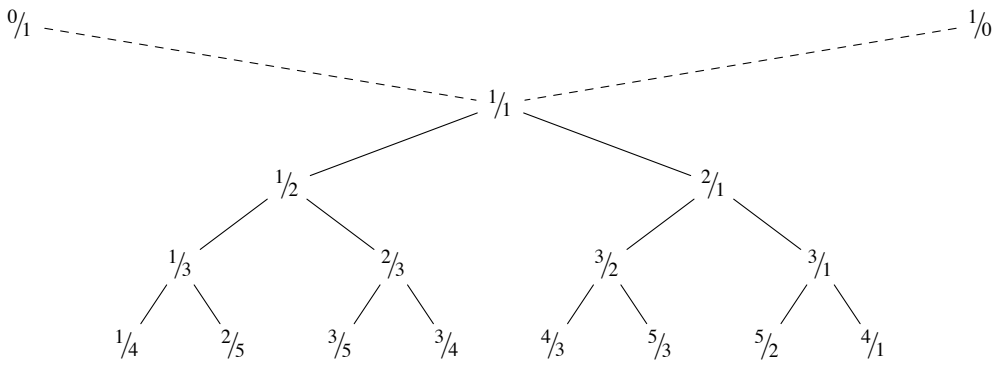


Fig. 1. The first few levels of the Stern-Brocot tree.

Moreover, *pgcd* is surjective: every finite boolean sequence is the *pgcd* of some pair. The function *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 *ungcd*:

$$\begin{array}{ll} \mathit{rats}_3 & :: [\mathit{Rational}] \\ \mathit{rats}_3 & = \mathit{map}\ (\mathit{mkRat} \circ \mathit{curry}\ \mathit{ungcd}\ 1)\ \mathit{boolseqs} \\ \mathit{boolseqs} & = [\,] : [b : bs \mid bs \leftarrow \mathit{boolseqs}, b \leftarrow [\mathit{False}, \mathit{True}]] \\ \mathit{mkRat}\ (m,n) & = {}^{m}\!/_{n} \end{array}$$

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, $\mathit{pgcd}\,(3,4)$ is $[\mathit{False}, \mathit{True}, \mathit{True}]$, 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 $^3/_4$ has ancestors $^2/_3, ^1/_2, ^1/_1, ^0/_1, ^1/_0$, of which $^1/_1$ and $^1/_0$ are to the right and the others to the left. The rightmost left ancestor is $^2/_3$, and the leftmost right ancestor $^1/_1$, and indeed $^3/_4 = ^{2+1}/_{3+1}$. That is why we included the two pseudo-nodes $^0/_1$ and $^1/_0$ in Figure 1: they are needed to make this relationship work for nodes like $^1/_3$ and $^3/_1$ 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.

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Tree}\ a & = \mathit{Node}\ (a, \mathit{Tree}\ a, \mathit{Tree}\ a) \\
\mathit{foldt}\ f\ (\mathit{Node}\ (a,x,y)) & = f\ (a, \mathit{foldt}\ f\ x, \mathit{foldt}\ f\ y) \\
\mathit{unfoldt}\ f\ x & = \textbf{let}\ (a,y,z) = f\ x\ \textbf{in}\ \mathit{Node}\ (a, \mathit{unfoldt}\ f\ y, \mathit{unfoldt}\ f\ z) \\
\mathit{rats}_4 & :: [\mathit{Rational}] \\
\mathit{rats}_4 & = \mathit{bf}\ (\mathit{unfoldt}\ \mathit{step}\ ((0,1),(1,0))) \\
 & \quad \textbf{where}\ \mathit{step}\ (l,r) = \textbf{let}\ m = \mathit{adj}\ l\ r\ \textbf{in} \\
 & \qquad\qquad (\mathit{mkRat}\ m, (l,m), (m,r)) \\
\mathit{adj}\ (m,n)\ (m',n') & = (m+m', n+n') \\
\mathit{bf} & = \mathit{concat} \circ \mathit{foldt}\ \mathit{glue} \\
 & \quad \textbf{where}\ \mathit{glue}\ (a, xs, ys) = [a] : \mathit{zipWith}\ (+\!\!+)\ xs\ ys
\end{array}
$$

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 $^{m+m'}/_{n+n'}$ between each existing adjacent pair $^m/_n, ^{m'}/_{n'}$.

$$
\begin{array}{ll}
\mathit{rats}_5 & :: [\mathit{Rational}] \\
\mathit{rats}_5 & = \mathit{concat}\ (\mathit{unfolds}\ \mathit{infill}\ [(0,1),(1,0)]) \\
\mathit{unfolds}\ f\ a & = \textbf{let}\ (b,a') = f\ a\ \textbf{in}\ b : \mathit{unfolds}\ f\ a' \\
\mathit{infill}\ xs & = (\mathit{map}\ \mathit{mkRat}\ ys, \mathit{interleave}\ xs\ ys) \\
 & \quad \textbf{where}\ ys = \mathit{zipWith}\ \mathit{adj}\ xs\ (\mathit{tail}\ xs) \\
\mathit{interleave}\ (x:xs)\ ys & = x : \mathit{interleave}\ ys\ xs \\
\mathit{interleave}\ [\,]\ \ [\,] & = [\,]
\end{array}
$$

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, $^5/_2 < e < ^3/_1$, so e has a representation starting $[\mathit{True}, \mathit{True}, \mathit{False}, \mathit{True}, \ldots]$.

4 The Calkin-Wilf tree

The Stern-Brocot tree is the trie of the mapping from boolean sequences $\mathit{pgcd}\ (m,n)$ to rationals $^m/_n$. 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 $\mathit{pgcd}\ (m,n)$ to the same rational $^m/_n$. We call this tree the Calkin-Wilf tree, after its two explorers (Calkin & Wilf, 2000), whose work is promoted as one of Aigner and

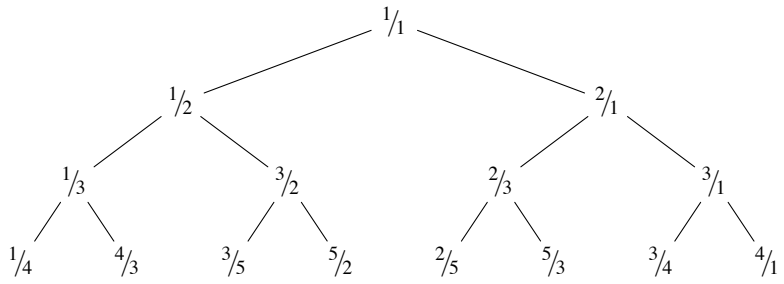


Fig. 2. The first few levels of the Calkin-Wilf tree.

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:

$$
\begin{aligned}
&\mathit{rats}_6 :: [\mathit{Rational}] \\
&\mathit{rats}_6 = \mathit{bf}\ (\mathit{unfoldt}\ \mathit{step}\ (1,1)) \\
&\qquad \textbf{where}\ \mathit{step}\ (m,n) = (^m/_n, (m, m+n), (n+m, n))
\end{aligned}
$$

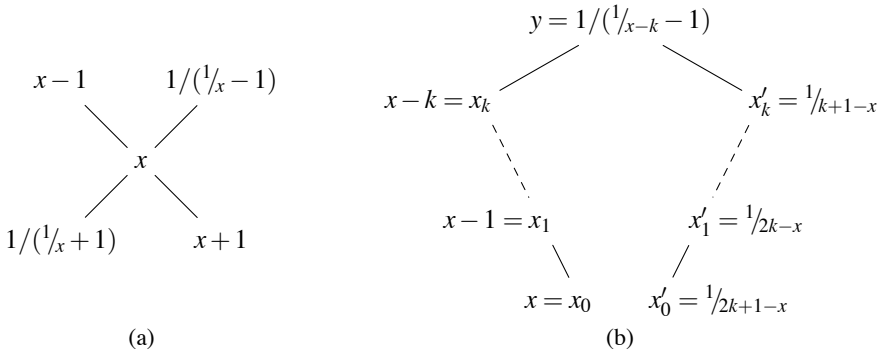


Fig. 3. The neighbours (a) and successor (b) of an element x in the Calkin-Wilf tree.

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.

$$\begin{aligned} &\mathit{iterate} \quad :: (a \to a) \to a \to [a] \\ &\mathit{iterate}\, f\, x = x : \mathit{iterate}\, f\, (f\, x) \end{aligned}$$

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 $y = 1\,/\,(1/x-k - 1)$, 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:

$$\begin{aligned}&\mathit{rats}_7 \;::\; [\mathit{Rational}]\\&\mathit{rats}_7 \;=\; \mathit{iterate}\ \mathit{next}\ 1\\&\mathit{next}\ x = \mathit{recip}\ (\mathit{fromInteger}\ n + 1 - y)\ \textbf{where}\ (n,y) = \mathit{properFraction}\ x\end{aligned}$$

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 $\mathit{next}\ x$ of a rational x is the denominator of x, so in fact the sequence of numerators $1,1,2,1,3,2,3\ldots$ determines the sequence of rationals. This sequence is actually the solution to a natural counting problem: the ith 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 $\mathit{fusc}\ n = \mathit{fusc}\ n'$ 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:

$$\begin{aligned}&\mathit{rats}_8 :: [\mathit{Rational}]\\&\mathit{rats}_8 = \mathit{iterate}\ \mathit{next}'\ 0\\&\qquad\textbf{where}\ \mathit{next}'\ 0 &&= 1\\&\qquad\qquad \mathit{next}'\ x \mid x > 0 &&= \mathit{negate}\ x\\&\qquad\qquad\qquad \mid \mathit{otherwise} &&= \mathit{next}\ (\mathit{negate}\ x)\end{aligned}$$

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 $[a_0, a_1, \ldots, a_n]$. 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 $[a_0, a_1, \ldots, a_n]$ 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

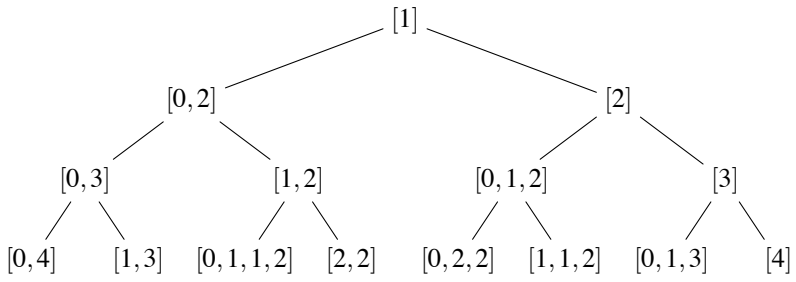


Fig. 4. The first few levels of the Calkin-Wilf tree, as continued fractions.

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:

$$\begin{aligned}
&\mathit{negatecf}\ [n_0] &&= [-n_0]\\
&\mathit{negatecf}\ [n_0,2] &&= [-n_0-1,2]\\
&\mathit{negatecf}\ (n_0:1:n_2:ns) &&= (-n_0-1):(n_2+1):ns\\
&\mathit{negatecf}\ (n_0:n_1:ns) &&= (-n_0-1):1:(n_1-1):ns
\end{aligned}$$

Given this implementation of negation, it is straightforward to derive the following data refinement of rats_7. That is, if c is the continued fraction representation of rational x, then $\mathit{nextcf}\ c$ is the continued fraction representation of $\lfloor x\rfloor + 1 - \{x\}$.

$$\begin{aligned}
&\textbf{type}\ CF = [\mathit{Integer}]\\
&\mathit{rats}_9 :: [CF]\\
&\mathit{rats}_9 = \mathit{iterate}\ (\mathit{recipcf} \circ \mathit{nextcf})\ [1]\\
&\quad \textbf{where}\ \mathit{nextcf}\ [n_0] &&= [n_0+1]\\
&\qquad\quad \mathit{nextcf}\ [n_0,2] &&= [n_0,2]\\
&\qquad\quad \mathit{nextcf}\ (n_0:1:n_2:ns) &&= n_0:(n_2+1):ns\\
&\qquad\quad \mathit{nextcf}\ (n_0:n_1:ns) &&= n_0:1:(n_1-1):ns\\
&\qquad\quad \mathit{recipcf}\ (0:ns) &&= ns\\
&\qquad\quad \mathit{recipcf}\ ns &&= 0:ns
\end{aligned}$$

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:

$$\begin{aligned}
&\mathit{cf2rat} :: CF \to \mathit{Rational}\\
&\mathit{cf2rat} = \mathit{mkRat} \circ \mathit{foldr}\ \mathit{op}\ (1,0)\\
&\qquad\qquad \textbf{where}\ \mathit{op}\ m\ (n,d) = (m \times n + d, n)
\end{aligned}$$

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 $L^{a_n}R^{a_{n-1}}L^{a_{n-2}}\cdots R^{a_0}$, then x is represented by the continued fraction $[a_0,a_1,\ldots,a_n+1]$ (which is not regular if $a_n=0$, but normalizes then to $[a_0,a_1,\ldots,a_{n-1}+1]$). For example, the rational $3/4$ 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:

$$\begin{aligned} &gcd && :: (Integer, Integer) \to Integer \\ &gcd\ (m,n) &&= \textbf{if } m<n \textbf{ then } gcd\ (m, n \bmod m) \textbf{ else} \\ &&&\quad \textbf{if } m>n \textbf{ then } gcd\ (m \bmod n, n)\ \textbf{ else } m \end{aligned}$$

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 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

$$\begin{aligned} sb = zipW\ mkRat\ (t,u) \\ \quad \textbf{where } t &= Node\ (1,t,zipW\ (uncurry\ (+))\ (t,u)) \\ u &= mirror\ t \end{aligned}$$

of the Stern-Brocot tree, where

$$\begin{aligned} zipW\ f = unfoldt\ (apply\ f) \\ \quad \textbf{where } apply\ f\ (Node\ (a,t,u), Node\ (b,v,w)) = (f\ (a,b), (t,v), (u,w)) \end{aligned}$$

and

$$mirror = foldt\ switch\ \textbf{where}\ switch\ (a,t,u) = Node\ (a,u,t)$$

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

$$\begin{aligned} &rats_{10} :: [Rational] \\ &rats_{10} = bf\ (unfoldt\ step\ (1,1)) \\ &\qquad \textbf{where}\ step\ (m,n) = (m/n, (n+m,n), (n,n+m)) \end{aligned}$$

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), *Millenial 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. Credited in *American mathematical monthly*, **110**, 642–643.

Thue, Axel. (1912). Uber 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.

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

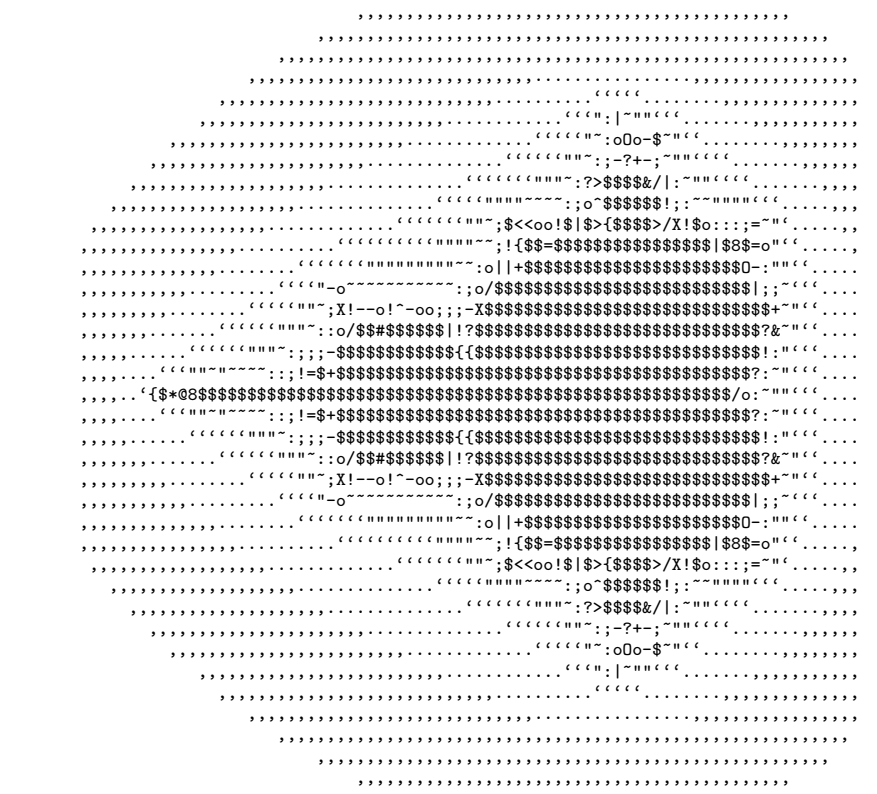


Fig. 1. A picture of the Mandelbrot Set

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.

$$\textbf{type}\ \mathit{Point} \quad = \quad (\mathit{Float}, \mathit{Float})$$

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, *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:[1]

$$\begin{array}{lcl} \mathit{next} & :: & \mathit{Point} \to \mathit{Point} \to \mathit{Point} \\ \mathit{next}\ (u,\ v)\ (x,\ y) & = & (x * x - y * y + u,\ 2 * x * y + v) \end{array}$$

If we pick a point p and another point z, then we can apply $\mathit{next}\ p$ repeatedly to z to generate the following infinite sequence:

$$[z,\ \mathit{next}\ p\ z,\ \mathit{next}\ p\ (\mathit{next}\ p\ z),\ \mathit{next}\ p\ (\mathit{next}\ p\ (\mathit{next}\ p\ z)),\ \ldots$$

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:

$$\begin{array}{lcl} \mathit{mandelbrot} & :: & \mathit{Point} \to [\mathit{Point}] \\ \mathit{mandelbrot}\ p & = & \mathit{iterate}\ (\mathit{next}\ p)\ (0,\ 0) \end{array}$$

For example, if we pick p as the origin, then all of the points are the same:

$$\begin{array}{l} \mathit{mandelbrot}\ (0,\ 0) \\ \quad \Longrightarrow [(0,\ 0),\ (0,\ 0),\ (0,\ 0),\ (0,\ 0),\ (0,\ 0),\ \ldots \end{array}$$

If we start with larger coordinate values, then the numbers can grow rapidly:

$$\begin{array}{l} \mathit{mandelbrot}\ (0.5,\ 0) \\ \quad \Longrightarrow [(0.5,\ 0),\ (0.75,\ 0),\ (1.0625,\ 0),\ (1.62891,\ 0),\ (3.15334,\ 0),\ \ldots \end{array}$$

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:

$$\begin{array}{l} \mathit{mandelbrot}\ (0.1,\ 0) \\ \quad \Longrightarrow [(0.1,\ 0),\ (0.11,\ 0),\ (0.1121,\ 0),\ (0.112566,\ 0),\ (0.112671,\ 0),\ \ldots \end{array}$$

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 *mandelbrot* (0.1, 0) from the 100th onwards (and possibly some before) are in fact equal. (Switching from *Float* to a double precision

[1] 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 $\mathit{next}\ p\ z = z^2 + p$, 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:

$$\begin{array}{lcl} \mathit{fairlyClose} & :: & \mathit{Point} \to \mathit{Bool} \\ \mathit{fairlyClose}\ (u,\ v) & = & (u * u + v * v) < 100 \end{array}$$

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*:

$$\begin{array}{lcl} \mathit{inMandelbrotSet} & :: & \mathit{Point} \to \mathit{Bool} \\ \mathit{inMandelbrotSet}\ p & = & \mathit{all}\ \mathit{fairlyClose}\ (\mathit{mandelbrot}\ p) \end{array}$$

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:

$$\textbf{type}\ \mathit{Image}\ \mathit{color} \quad = \quad \mathit{Point} \to \mathit{color}$$

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:

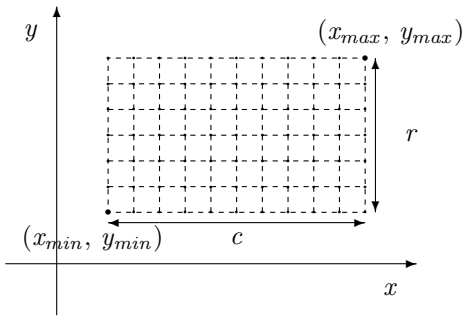
$$\begin{array}{lcl} \mathit{fracImage} & :: & (\mathit{Point} \to [\mathit{Point}]) \to [\mathit{color}] \to \mathit{Image}\ \mathit{color} \\ \mathit{fracImage}\ \mathit{fractal}\ \mathit{palette} & = & \mathit{chooseColor}\ \mathit{palette}\ .\ \mathit{fractal} \end{array}$$

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:

$$\textbf{type}\ \mathit{Grid}\ a \quad = \quad [[a]]$$

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 (x_min, y_min) (x_max, y_max)
    = [[(x, y) | x ← for c x_min x_max] | y ← for r y_min y_max]
```

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+={^O#%&@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[2]. We will not explore these possibilities

[2] 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

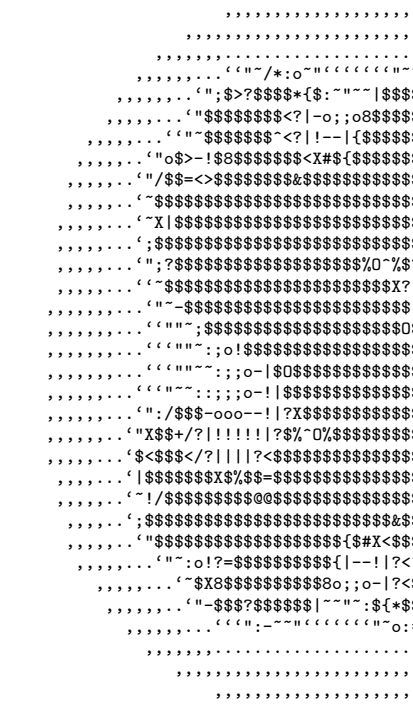


Fig. 2. The Julia set for (0.32,0.043)

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

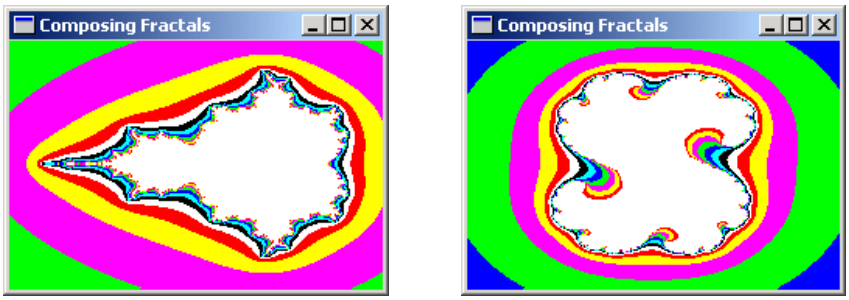


Fig. 3. Graphical Displays of Mandelbrot and Julia Sets

images use the same parameters as in the original figures but with a finer resolution grid of 240 by 160 pixels.

$$\begin{aligned} \mathit{figure3left} &= \mathit{draw}\ \mathit{points}\ \mathit{mandelbrot}\ \mathit{rgbPalette}\ \mathit{rgbRender} \\ &\quad \textbf{where}\ \mathit{points} = \mathit{grid}\ 240\ 160\ (-2.25, -1.5)\ (0.75, 1.5) \end{aligned}$$

$$\begin{aligned} \mathit{figure3right} &= \mathit{draw}\ \mathit{points}\ (\mathit{julia}\ (0.32, 0.043))\ \mathit{rgbPalette}\ \mathit{rgbRender} \\ &\quad \textbf{where}\ \mathit{points} = \mathit{grid}\ 240\ 160\ (-1.5, -1.5)\ (1.5, 1.5) \end{aligned}$$

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.

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 α-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

Haskell'04, September 22, 2004, Snowbird, Utah, USA.

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

$$\mathtt{data\ Nat} = \mathtt{Zero} \mid \mathtt{Suc\ Nat}$$

We should like to synthesize some statement corresponding to

$$\begin{array}{l}\forall \mathtt{P} \in \mathtt{Nat} \to \mathtt{Prop}.\\ \mathtt{P\,Zero} \to\\ (\forall \mathtt{k} \in \mathtt{Nat}.\,\mathtt{P\,k} \to \mathtt{P}\,(\mathtt{Suc\,k})) \to\\ \forall \mathtt{n} \in \mathtt{Nat}.\,\mathtt{P\,n}\end{array}$$

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

$$\mathtt{data\ Nat} = \mathtt{Zero} \mid \mathtt{P\ Nat} \quad \text{or even} \quad \mathtt{data\ P} = \mathtt{Zero} \mid \mathtt{Suc\ P}$$

We'll have to be careful not to end up with such nonsense as

$$\begin{array}{l}\forall \mathtt{P} \in \mathtt{Nat} \to \mathtt{Prop}.\\ \mathtt{P\,Zero} \to\\ (\forall \mathtt{k} \in \mathtt{Nat}.\,\mathtt{P\,k} \to \mathtt{P}\,(\mathtt{P\,k})) \to\\ \forall \mathtt{n} \in \mathtt{Nat}.\,\mathtt{P\,n}\end{array} \quad \text{or} \quad \begin{array}{l}\forall \mathtt{P} \in \mathtt{P} \to \mathtt{Prop}.\\ \mathtt{P\,Zero} \to\\ (\forall \mathtt{k} \in \mathtt{P}.\,\mathtt{P\,k} \to \mathtt{P}\,(\mathtt{Suc\,k})) \to\\ \forall \mathtt{n} \in \mathtt{P}.\,\mathtt{P\,n}\end{array}$$

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**,

$$\begin{array}{l}\forall - \in \mathtt{Nat} \rightarrow \mathtt{Prop}.\\ 0\,\mathtt{Zero} \rightarrow\\ (\forall - \in \mathtt{Nat}.\,2\,0 \rightarrow 3\,(\mathtt{Suc}\,1)) \rightarrow\\ \forall - \in \mathtt{Nat}.\,3\,0\end{array}$$

or from the outside inward—de Bruijn **levels**.

$$\begin{array}{l}\forall 0 \in \mathtt{Nat} \rightarrow \mathtt{Prop}.\\ 0\,\mathtt{Zero} \rightarrow\\ (\forall 2 \in \mathtt{Nat}.\,0\,2 \rightarrow 0\,(\mathtt{Suc}\,2)) \rightarrow\\ \forall 3 \in \mathtt{Nat}.\,0\,3\end{array}$$

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 α-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 λ-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:[1]

```
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 Names can meaningfully be compared with the ordinary (==) test—α-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 B 0 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 B 0. 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)
```

[1] 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 B 0) 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 α-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 α-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.

$$\text{type Name} = \text{Stack}(\text{String}, \text{Int})$$

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':

$$xs \succcurlyeq (xs \lhd\!+ ys)$$

$$\begin{array}{l}\text{infixl}\,4 \lhd\!+ \\ (\lhd\!+) :: \text{Stack}\,x \to \text{Stack}\,x \to \text{Stack}\,x \\ xs \lhd\!+ \text{Empty} \quad = xs \\ xs \lhd\!+ (ys :\!< y) = xs \lhd\!+ ys :\!< y\end{array}$$

We say that two names are **independent**, $xs \perp ys$, if neither $xs \succcurlyeq ys$ nor $ys \succcurlyeq xs$. 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.

$$xs \perp ys \to (xs \lhd\!+ xs') \perp (ys \lhd\!+ ys')$$

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.

$$root \begin{cases} root\text{'s } \textbf{variables}: \\ root :\!< (\text{“x”},0),\ \ldots,\ root :\!< (\text{“x”},m), \\ root :\!< (\text{“y”},0),\ \ldots,\ root :\!< (\text{“y”},n), \\ \ldots \\ root\text{'s } \textbf{agents}: \\ root :\!< (\text{“a”},0) \begin{cases} (root :\!< (\text{“a”},0))\text{'s } \textbf{variables}: \\ root :\!< (\text{“a”},0) :\!< (\text{“x”},0),\ \ldots \\ (root :\!< (\text{“a”},0))\text{'s } \textbf{agents}: \\ root :\!< (\text{“a”},0) :\!< (\text{“a”},0),\ \ldots \end{cases} \\ \vdots \\ root :\!< (\text{“a”},k) \begin{cases} (root :\!< (\text{“a”},k))\text{'s } \textbf{variables}: \\ root :\!< (\text{“a”},k) :\!< (\text{“x”},0),\ \ldots \\ (root :\!< (\text{“a”},k))\text{'s } \textbf{agents}: \\ root :\!< (\text{“a”},k) :\!< (\text{“a”},0),\ \ldots \end{cases} \end{cases}$$

Note the convenience of $(\text{String}, \text{Int})$ as the type of name elements. The Strings give us legibility; the Ints an easy way to express uniform sequences of distinct name-extensions $x_0, \ldots x_n$. Two little helpers will make simple names easier to construct:

$$\begin{array}{l}\text{infixl}\,6 /\!/ \\ (/\!/) :: \text{Name} \to \text{String} \to \text{Name} \\ root /\!/ s = root :\!< (s, 0)\end{array}$$

$$\begin{array}{l}\textbf{nm} :: \text{String} \to \text{Name} \\ \textbf{nm}\, s = \text{Empty} /\!/ s\end{array}$$

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*.

$$\text{type Agency}\ agentT = \text{Name} \to agentT$$

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

$$\begin{array}{l}\text{infix} \longleftarrow \\ (\longleftarrow) :: \text{MonadPlus}\, m \Rightarrow \\ \qquad \text{Agency}\ (\text{Expr} \to m\,(\text{Binding}, \text{Expr}))\end{array}$$

That is, $(root \longleftarrow)$ 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 $\longrightarrow$.

$$\begin{array}{l}\text{type Prefix} = \text{Stack Binding} \\ \\ \text{infixr}\,6 \twoheadrightarrow \\ (\twoheadrightarrow) :: \text{Prefix} \to \text{Expr} \to \text{Expr} \\ \text{Empty} \qquad\quad \twoheadrightarrow expr \quad = expr \\ (binds :\!< bind) \twoheadrightarrow range = binds \twoheadrightarrow bind \longrightarrow range\end{array}$$

The corresponding destructor is an *agency*. Given a *root* and a string x, it delivers a quantifier prefix with names of the form $root :\!< (x, \_i)$ where the 'subscript' $\_i$ is numbered from 1:

$$\begin{array}{l}\textbf{unprefix} :: \text{Agency}\,(\text{String} \to \text{Expr} \to (\text{Prefix}, \text{Expr})) \\ \textbf{unprefix}\ root\ x\ expr = \textbf{intro}\,1\,(\text{Empty}, expr) \quad \text{where} \\ \quad \textbf{intro} :: \text{Int} \to (\text{Prefix}, \text{Expr}) \to (\text{Prefix}, \text{Expr}) \\ \quad \textbf{intro}\ \_i\ (binds, expr) = \text{case}\ (root :\!< (x, \_i)) \longleftarrow expr\ \text{of} \\ \qquad \text{Just}\,(bind, range) \to \textbf{intro}\ (\_i + 1)\ (binds :\!< bind, range) \\ \qquad \text{Nothing} \qquad\quad \to (binds, expr)\end{array}$$

Note that **intro** specifically exploits the Maybe instance of the monadically lifted binding agency ($\longleftarrow$).

If *root* is independent of all the names in *expr*—which it will be, if we maintain our hierarchical discipline—and

$$\textbf{unprefix}\ root\ x\ expr\ =\ (binds, range)$$

then *range* is unquantified and $expr = binds \twoheadrightarrow range$.

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 $(x_1, \ldots, x_n)$. 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 $$.

```
infixl 9 $$
($$) :: 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 $\forall x_1 : X_1. \ldots \forall x_m : X_m. R\, e_1 \ldots e_n$, 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 λ-abstraction and β-redices, 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
  (F f, 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:

```
infixl 9 –$$
(–$$) :: Name → Prefix → Expr
f –$$ parameters = apply (F f) 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 λ-calculus, these tools make it easy to implement λ-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:

$$\text{data}\ \left(\frac{X : \star;\ n : \mathsf{Nat}}{\mathsf{Vec}\, X\, n : \star}\right)$$

$$\text{where}\ \left(\frac{}{\mathsf{Vnil} : \mathsf{Vec}\, X\, \mathsf{Zero}}\right);\ \left(\frac{x : X;\ xs : \mathsf{Vec}\, X\, n}{\mathsf{Vcons}\, x\, xs : \mathsf{Vec}\, X\, (\mathsf{Suc}\, n)}\right)$$

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:

$$\begin{aligned}
\mathsf{Vec} &: \forall X \in \mathsf{Set}.\, \forall n \in \mathsf{Nat}.\, \mathsf{Set} \\
\mathsf{Vnil} &: \forall X \in \mathsf{Set}.\, \mathsf{Vec}\, X\, \mathsf{Zero} \\
\mathsf{Vcons} &: \forall X \in \mathsf{Set}.\, \forall n \in \mathsf{Nat}.\, \forall x \in X.\, \forall xs \in \mathsf{Vec}\, X\, n.\, \mathsf{Vec}\, X\, (\mathsf{Suc}\, n)
\end{aligned}$$

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*,[2] 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:

$$
\begin{array}{ll}
\mathsf{Vec}\text{-}\mathbf{Ind} \in & \\
\quad \left.\begin{array}{l} \forall X \in \mathsf{Set}. \\ \forall n \in \mathsf{Nat}. \\ \forall xs \in \mathsf{Vec}\,X\,n. \end{array}\right\} & \textbf{targets} \\
\quad \forall P \in \forall X \in \mathsf{Set}.\, \forall n \in \mathsf{Nat}.\, \forall xs \in \mathsf{Vec}\,X\,n.\, \mathsf{Set}. & \textbf{motive} \\
\quad \left.\begin{array}{l} \forall m_n \in \forall X \in \mathsf{Set}.\, P\,X\,\mathsf{Zero}\,(\mathsf{Vnil}\,X). \\ \forall m_c \in \forall X \in \mathsf{Set}.\, \forall n \in \mathsf{Nat}.\, \forall x \in X. \\ \qquad \forall xs \in \mathsf{Vec}\,X\,n.\, \forall h \in P\,X\,n\,xs. \\ \qquad P\,X\,(\mathsf{Suc}\,X)\,(\mathsf{Vcons}\,X\,n\,x\,xs). \end{array}\right\} & \textbf{methods} \\
\quad P\,X\,n\,xs &
\end{array}
$$

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

   $$\mathsf{F} : \forall i_1 : I_1. \ldots \forall i_n : I_n. \mathsf{Set}$$

   Correspondingly, the target prefix is $\forall \vec{\imath} : \vec{I}.\, \forall x : \mathsf{F}\,\vec{\imath}$, and the motive has type $P : \forall \vec{\imath} : \vec{I}.\, \forall x : \mathsf{F}\,\vec{\imath}.\, \mathsf{Set}$.

2. Each constructor has type

   $$\mathsf{c} : \forall a_1 : A_1. \ldots \forall a_m. : A_m. \mathsf{F}\,s_1\,\ldots\,s_n$$

   where the $\vec{s}$ do not mention F. The corresponding method has type

   $$\forall \vec{a} : \vec{A}.\, \forall \vec{h} : \vec{H}.\, P\,\vec{s}\,(\mathsf{c}\,\vec{a})$$

   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

   $$a : \forall y_1 : Y_1. \ldots \forall y_k : Y_k. \mathsf{F}\,\vec{r}$$

   where F is not mentioned[3] in the $\vec{Y}$ or the $\vec{r}$. The corresponding inductive hypothesis is

   $$h : \forall \vec{y} : \vec{Y}.\, P\,\vec{r}\,(a\,\vec{y})$$

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

```
data InfTree : ⋆ where Leaf : InfTree
                       Node : (Nat → InfTree) → InfTree
```

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:

```
makeIndElim :: Agency (Binding → Prefix → Binding)
makeIndElim root (family :∈ famtype) constructors =
  root :∈ targets ↠
          motive ⟶
          fmap method constructors ↠
          bName motive -$$ targets
            where  — constructions from condition 1
              ForAll indices set [] =
                analysis root "i" famtype
              targets = indices :<
                        root // "x" :∈ family -$$ indices
              motive = root // "P" :∈ targets ↠
                                      F (nm "Set")
              method :: Binding → Binding
              ...
```

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.

```
method :: Binding → Binding
method (con :∈ contype) =
  meth :∈ conargs ↠
          (indhyp =<< conargs) ↠
          bVar motive $$ conindices :$ (con -$$ conargs)
            where
              meth = root // "m" <+ con
              ForAll conargs fam conindices =
                analysis meth "a" contype
              indhyp :: Binding → Prefix
              ...
```

The method's type says that the motive should hold for those targets

[2] 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.

[3] This condition is known as *strict positivity*.

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 α-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 bookkeeping 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 λ-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

$$\forall \vec{s} : \vec{S}. \forall x : \mathsf{F}\vec{t}.\, T$$

into an equivalent problem which is immediately susceptible to elimination with operators like those constructed in our example.

$$\begin{array}{l}\forall \vec{t} : \vec{I}. \forall x' : \mathsf{F}\vec{t}. \\ \forall \vec{s} : \vec{S}. \forall x : \mathsf{F}\vec{t}.\, T. \\ \vec{t} = \vec{t} \rightarrow x' = x \rightarrow \\ T\end{array}$$

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 Mathematicæ*, 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*. Almquist and Wiksell, Stockholm, 1965.

[25] P. Wadler. How to Replace Failure by a list of Successes. In Jouannaud [13], pages 113–128.

Type-Safe Cast

Stephanie Weirich*
Department of Computer Science
Cornell University
Ithaca, NY 14853
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 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 α, that is it executes the same code regardless of the identity of α, 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 α and `find[`α`]` instantiates this type for the recursive call. Unfortunately, this function is of type

$\forall\alpha$`. (string` $\times$ α`) list` $\rightarrow$ `string` $\rightarrow$ α.

If we were looking up an `int`, the list passed to `find` could only contain pairs of `string`s and `int`s. We would not be

*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.

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` τ'

coerces a value of type τ with τ' substituted for β into one of type $\exists\beta.\tau$. Conversely, the instruction

`unpack (`β`,x) = e in e'`

destructs an existential package `e` of type $\exists\beta.\tau$, binding a new type variable β and a new term variable `x` of type τ within the expression `e'`.

Therefore, we can create a symbol table of type `(string` $\times$ $\exists\beta.\beta$`) list` with the expression

```
[ ("x", pack 1 as ∃β.β hiding int) ;
  ("y", pack (2,3) as ∃β.β 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 =
   Λα. fn table => fn (symbol, obj) =>
      (symbol, pack obj as ∃β.β hiding α) :: 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 α. If we use the symbol table correctly, then α and β will abstract the same type, but we need to verify this property before we can convert something of type β to type α. We can not compare α and β and still remain parametric in α and β. 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 λ_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, α may also be examined. In λ_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 λ_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 λ_i^{ML}, a simple function, `sametype`, to compare two types and return true if they match, can be implemented with nested `typecase`s. 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.[1] For product and function types, `sametype` calls itself recursively on the subcomponents of the type. Each of these branches binds type variables (such as α_1 and α_2) to the subcomponents of the types so that they may be used in the recursive call.

```
let rec sametype  =
   Λα. Λβ.
      typecase (α) of
        int =>
          typecase (β) of
            int => true
          | _   => false
      | (α1 × α2)=>
          typecase (β) of
            (β1 × β2) =>
            sametype[α1][β1]
               andalso sametype[α2][β2]
          | _ => false
      | (α1 → α2) =>
          typecase (β) of
            (β1 → β2) =>
            sametype[α1][β1]
               andalso sametype[α2][β2]
          | _ => false
```

As these nested `typecase`s are tedious to write, we borrow from the pattern matching syntax of Standard ML, and abbreviate this function as:

[1] For brevity, we only include `int`, product types, and function types in the examples.

```
let rec sametype  =
   Λα. Λβ.
      typecase (α,β) of
        (int,int) => true
      | (α1 × α2,β1 × β2) =>
        sametype[α1][β1]
           andalso sametype[α2][β2]
      | (α1 → α2,β1 → β2) =>
        sametype[α1][β1]
           andalso sametype[α2][β2]
      | (_,_) => 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 =
   Λα. fn table => fn symbol =>
      case table of
        [] => raise NotFound
      | ( (name, package) :: rest ) =>
        unpack (β,value) = package in
        if symbol = name
              andalso sametype[α][β]
        then value
        else find[α] rest symbol
```

we discover that this use does not type check. The return type for `find` is the existentially bound β which escapes its scope. Even though we have added a dynamic check that α is equivalent to β, the check does not change the type of `value` from β to α.

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 λ_i^{ML} the type of each branch of a `typecase` can depend on the analyzed type.

```
typecase τ of
   int => fn (x:int) => x + 3
 | α → β =>
   fn(x:α → β) => x
 | α × β =>
   fn(x:α × β) => x
```

For example, although the first branch above is of type `int` → `int`, the second of type $(\beta \to \gamma) \to (\beta \to \gamma)$, and the third of type $(\beta \times \gamma) \to (\beta \times \gamma)$, all three branches are instances of the type schema $\gamma \to \gamma$, when γ is replaced with the identity of τ for that branch. Therefore, this entire `typecase` expression can be safely assigned the type $\tau \to \tau$.

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

```
typecase [γ.γ → γ] τ 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 $\forall\alpha.\forall\beta.\alpha \to \beta$. This function will just return its argument (at the new type) if the type arguments match, and raise an exception otherwise.[2]

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.

```
cast : ∀α. ∀β. α → β
let rec cast =
   Λα. Λβ.
     typecase [δ1,  δ2. δ1 → δ2](α,β) of
      (int, int) =>
        fn (x:int) => x
    | (α1 × α2,  β1 × β2) =>
        let val f = cast [α1][β1]
            val g = cast [α2][β2]
        in
           fn (x:α1 × α2) =>
             (f (fst x), g (snd x))
        end
    | (α1 → α2,  β1 → β2) =>
        let val f = cast [β1][α1]
            val g = cast [α2][β2]
        in
          fn (x:α1 → α2) =>
            g ∘ x ∘ f
        end
    | (_,_) => raise CantCast
```

Figure 1: First Solution

2. FIRST SOLUTION

An initial implementation of `cast` using the facilities of λ_i^{ML} appears in Figure 1. In the first branch of the `typecase`, α

[2]It would also be reasonable to produce a function of type $\alpha \to (\beta$ `option`$)$, but checking the return values of recursive calls for `NONE` only lengthens the example.

and β 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 α and β 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 (α_1 to β_1 and α_2 to β_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 α and β are both function types, due to contravariance. Here, given `x`, a function from α_1 to α_2, we want to return a function from β_1 to β_2. We can apply cast to α_2 and β_2 to get a function, `g`, that casts the result type, and compose `g` with the argument `x` to get a function from α_1 to β_2. Then we can compose that resulting function with a reverse cast from β_1 to α_1 to get a function from β_1 to β_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 τ_1 and τ_2 are both `int`, the result of `cast[`τ_1`][`τ_2`]` 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 $\alpha_1 \to \beta_1$) 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 $(\alpha_1 \times \alpha_2) \to (\beta_1 \times \alpha_2)$ and one from $(\beta_1 \times \alpha_2) \to (\beta_1 \times \beta_2)$.

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 α and β for analysis, but an additional *type constructor*[3] argument γ. When γ is applied to the type α we get the type of `x`, when it is applied to β we get the return type of the cast. For example, if γ is $\lambda\delta{:}*.\delta \times \alpha_2$, we get a function from type $\alpha \times \alpha_2$ to $\beta \times \alpha_2$.

[3]We create type constructors with λ-notation, and annotate the bound variable with its *kind*. Kinds classify types and type constructors: $*$ is the kind of types, and if κ_1 and κ_2 are kinds, $\kappa_1 \to \kappa_2$ is the the kind of type constructors from κ_1 to κ_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 α, β, and γ with their kinds. As α and β are types, they are annotated with kind $*$, but γ is a function from types to types, and so has kind $* \to *$. 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 α_1 to β_1) by supplying the type constructor $\lambda\delta{:}*.\gamma(\delta \times \alpha_2)$ for γ. 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 β_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 $\gamma(\alpha_1 \to \alpha_2)$ to type $\gamma(\beta_1 \to \alpha_2)$. 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 α is in the class `Show` if there is some function named `show` defined for type $\alpha \rightarrow$ `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 α and β 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 `typecase`s. 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 β_1 to α_1, we need α_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 λ_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 `(CF α1, CF α2) => (α1, α2) → (β1, β2)`. 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 α_1 and α_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 β_1 and β_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 λ_i^{ML} solution. As the type of this function should be `(CT α1, CF α2) => (α1 → α2) → (β1 → β2)`, 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 α_2 instance of `CF`, that requires that β_2 be in the class `CT`. However, we would also like to `cast` the argument of the function in the opposite direction, from β_1 to α_1. Therefore we need β_1 to be in the class `CF`, and α_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 γ as well as α and β. This addition leads to the new definitions of `CT` and `CF`. Note that in the type of `doFn`, α_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 examineded. In Haskell we cannot explicitly instantiate the type constructor argument as we did in λ_i^{ML}, but we can give Haskell enough hints to infer the correct one.[4] 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"
```

[4]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` α_1`, CF` α_2`) =>` γ $(\alpha_1,\alpha_2) \rightarrow \gamma$ (β_1,β_2).

Therefore `z` is of type γ (α_1, α_2) so `LP z` is of type `LProd` α_2 γ α_1. 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` β`) =>` $\gamma'\alpha \rightarrow \gamma'\beta$, so to typecheck the call `cast' (LP z)`, we only need to find an α in the class `CT` and a γ' 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 γ' and α is a simple match – γ' is the partial application `LProd` α_2 γ and α is α_1. Thus, the result of `cast'` is of type `(LProd` α_2 γ`)` β, for some β in `CT`, and `y` is of type γ (β, α_2).

Now `RP y` is of type `RProd` β γ α_2, so we need the α_2 instance of `cast'` for the second call. This instance is of type $\forall\gamma''$. `CT` β' `=>` γ'' $\alpha_2 \rightarrow \gamma''$ β'. This γ'' 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 γ (β, β'). Comparing this type to the return type of `doProd`, we unify β with β_1 and β' with β_2. This unification satisfies our constraints for the two calls to `cast'`, as we assumed that both β_1 and β_2 are in the class `CT`.

Just as in the second λ_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 λ_i^{ML}, $\exists\alpha$. `CF` α `=>` α 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 λ_R [2], a version of λ_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.[5]

```
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

[5]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 τ provides an injection from τ into the exception type. Coercing a value from the dynamic type to τ is matching the exception constructor with `E`.

Alternatively, if the language supports subtyping and downcasting, 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 α 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 fonctionelle 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
```

J. Functional Programming **1** (1): 1–000, January 1993

FUNCTIONAL PEARLS

A Poor Man's Concurrency Monad

Koen Claessen
Chalmers University of Technology
email: koen@cs.chalmers.se

Abstract

Without adding any primitives to the language, we define a concurrency monad transformer in Haskell. This allows us to add a limited form of concurrency to any existing monad. The atomic actions of the new monad are lifted actions of the underlying monad. Some extra operations, such as `fork`, to initiate new processes, are provided. We discuss the implementation, and use some examples to illustrate the usefulness of this construction.

1 Introduction

The concept of a *monad* (Wadler, 1995) is nowadays heavily used in modern functional programming languages. Monads are used to model some form of computation, such as non-determinism or a stateful calculation. Not only does this solve many of the traditional problems in functional programming, such as I/O and mutable state, but it also offers a general framework that abstracts over many kinds of computation.

It is known how to use monads to model concurrency. To do this, one usually constructs an imperative monad, with operations that resemble the Unix `fork` (Jones & Hudak, 1993). For reasons of efficiency and control, *Concurrent Haskell* (Peyton Jones *et al.*, 1996) even provides primitive operations, which are defined outside the language.

This paper presents a way to model concurrency, generalising over arbitrary monads. The idea is to have *atomic* actions in some monad that can be *lifted* into a concurrent setting. We explore this idea within the language; we will not add any primitives.

2 Monads

To express the properties of monads in Haskell, we will use the following type class definition. The bind operator of the monad is denoted by $(\star)$, and the unit operator by `return`.

```
class Monad m where
  (⋆)    :: m α → (α → m β) → m β
  return :: α → m α
```

Furthermore, throughout this paper we will use the so-called *do*-notation as syntactic sugar for monadic expressions. The following example illustrates a traditional monadic expression on the left, and the same, written in do-notation, on the right.

```
expr1 ⋆ λx.                do x ← expr1
expr2 ⋆ λ_.                 ; expr2
expr3 ⋆ λy.                 ; y ← expr3
return expr4                ; return expr4
```

As an example, we present a monad with output, called the *writer monad*. This monad has an extra operator called `write`. It takes a string as argument, which becomes output in a side effect of the monad. The bind operator (⋆) of the monad has to take care of combining the output of two computations.

A monad having this operator is an instance of the following class.

```
class Monad m ⇒ Writer m where
  write :: String → m ()
```

A typical implementation of such a monad is a pair containing the result of the computation, together with the output produced during that computation.

```
type W α = (α, String)

instance Monad W where
  (a, s) ⋆ k = let (b, s') = k a in (b, s++s')
  return x = (x, "")

instance Writer W where
  write s = ((), s)
```

Note how the bind operator concatenates the output of the two subactions.

Most monads come equipped with a *run* function. This function executes a computation, taking the values inside one level downwards. The monad `W` has such a run function, we call it `output`, which returns the output of a computation in `W`.

```
output        :: W α → String
output (a, s) = s
```

2.1 Monad Transformers

Sometimes, a monad is parametrised over another monad. This is mostly done to add more functionality to an existing monad. In this case we speak of a *monad transformer* (Liang *et al.*, 1995). An example is the exception monad transformer; it adds a way to escape a monadic computation with an error message. In general, operations that work on one specific monad can be *lifted* into the new, extended monad.

Again, we can express this by using a type class.

```
class MonadTrans τ where
  lift :: Monad m ⇒ m α → (τ m) α
```

A type constructor τ forms a monad transformer if there is an operation `lift` that transforms any action in a monad m into an action in a monad $\tau\ m$.

In this paper we will discuss a monad transformer called `C`. It has the interesting property that any monadic action that is lifted into the new monad will be considered an *atomic* action in a concurrent setting. Also some extra operations are provided for this monad, for example `fork`, which deals with process initiation.

3 Concurrency

How are we going to model concurrency? Since we are not allowed to add primitives to the language, we are going to simulate concurrent processes by *interleaving* them. Interleaving implements concurrency by running the first part of one process, suspending it, and then allowing another process to run.

3.1 Continuations

To suspend a process, we need to grab its future and stick it away for later use. *Continuations* are an excellent way of doing this. We can change a function into *continuation passing style* by adding an extra parameter, the continuation. Instead of producing the result directly, the function will now apply the continuation to the result. We can view the continuation as the *future* of the computation, as it specifies what to do with the result of the function.

Given a computation type `Action`, a function that uses a continuation with result type α has the following type.

```
type C α = (α → Action) → Action
```

The type `Action` contains the actual computation. Since, in our case, we want to parametrise this over an arbitrary monad, we want `Action` (and also `C`) to be dependent on a monad m.

```
type C m α = (α → Action m) → Action m
```

`C` is the concurrency monad transformer we use in this paper. That means that `C` m is a monad, for every monad m.

```
instance Monad m ⇒ Monad (C m) where
  f ⋆ k      = λc. f (λa. k a c)
  return x   = λc. c x
```

Sequencing of continuations is done by creating a new continuation for the left computation that contains the right computation. The unit operator just passes its argument to the continuation.

3.2 Actions

The type `Action` m specifies the actual actions we can do in the new monad. What does this type look like? For reasons of simplicity, flexibility, and expressiveness (Scholz, 1995), we implement it as a datatype that describes the different actions we provide in the monad.

First of all, we need atoms, which are computations in the monad m. We are inside a continuation, so we want these atomic computations to return a new action. Also, we need a constructor for creating new processes. Lastly, we provide a constructor that does not have a continuation; we will use it to end a process. We also call this the empty process.

```
data Action m
  = Atom (m (Action m))
  | Fork (Action m) (Action m)
  | Stop
```

To express the connection between an expression of type `C` $m\ \alpha$ and an expression of type `Action` m, we define a function `action` that transforms one into the other. It finishes the computation by giving it the `Stop` continuation.

```
action     :: Monad m ⇒ C m α → Action m
action m   = m (λa. Stop)
```

To make the constructors of the datatype `Action` easily accessible, we can define functions that correspond to them. They will create an action in the monad `C` m.

The first function is the function `atom`, which turns an arbitrary computation in the monad m into an atomic action in `C` m. It runs the atomic computation and monadically returns a new action, using the continuation.†

```
atom     :: Monad m ⇒ m α → C m α
atom m   = λc. Atom (do a ← m ; return (c a))
```

† This is actually the monadic map, but because `Functor` is not a superclass of `Monad` in Haskell we cannot use `map`.

In addition, we have a function that uses the `Stop` constructor, called `stop`. It discards any continuation, thus ending a computation.

```
stop :: Monad m ⇒ C m α
stop = λc. Stop
```

To access `Fork`, we define two operations. The first, called `par`, combines two computations into one by forking them both, and passing the continuation to both parts. The second, `fork`, resembles the more traditional imperative fork. It forks its argument after turning it into an action, and continues by passing () to the continuation.

```
par          :: Monad m ⇒ C m α → C m α → C m α
par m1 m2    = λc. Fork (m1 c) (m2 c)

fork         :: Monad m ⇒ C m α → C m ()
fork m       = λc. Fork (action m) (c ())
```

The type constructor `C` is indeed a monad transformer. Its lifting function is the function `atom`; every lifted action becomes an atomic action in the concurrent setting.

```
instance MonadTrans C where
  lift = atom
```

We have now defined ways to construct actions of type `C` $m\ \alpha$, but we still can not *do* anything with them. How do we model concurrently running actions? How do we interpret them?

3.3 Semantics

At any moment, the status of the computation is going to be modelled by a list of (concurrently running) actions. We will use a scheduling technique called *round-robin* to interleave the processes. The concept is easy: if there is an empty list of processes, we are done. Otherwise, we take a process, run its first part, take the continuation, and put that at the back of the list. We keep doing this recursively until the list is empty.

We implement this idea in the function `round`.

```
round :: Monad m ⇒ [Action m] → m ()
round []       = return ()
round (a : as) = case a of
  Atom am   → do a' ← am ; round (as ++ [a'])
  Fork a1 a2 → round (as ++ [a1, a2])
  Stop      → round as
```

An `Atom` monadically executes its argument, and puts the resulting process at the back of the process list. `Fork` creates two new processes, and `Stop` discards its process.

As for any monad, we need a run function for `C` m as well. It just transforms its argument into an action, creates a singleton process list, and applies the round-robin function to it.

```
run     :: Monad m ⇒ C m α → m ()
run m   = round [action m]
```

As we can see, the type α disappears in the result type. This means that we lose the result of the original computation. This seems very odd, but often (and in the cases of the examples in this paper) we are only interested in the *side effects* of a computation. It is possible to generalise the type of `run`, but that goes beyond the scope of this paper.

4 Examples

We will present two examples of monads that can be lifted into the new concurrent world.

4.1 Concurrent Output

Recall the writer monad example from Sect. 2. We can try lifting this monad into the concurrent world. To do this, we want to say that every instance of a writer monad can be lifted into a concurrent writer monad.[‡]

```
instance Writer m ⇒ Writer (C m) where
  write s = lift (write s)
```

The function `lift` here is the `atom` of the monad transformer `C`. Every `write` action, after lifting, becomes an atomic action. This means that no computation will produce output while another `write` is writing.

Before we present an example, we first define an auxilary function `loop`. This function works in any writer monad. It takes one argument, a string, and writes it repeatedly to the output.

```
loop    :: Writer m ⇒ String → m ()
loop s  = do write s ; loop s
```

We use this function to define a computation in $C\ m\ \alpha$ that creates two processes that are constantly writing. One process writes the string "`fish`", the other writes "`cat`".

[‡] Actually, we want to say this for all monad transformers at once, but Haskell does not currently allow us to express this.

```
example :: Writer m ⇒ C m ()
example = do write "start!"
             ; fork (loop "fish")
             ; loop "cat"
```

The result of the expression `output (run example)` looks like the following string.

"`start!fishcatfishcatfishcatfishcatfishca ...`"

Because we defined `write` as an atomic action, the writing of one "`fish`" and one "`cat`" cannot interfere. If we want finer grained behaviour, we can split one write action into several write actions, e.g. the separate characters of a string. A simple way of doing this is to change the lifting of `write`.

```
instance Writer m ⇒ Writer (C m) where
  write []     = return ()
  write (c : s) = do lift (write [c]) ; write s
```

The lifting is now done character-by-character. The result of the expression `output (run example)` now looks like this.

"`start!fciasthcfaitschafticsahtfciasthcfaitscha ...`"

4.2 Merging of Infinite Lists

A well known problem, called the *merging of infinite lists*, is as follows. Suppose we have an infinite list of infinite lists, and want to collapse this list into one big infinite list. The property we want to hold is that every element in any of the original lists is reachable within a finite number of steps in the new list. This technique is for example used to prove that the set **Q** of rationals has a countable number of elements.

Using the writer monad with the new lifting, we can solve this problem for an infinite list of infinite strings. The idea is that, for each string, we create a process that writes the string. If we fork this infinite number of processes, and run the resulting computation, the output will be the desired infinite string.

We will take a step back in order to present a piece of useful theory. There are monads that have a so-called *monoidal* structure on them. That means that there is an operator, denoted by $(+\!\!+)$, that combines two computations of the same type into one, and that there is an identity element for this operation, called `zero`. In Haskell, we can say:

```
class Monad m ⇒ Monoidal m where
  (++)  :: m α → m α → m α
  zero  :: m α
```

The function `concat`, with type `Monoidal` $m \Rightarrow [m\ \alpha] \to \alpha$, uses $(+\!\!+)$ and `zero` to concatenate a (possibly infinite) list of such computations together.

The reason we are looking at this is that `C` m admits a monoidal structure; the parallel composition `par` represents the $(+\!\!+)$, and the process `stop` represents its identity element `zero`.

```
instance Monad m ⇒ Monoidal (C m) where
  (++)  = par
  zero  = stop
```

This means we can use `concat` to transform an infinite list of processes into a process that concurrently runs these computations. To merge an infinite list of infinite strings, we transform every string into a writing process, fork them with `concat`, and extract the output.

```
merge :: [String] → String
merge  = output ∘ run ∘ concat ∘ map write
```

Of course, this function also works for finite lists, and can be adapted to act on more general lists than strings.

4.3 Concurrent State

In Haskell, the so-called `IO` monad provides *mutable state*. Within the monad we can create, access, and update pieces of storage. The type of a storage that contains an object of type α is `Var` α. The functions we use to control these `Var`s, the non-proper morphisms of `IO`, have the following types.

```
newVar   :: IO (Var α)
readVar  :: Var α → IO α
writeVar :: Var α → α → IO ()
```

In the lifted version of this monad, the `C IO` monad, we can have several concurrent processes sharing pieces of state. In a concurrent world however, we often want more structure on shared state. Concurrent Haskell (Peyton Jones *et al.*, 1996), an extension of Haskell with primitives for creating concurrent processes, recognised this. It introduces a new form of shared state: the `MVar`.

Like a `Var`, an `MVar` can contain a value, but it may also be empty. An `MVar` becomes empty after a process has done a read operation on it. Processes reading an empty `MVar` will block, until a new value is put into the `MVar`. `MVar`s are a powerful mechanism for creating higher level concurrent data abstractions. They can for example be used for synchronization and data sharing at the same time.

It is possible to integrate `MVar`s with our concurrency monad transformer, using the mutable state primitives we already have. First, we have to think of how to represent an `MVar`. An `MVar` can be in two different states; it can either be full (containing some value), or empty.

```
type MVar α  = Var (Maybe α)
data Maybe α = Just α | Nothing
```

We use the datatype `Maybe` to indicate that there is `Just` a value in an `MVar`, or `Nothing` at all.

Let us now define the operations that work on `MVar`s. The function that creates an `MVar` lifts the creation of a `Var`, and puts `Nothing` in it.

```
newMVar :: C IO (MVar α)
newMVar = lift ( do v ← newVar
                  ; writeVar v Nothing
                  ; return v )
```

We can use the same trick when writing to an `MVar`.[§]

```
writeMVar     :: MVar α → α → C IO ()
writeMVar v a = lift ( writeVar v (Just a) )
```

The hardest function to define is `readMVar`, since it has to deal with blocking. To avoid interference when reading an `MVar`, we perform an atomic action that pulls the value out of the `Var` and puts `Nothing` back. We introduce an auxilary function `takeVar`, working on the unlifted `IO` monad, that does this.

```
takeVar  :: MVar α → IO (Maybe α)
takeVar v = do a_m ← readVar v
             ; writeVar v Nothing
             ; return a_m
```

Once we have this function, the definition of a blocking `readMVar` is not hard anymore. We represent blocking by repeatedly trying to read the variable. We realise that this busy-wait implementation is very inefficient, and we indeed have used other methods as well (such as the one used in (Jones, M. *et al.*, 1997)), but we present the easiest implementation here.

```
readMVar  :: MVar α → C IO α
readMVar v = do a_m ← lift (takeVar v)
              ; case a_m of
                  Nothing → readMVar v
                  Just a  → return a
```

Note that `readMVar` itself is not an atomic action, so other processes can also read the `MVar` just after `takeVar`. Fortunately, at that point, the `MVar` is already blocked by the function `takeVar`. It is impossible for `readMVar` to be atomic, since other processes deserve a chance when it is blocking on an `MVar`.

[§] We are a bit sloppy here; the real semantics of `MVar`s is slightly different (Peyton Jones *et al.*, 1996).

For some examples of the use of `MVar`s, we refer the reader to the paper about Concurrent Haskell (Peyton Jones *et al.*, 1996), where `MVar`s are introduced.

5 Discussion

The work presented in this paper is an excellent example of the flexiblity of monads and monad transformers. The power of dealing with different types of computations in this way is very general, and should definitely be more widely used and supported by programming languages. We really had to push the Haskell type class mechanism to its limits in order to make this work. A slightly extended class mechanism would have been helpful (Peyton Jones *et al.*, 1997).

To show that this idea is more than just a toy, we have used this same setting to add concurrency to the graphical system *TkGofer* (Vullinghs *et al.*, 1996). The system increased in expressive power, and its implementation in simplicity. It turns out to be a very useful extension to TkGofer.

We have also experimented with lifting other well-known monads into this concurrent setting. Lifted lists, for example, can be used to express the infinite merging problem more concisely. However, a problem with the type system forced us to fool it in order to make this work. Exception and environment monads (Wadler, 1995) do have the expected behaviour, though we are not able to lift all of the non-proper morphisms of these monads. This is because some of them need a computation as an *argument*, so that lifting becomes non-trivial.

However, there are a few drawbacks. We have not implemented *real* concurrency. We simply allow interleaving of *atomic* actions, whose atomicity plays a vital role in the system. If one atomic action itself does not terminate, the concurrent computation of which it is a part of does not terminate either. We cannot change this, because we cannot step outside the language to interrupt the evaluation of an expression.

The source code of the functions and classes mentioned in this paper is publically available at http://www.cs.chalmers.se/~koen/Code/pearl.hs. It also contains another, more efficient but slightly bigger implementation of `MVar`s.

Acknowledgements

I would like to thank Richard Bird, Byron Cook, Andrew Moran, Thomas Nordin, Andrei Sabelfeld, Mark Shields, Ton Vullinghs, and Arjan van Yzendoorn for their useful comments on earlier drafts of this paper. Most of the work for this paper was done while visiting the Oregon Graduate Institute, and an earlier version was used as part of my Master's thesis at the University of Utrecht, under supervision of Erik Meijer.

References

Jones, M., & Hudak, P. (1993). Implicit and Explicit Parallel Programming in Haskell. Yale University. Tech. Rep. YALEU/DCS/RR-982.

Jones, M. *et al.* (1997). The Hugs System. Nottingham University and Yale University. Url: http://www.haskell.org.

Liang, Sh., Hudak, P., & Jones, M. (1995). Monad Transformers and Modular Interpreters. *Conference Record of 22nd POPL '95*. ACM.

Peyton Jones, S., Gordon, A., & Finne, S. (1996). Concurrent Haskell. *Proceedings of the 23rd POPL '96*. ACM.

Peyton Jones, S., Jones, M., & Meijer, E. (1997). Type Classes: An Exploration of the Design Space. *Proceedings of the Haskell Workshop of the ICPF '97*. ACM.

Scholz, E. (1995). A Concurrency Monad Based on Constructor Primitives. Universität Berlin.

Vullinghs, T., Schulte, W., & Schwinn, T. 1996 (June). *An Introduction to Tk-Gofer*. Tech. rept. 96-03. University of Ulm. Url: http://www.informatik.uni-ulm.de/pm/ftp/tkgofer.html.

Wadler, Ph. (1995). Monads for Functional Programming. *Advanced Functional Programming*. Lecture Notes in Computer Science. Springer Verlag.

FUNCTIONAL PEARLS

Explaining Binomial Heaps

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(e-mail: ralf@uran.informatik.uni-bonn.de*)*

1 Introduction

Functional programming languages are an excellent tool for teaching algorithms and data structures. This paper explains binomial heaps, a beautiful data structure for priority queues, using the functional programming language Haskell (Peterson & Hammond, 1997). We largely follow a deductive approach: using the metaphor of a tennis tournament we show that binomial heaps arise naturally through a number of logical steps. Haskell supports the deductive style of presentation very well: new types are introduced at ease, algorithms can be expressed clearly and succinctly, and Haskell's type classes allow to capture common algorithmic patterns. The paper aims at the level of an undergraduate student who has experience in reading and writing Haskell programs, and who is familiar with the concept of a priority queue.

2 Priority queues

The abstract data type 'priority queue' provides at least the following five operations: $\emptyset$ represents the empty queue; $\wr a \int$ denotes the queue, which contains a as the single element; *insert* a q inserts a into queue q; $q_1 \uplus q_2$ denotes the union of queues q_1 and q_2 (sometimes termed 'meld'); and *splitMin* q extracts a minimal element from q. The notation has been chosen to emphasize the fact that priority queues are conceptually bags, ie unordered collections possibly with duplicates. Priority queues are the data type of choice when an efficient access to the smallest element of a varying collection of elements is required. Applications include discrete event simulation and job scheduling. Here is the class definition for priority queues.

```
data MinView q a  =  Min a (q a) | Infty

class PriorityQueue q where
  ∅               ::  (Ord a) ⇒ q a
  ≀·∫             ::  (Ord a) ⇒ a → q a
  insert          ::  (Ord a) ⇒ a → q a → q a
  (⊎)             ::  (Ord a) ⇒ q a → q a → q a
  splitMin        ::  (Ord a) ⇒ q a → MinView q a
  insert a q      =   ≀a∫ ⊎ q
```

Table 1. *Ladies' singles of the 1996 All England Championship*

Quarterfinal	Semifinal	Final	Champion
K Date			
vs.	K Date		
M Pierce		**S Graf**	
S Graf	vs.		
vs.	**S Graf**		
J Novotna		vs.	S Graf
J Wiesner			
vs.	**A Sanchez Vicario**		
A Sanchez Vicario	vs.	A Sanchez Vicario	
M Fernandez			
vs.	M McGrath		
M McGrath			

While the meaning of the first four operations should be clear, *splitMin* is in need of explanation. The call *splitMin* q has two possible outcomes: if q is empty *splitMin* q returns *Infty*, otherwise it yields *Min* a q' where a is a minimal element of q and q' consists of all elements in q except a.

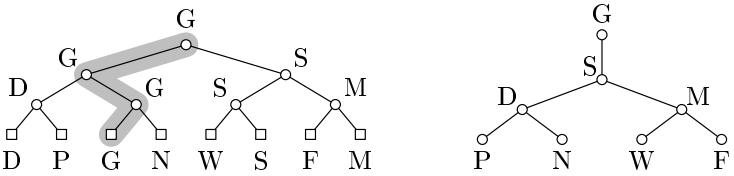
Note that *PriorityQueue* is a constructor class (Jones, 1995); the class variable q ranges over type constructors rather than types (q has kind $* \to *$). A similar comment applies to the definition of *MinView*: the parameter q is a type constructor while a is a type.

To derive an efficient implementation of priority queues we proceed in two steps. First, we consider a simple instance of the problem, namely, to determine the smallest of a *fixed* bag of elements. In a second step we drop the restriction that all elements are given in advance by making the algorithms *incremental*.

3 Tournament trees

Let us assume that we look for the best of, say, n tennis players. The first idea which probably crosses one's mind is to organize a knock-off tournament.[†] For definiteness, consider the results of the Ladies' singles of the 1996 All England Championship listed in Table 1. We see that the course of matches forms a full binary tree; each external node corresponds to a participant, and each internal node corresponds to the winner of a match. The tree representation is shown in Figure 1(a). Regarded as a data structure for priority queues tournament trees appear to be deficient because of the many repeated entries they contain. The champion, for instance, appears on every level of the tree. We may repair this defect if we label each internal node with

[†] It is important to be concrete here. If we posed the abstract problem of determining the smallest of a bag of elements we would probably provoke solutions like *foldl min* ∞.



(a) tournament tree (b) topped loser tree

Fig. 1. Different representations of the tournament listed in Table 1

the loser of the match, instead of the winner, and drop the external nodes altogether. We thus turn the tree of winners into a *tree of losers*. Since every participant—with the exception of the champion—loses exactly one match the labelling of nodes is now unique. If we place the champion additionally at the top of the tree we obtain the structure displayed in Figure 1(b).

To represent topped loser trees in Haskell we reuse the data type *MinView* building upon the standard definition of labelled binary trees.

```
type ToppedTree a  =  MinView BinTree a
data BinTree a     =  Bin a (BinTree a) (BinTree a) | Empty
```

Let us consider next how to determine the second-best player. Assuming a transitive ranking only those players who lost to the champion must be taken into account. In the example above there are three candidates for the second prize: N, D, and S. Consequently, two additional matches are required; the tree-structure of the tournament suggests to let S compete against the winner of D vs. N.

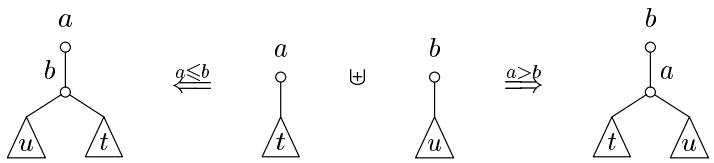
Unfortunately, we cannot set up the matches on the basis of a topped loser tree. It simply does not provide enough information to determine the champion's opponents. This can be seen if we annotate its edges with the established ranking, cf Figure 2(a). The problem is that some players dominate the left, others the

(a) original loser tree (b) left-ordered loser tree

Fig. 2. Determining the second-best player

right subtree but we cannot tell which one. A cure, however, is ready at hand: We arrange the tree such that a loser always dominates the left subtree. Then the champion's opponents are located on the right spine. The modified tree is displayed in Figure 2(b).

The left-ordering property must be taken into account whenever a match is played, ie two topped trees are melded.



Note that the subtrees, t and u, are swapped if $a \leqslant b$. Having fixed the data structure and its properties we are now ready to give the first implementation of priority queues.‡

```
instance PriorityQueue ToppedTree where
  ∅                                  =  Infty
  ⦇a⦈                                =  Min a Empty
  Infty ⊎ u                          =  u
  t ⊎ Infty                          =  t
  Min a t ⊎ Min b u
    | a ⩽ b                          =  Min a (Bin b u t)
    | otherwise                      =  Min b (Bin a t u)
  splitMin Infty                     =  Infty
  splitMin (Min a t)                 =  Min a (secondBest t)
    where secondBest Empty           =  Infty
          secondBest (Bin a' l r)    =  Min a' l ⊎ secondBest r
```

This instance has the amazing property that $(\uplus)$ can be performed in constant time. On the negative side *splitMin* exhibits $\Theta(n)$ worst-case behaviour.§ Consider, for example, the call *splitMin* (*foldr insert* $\emptyset$ $[n, n-1..1]$). The crux is that $(\uplus)$ is repeatedly applied to trees of different sizes resulting in a degenerated tree. Interestingly, the performance depends very much on the order of elements: the call *splitMin* (*foldr insert* $\emptyset$ $[1..n]$), for example, takes only constant time.

4 Bottom-up tournament trees

We have seen that repeated insertions into a topped tree may result in a degenerated list-like structure degrading the performance of subsequent *splitMin* operations. This never happens in a tennis tournament because the participants are known in advance. Dropping this assumption we are faced with the following problem:

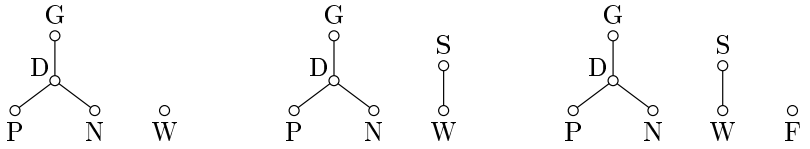
Again, we look for the best of n tennis players. But, due to prior commitments the players do not arrive at the same time; rather they join the tournament one after the other. We probably do not even know how many players are going to

‡ The instance declaration is not legal Haskell since *ToppedTree* is not a datatype defined by **data** or by **newtype**. A datatype, 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.

§ Since Haskell is a lazy language the bounds are amortized rather than worst-case bounds (Okasaki, 1996b).

participate. We only require the series of matches to be fair: opponents should always have played the same number of matches, ie only trees of equal size should be linked. Can we make arrangements to cope with this—from an organizer's point of view quite unpleasant—situation?

The answer is yes and the solution is quite simple, too. Returning to the Ladies' singles assume that the participants arrive in the following order: D, P, G, N, W, S, F, and M. Now, whenever a new player shows up we perform as many matches as possible. Here are three snapshots just after W, S, and F joined the tournament.

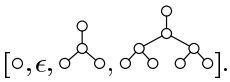


When the last player, M, arrives three pending matches can be carried out resulting in the tree of Figure 1(b). We see that the original tree is constructed in a left-to-right, bottom-up fashion. A match is performed if and only if there are two trees of equal size. This condition guarantees that all loser trees are perfectly balanced. The resulting structures, topped full binary trees, are called *pennants* by Sack and Strothotte (1990). Since a full binary tree of height h contains $2^{h+1} - 1$ nodes, a pennant of height h contains 2^h nodes. This implies that a tournament of size n contains a pennant of height i if the i-th bit in the binary representation of n is 1.

In Haskell we represent a tournament by a list of topped trees. For reasons, that will become clear later, we call the data structure 'binary binomial heap'.

```
type BinBinomialHeap a = [ToppedTree a]
```

A tournament of size $13 = 1 + 0 + 4 + 8$, for example, is represented by the list



The list contains an empty pennant (abbreviated by ϵ) which corresponds to the 0 in the binary representation of 13. Okasaki (1996a) discusses alternative representations.

5 Digression: Binary addition

Since the representation of a tournament is uniquely determined by the binary decomposition of the number of participants it probably comes as no surprise that the operations on heaps resemble arithmetic functions on binary numbers: inserting an element is analogous to incrementing a number, melding two tournaments is analogous to adding two numbers. For that reason we will briefly review the method of summing binary numbers. To abstract away from clerical details of representation we first introduce a class for binary digits.

```
class (Eq b) ⇒ BinaryDigit b where
  zero          :: b
  carry, sum    :: b → b → b
```

The function *sum* calculates the sum bit of two bits, *carry* accordingly determines the carry bit. Recall from the course on computer architecture that a single step of the binary addition is performed by a full adder which calculates the sum of three bits, the two input bits and the carry bit.

$$\begin{array}{lcl} \mathit{fullAdder} & :: & (\mathit{BinaryDigit}\ b) \Rightarrow b \to b \to b \to (b, b) \\ \mathit{fullAdder}\ c\ a\ b & = & (s_2, \mathit{sum}\ c_1\ c_2) \\ \quad \textbf{where}\ (s_1, c_1) & = & \mathit{halfAdder}\ a\ b \\ \qquad\quad (s_2, c_2) & = & \mathit{halfAdder}\ c\ s_1 \end{array}$$

A half adder calculates the sum of two bits.

$$\begin{array}{lcl} \mathit{halfAdder} & :: & (\mathit{BinaryDigit}\ b) \Rightarrow b \to b \to (b, b) \\ \mathit{halfAdder}\ a\ b & = & (\mathit{sum}\ a\ b, \mathit{carry}\ a\ b) \end{array}$$

A binary number is represented by a list of binary digits, the least significant digit coming *first*. Keeping the digits in increasing order of weight is a natural choice since addition proceeds from the least to the most significant digit. To make the representation unique we furthermore disallow trailing zeros. The Haskell function for binary addition, *addWithCarry*, essentially corresponds to a ripple-carry addition (Cormen *et al.*, 1991, p. 661–662). Contrary to an electronic circuit we must arrange for the likely case that the numerals have unequal length. The helper function *addDigit* takes care of these cases.

$$\begin{array}{lcl} \mathit{add} & :: & (\mathit{BinaryDigit}\ b) \Rightarrow [b] \to [b] \to [b] \\ \mathit{add}\ x\ y & = & \mathit{addWithCarry}\ \mathit{zero}\ x\ y \end{array}$$

$$\begin{array}{lcl} \mathit{addWithCarry} & :: & (\mathit{BinaryDigit}\ b) \Rightarrow b \to [b] \to [b] \to [b] \\ \mathit{addWithCarry}\ c\ [\,]\ y & = & \mathit{addDigit}\ c\ y \\ \mathit{addWithCarry}\ c\ x\ [\,] & = & \mathit{addDigit}\ c\ x \\ \mathit{addWithCarry}\ c\ (a : x)\ (b : y) & = & s : \mathit{addWithCarry}\ c'\ x\ y \\ \quad \textbf{where}\ (s, c') & = & \mathit{fullAdder}\ c\ a\ b \end{array}$$

$$\begin{array}{lcl} \mathit{addDigit} & :: & (\mathit{BinaryDigit}\ b) \Rightarrow b \to [b] \to [b] \\ \mathit{addDigit}\ c\ x \mid c \equiv \mathit{zero} & = & x \\ \mathit{addDigit}\ c\ [\,] & = & [c] \\ \mathit{addDigit}\ c\ (a : x) & = & s : \mathit{addDigit}\ c'\ x \\ \quad \textbf{where}\ (s, c') & = & \mathit{halfAdder}\ c\ a \end{array}$$

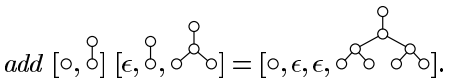
If we represent binary digits by integers,

$$\begin{array}{l} \textbf{instance}\ \mathit{BinaryDigit}\ \mathit{Int}\ \textbf{where} \\ \quad \begin{array}{lcl} \mathit{zero} & = & 0 \\ \mathit{carry}\ m\ n & = & (m + n)\ \text{`}\mathit{div}\text{`}\ 2 \\ \mathit{sum}\ m\ n & = & (m + n)\ \text{`}\mathit{mod}\text{`}\ 2 \end{array} \end{array}$$

we can try adding 3 and 6: $add\ [1,1]\ [0,1,1] = [1,0,0,1]$. Due to the overloading we can reuse *add* to meld two lists of pennants. All we have to do is to supply the following instance declaration.

```
instance (Ord a) ⇒ BinaryDigit (ToppedTree a) where
  zero                          =  Infty
  carry (Min a t) (Min b u)
    | a ⩽ b                     =  Min a (Bin b u t)
    | otherwise                 =  Min b (Bin a t u)
  carry _ _                     =  Infty
  sum Infty u                   =  u
  sum t Infty                   =  t
  sum (Min _ _) (Min _ _)       =  Infty
```

Note that we omit the necessary, but straightforward *Eq* instance declaration for topped trees. The function *carry* corresponds to melding two topped trees. The only difference is that *carry* returns *Infty* if one of the arguments is empty. The function *sum* determines the 'remainder' of a meld. It is instructive to see *add* in action:



The running time of *add* is determined by the length of the input lists. Since a tournament of size n comprises $\lceil \log_2(n+1) \rceil$ pennants the worst-case running time of $(\uplus)$ $(= add)$ is $\Theta(\log n)$. For *insert* one can derive tighter bounds: incrementing a binary number only takes $\Theta(1)$ amortized time (Okasaki, 1996b).

6 Binary binomial heaps

It remains to implement *splitMin* which puts our implementation on the testbench. Since we increased the cost of $(\uplus)$ from $\Theta(1)$ to $\Theta(\log n)$, we expect the running time of *splitMin* to improve. The operation proceeds in three steps. First, a pennant with a minimal root is determined and replaced by *zero*. This is most easily accomplished if we are able to compare topped trees as a whole. The following instance declaration defines a suitable ordering.

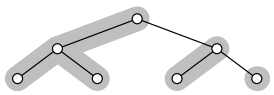
```
instance (Ord a) ⇒ Ord (ToppedTree a) where
  t ⩽ Infty                     =  True
  Infty ⩽ u                     =  False
  Min a _ ⩽ Min b _             =  a ⩽ b
```

Note that the loser trees are not taken into account. This guarantees that comparing two trees has the same complexity as comparing two elements.

```
extractMin                 ::  (Ord t, BinaryDigit t) ⇒ [t] → MinView [] t
extractMin []              =   Infty
extractMin (a : x)         =   case extractMin x of
  Infty                    →   Min a []
  Min b y | a ⩽ b          →   Min a (zero : x)
          | otherwise      →   Min b (a : y)
```

The type of *extractMin* is more general than actually needed: *extractMin* is not restricted to topped trees but works on binary digits. This extra generality pays off in the subsequent section when we change the representation of tournaments.

Having determined the required pennant we are left with the problem of merging the remaining list of pennants with a single full binary tree. Fortunately, we can convert a full binary tree into a tournament using the natural correspondence between binary trees and lists of topped trees. The figure on the right emphasizes the pennants hidden in a full binary tree.



This correspondence is easily coded as a Haskell program which constitutes the second step.

```
dismantle                  ::  BinTree a → [ToppedTree a]
dismantle Empty            =   []
dismantle (Bin a l r)      =   Min a l : dismantle r
```

In the third and last step we simply meld the two lists. Note that the pennants resulting from the binary tree must be reversed beforehand. Since all three steps require $\Theta(\log n)$ time in the worst case, *splitMin* requires $\Theta(\log n)$ time, as well.

```
instance PriorityQueue BinBinomialHeap where
  ∅                        =   []
  ⟅a⟆                      =   [Min a Empty]
  (⊎)                      =   add
  splitMin q               =   case extractMin q of
    Infty                  →   Infty
    Min (Min a t) ts       →   Min a (reverse (dismantle t) ⊎ ts)
```

This instance has the amazing feature that the running time of $(\uplus)$ is completely independent of the elements contained in the queues—quite contrary to our first implementation. The pro is also a con: binomial heaps do not adapt to the input data. You better not use binomial heaps for sorting.

7 Multiway binomial heaps

A data structure for priority queues records our partial knowledge about the ordering of its constituents. Different data structures give rise to different types of

orderings: ordered lists, for example, correspond to chains, unordered lists to antichains. It is instructive to draw the underlying ordering of a pennant. The diagrams for pennants of height 2, 3, and 4 are shown on the right. A moment's reflection reveals that these *multiway trees* are obtained through the natural correspondence between binary trees and forests, described by Knuth (1968, p. 333–334). Figure 3 illustrates the special case of transforming a topped binary tree into a multiway tree.

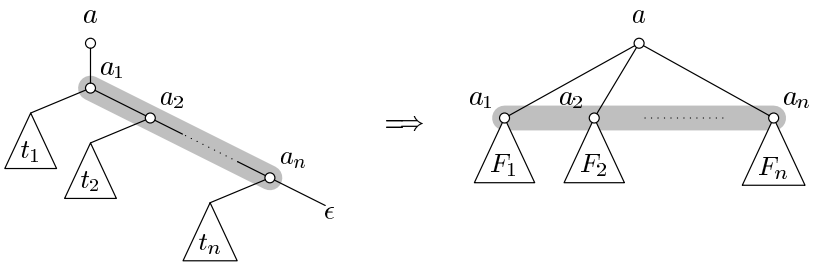


Fig. 3. Natural correspondence between topped binary trees and multiway trees (F_i is the forest corresponding to the binary tree t_i)

This correspondence suggests to implement priority queues directly by multiway trees. The data type for general trees is defined as follows.

```
data Tree a   =  Root a (Forest a) | Void
type Forest a =  [Tree a]
```

Dictated by the application we allow that a tree is empty. We restrict, however, the use of *Void* to the top-level, ie *Void* must not appear beneath a *Root* node. The transformation pictured in Figure 3 can be rigorously defined by a Haskell function.

```
tree                 ::  ToppedTree a → Tree a
tree Infty           =   Void
tree (Min a t)       =   Root a (forest t)

forest               ::  BinTree a → Forest a
forest Empty         =   []
forest (Bin a l r)   =   Root a (forest l) : forest r
```

If we apply *tree* to a pennant we obtain a *binomial tree*. This term is motivated by the fact that a binomial tree of height h contains $\binom{h}{d}$ nodes at depth d. Both, trees and coefficients, are based on a similar inductive scheme. A binomial tree of height $h+1$ consists of two trees of height h that are linked together: one is made the leftmost child of the other (see the definition of *carry* below). Binomial coefficients satisfy the recurrence $\binom{h+1}{d} = \binom{h}{d} + \binom{h}{d-1}$.

With regard to the ordering we have that a binomial tree satisfies the *heap property*: Every node is smaller or equal than any of its descendants.

Since *ToppedTree* and *Tree* are merely different representations of the same underlying structure it is not difficult to adapt the code of the preceding sections to multiway trees. We start by declaring binomial heaps.

```
type BinomialHeap a = Forest a
```

To be able to reuse *add* for melding we make *Tree a* an instance of *BinaryDigit*.

```
instance (Ord a) ⇒ BinaryDigit (Tree a) where
  zero                          = Void
  carry t@(Root a ts) u@(Root b us)
    | a ⩽ b                     = Root a (u : ts)
    | otherwise                 = Root b (t : us)
  carry _ _                     = Void
  sum Void u                    = u
  sum t Void                    = t
  sum (Root _ _) (Root _ _)     = Void
```

Here is our final implementation of priority queues. For reasons of space we omit the *Eq* and *Ord* instance declarations for trees.

```
instance PriorityQueue BinomialHeap where
  ∅                             = []
  ⦇a⦈                           = [Root a []]
  (⊎)                           = add
  splitMin q                    = case extractMin q of
    Infty                       → Infty
    Min (Root a ts) us          → Min a (reverse ts ⊎ us)
```

Having two different representations of the same structure we may now weigh up pros and cons of each. The binary coding of binomial heaps is more space economical. If we estimate the space usage of the term $C\ e_1\ \ldots\ e_k$ at $k+1$ cells, a binary heap of size $n \geqslant 1$ consumes $3 + 4(n-1)$ cells, whereas a multiway heap requires $3 + 6(n-1)$ cells. On the other hand linking and splitting is slightly faster with the second representation. Linking two pennants requires 7 cells and produces 6 cells of garbage whereas linking two binomial trees requires only 6 cells producing 3 cells of garbage. Furthermore, since binomial heaps use lists both for representing the subtrees of a tree and for the entire forest the dismantling step becomes superfluous. Practical experience shows that the advantages and disadvantages nearly counterbalance each other: binary heaps are only marginally faster. They win the race, however, due to the lower space requirements.

8 Further reading

Tournament trees and loser trees already appear in (Knuth, 1973). Binomial heaps were discovered by Vuillemin (1978). Readers interested in an average case analysis

of insertion and deletion are referred to (Brown, 1978). Many variants and refinements of binomial heaps have been developed. An implicit array-based representation of binomial heaps (Carlsson *et al.*, 1988), for example, employs *heap-ordered* pennants. Binomial heaps also form the basis for an implementation of min-max priority queues (Khoong & Leong, 1993). We have seen that the loser trees in binary binomial heap are always perfectly balanced. Høyer (1994) shows that we may relax this condition and use some form of height-balancing instead.

The first functional implementation of binomial heaps is due to King (1994); (King, 1996, p. 28–42) additionally contains a simple proof of correctness. Okasaki (1996b) shows how to turn the amortized $\Theta(1)$ time bound for *insert* into a worst-case bound by scheduling delayed computations. Alternatively, one can employ a non-standard number system which avoids cascading carries: Such a variant based on skew binary numbers (Myers, 1983) is given in (Brodal & Okasaki, 1996). In fact, *loc.cit.* presents an optimal implementation of priority queues. Constant worst-case running time for $(\uplus)$ is achieved using a technique called data-structural bootstrapping (Buchsbaum, 1993). In essence $(\uplus)$ is reduced to *insert* by allowing queues to contain other queues.

9 Acknowledgements

Thanks are due to Michael Beetz, Thomas Bode, Martina Doelp, Ulrike Griefahn, Anja Hartmann, Jürgen Kalinski, and an anonymous referee for their helpful comments on a draft version of this article.

References

Brodal, Gerth Stølting, & Okasaki, Chris. (1996). Optimal purely functional priority queues. *Journal of functional programming*, **6**(6), 839–857.

Brown, Mark R. (1978). Implementation and analysis of binomial queue algorithms. *SIAM journal on computing*, **7**(3), 298–319.

Buchsbaum, Adam L. 1993 (June). *Data-structural bootstrapping and catenable deques.* Ph.d. thesis, Department of Computer Science, Princeton University.

Carlsson, Svante, Munro, J. Ian, & Poblete, Patricio V. (1988). An implicit binomial queue with constant insertion time. *Pages 1–13 of: Proceedings of 1st Scandinavian workshop on algorithm theory.* Lecture Notes in Computer Science, vol. 318. Springer-Verlag.

Cormen, Thomas H., Leiserson, Charles E., & Rivest, Ronald L. (1991). *Introduction to algorithms.* Cambridge, Massachusetts: The MIT Press.

Høyer, Peter. 1994 (October). *A general technique for implementation of efficient priority queues.* Tech. rept. PP-1994-33. Department of Mathematics and Computer Science, Odense University.

Jones, Mark P. (1995). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming*, **5**(1), 1–35.

Khoong, C. M., & Leong, H. W. (1993). Double-ended binomial queues. *Pages 128–137 of: Proceedings 4th international symposium ISAAC'93.* Lecture Notes in Computer Science, no. 762. Springer Verlag.

King, D. J. (1994). Functional binomial queues. Hammond, K., Turner, D. N., & Sansom, P. M. (eds), *Glasgow functional programming workshop*. Ayr, Scotland: Springer Verlag.

King, D.J. 1996 (March). *Functional programming and graph algorithms*. Ph.d. thesis, Department of Computer Science, University of Glasgow.

Knuth, Donald E. (1968). *The art of computer programming, Volume 1: Fundamental algorithms*. Addison-Wesley Publ. Comp., Inc.

Knuth, Donald E. (1973). *The art of computer programming, Volume 3: Sorting and searching*. Addison-Wesley Publ. Comp., Inc.

Myers, Eugene W. (1983). An applicative random-access stack. *Information processing letters*, **17**(5), 241–248.

Okasaki, Chris. 1996a (September). *Purely functional data structures*. Ph.d. thesis, School of Computer Science, Carnegie Mellon University.

Okasaki, Chris. 1996b (May). The role of lazy evaluation in amortized data structures. *Pages 62–72 of: ACM SIGPLAN international conference on functional programming*.

Peterson, J., & Hammond, K. 1997 (March). *Report on the programming language Haskell 1.4, a non-strict, purely functional language*. Research Report YALEU/DCS/RR-1106. Yale University, Department of Computer Science.

Sack, Jörg-Rüdiger, & Strothotte, Thomas. (1990). A characterization of heaps and its applications. *Information and computation*, **86**(1), 69–86.

Vuillemin, Jean. (1978). A data structure for manipulating priority queues. *Communications of the ACM*, **21**(4), 309–315.

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:

```
zero ++ p  =  p
p ++ zero  =  p
p ++ (q ++ r)  =  (p ++ q) ++ r
```

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:

```
zero >>= f  =  zero
p >>= const zero  =  zero
(p ++ q) >>= f  =  (p >>= f) ++ (q >>= f)
p >>= (\a -> f a ++ g a)  =  (p >>= f) ++ (p >>= g)
```

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. Functional Programming **1** (1): 1–000, January 1998. Printed in the United Kingdom

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 $\cos x$,

$$1 - x^2/2! + x^4/4! - x^6/6! + \cdots,$$

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,

$$F(x) = f_0 + xf_1 + x^2 f_2 + \cdots,$$

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 `Rational`s, 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 .*                    -- same precedence as *
(.*):: 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 cleanness of the stream formulation is now apparent. Gone is all the finicky indexing of the usual convolution formula,

$$(\sum_{i=0}^{\infty} f_i x^i)(\sum_{i=0}^{\infty} g_i x^i) = \sum_{i=0}^{\infty} x^i \sum_{j=0}^{j=i} f_j g_{i-j}.$$

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 nth 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.†

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

† 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.

`Integer`s 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, $1 + x + x^2 + x^3 + \cdots$, 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

$$\frac{1}{(1-x)^2} = \frac{d}{dx}\frac{1}{1-x} = \frac{d}{dx}(1 + x + x^2 + x^3 + \ldots) = 1 + 2x + 3x^2 + \ldots$$

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

$$\begin{array}{ll} \frac{d}{dx}\sin x = \cos x, & \sin(0) = 0, \\ \frac{d}{dx}\cos x = -\sin x, & \cos(0) = 1, \end{array}$$

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 $F = x^2F_2$, then $Q = xF_2^{1/2}$. 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 $n >= 0$, 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 $1+\mathbf{tree}+\mathbf{tree^2}+\ldots$, 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 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. Karczmarczuk (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)))
iszero n fs = (take n fs) == (take n ps0)
main = (iszero 30 test1) && (iszero 30 test2)
```

J. Functional Programming **7** (5): 549–554, September 1997. Printed in the United Kingdom

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 adresses 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 surrouding context.

Example. Assume that we consider the parse tree of arithmetic expressions, with string items. The expression $a \times b + c \times d$ 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: $F(T_1, ..., T_n)$ 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 $Node(F, i)$, with $1 \leq i \leq n$, each of arity n, used when going down the i-th subtree of an F-term. More precisely, $Node(F, i)$ 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.inria.fr:INRIA/Projects/cristal`)

Paulson, L. C. (1991) *ML for the Working Programmer*. Cambridge University Press.

FUNCTIONAL PEARLS

The Third Homomorphism Theorem

Jeremy Gibbons
Department of Computer Science
University of Auckland
Private Bag 92019, Auckland, New Zealand.
Email: jeremy@cs.auckland.ac.nz

Abstract

The *Third Homomorphism Theorem* is a folk theorem of the constructive algorithmics community. It states that a function on lists that can be computed both from left to right and from right to left is necessarily a *list homomorphism*—it can be computed according to *any* parenthesization of the list.

We formalize and prove the theorem, and use it to improve an $O(n^2)$ sorting algorithm to $O(n \log n)$.

1 Introduction

List homomorphisms are those functions on finite lists that *promote* through list concatenation—that is, functions h for which there exists a binary operator $\odot$ such that, for all finite lists x and y,

$$h\ (x +\!\!+\ y) \quad = \quad h\ x \odot h\ y$$

where '$+\!\!+$' denotes list concatenation. Such functions are ubiquitous in functional programming. Some examples of list homomorphisms are:

- the identity function *id*;
- the map function *map* f, which applies a given function f to every element of a list;
- the function *concat*, which concatenates a list of lists into a single long list;
- the function *head*, which returns the first element of a list;
- the function *length*, which returns the length of a list;
- the functions *sum*, *min* and *all*, which return the sum, the smallest and the boolean conjunction of the elements of a list, respectively.

However, there are also many useful list functions that are not list homomorphisms. One example is the function *lsp*, which returns the longest sorted prefix of

a list. Knowing $lsp\ x$ and $lsp\ y$ is not enough to allow computation of $lsp\ (x +\!\!+ y)$. This function is a typical example of a *leftwards* function—one which can be computed from right to left. Dually, the *rightwards* functions can be computed from left to right.

One obvious relationship between homomorphisms and leftwards and rightwards functions is known as the Specialization Theorem (Bird, 1987): all homomorphisms are also leftwards and rightwards functions. In the Constructive Algorithmics community, this has become known as the 'Second Homomorphism Theorem'. (The 'First Homomorphism Theorem' states that a homomorphism can be factored into the composition of reduction—a homomorphism whose value on a singleton list is the sole element of that list—with a map, and conversely that any such composition is a homomorphism.)

The subject of this paper is another relationship between homomorphisms and leftwards and rightwards functions. This relationship is much less obvious, but is equally useful. It is the converse of the Specialization Theorem, and states that any function that is both leftwards and rightwards is also a homomorphism. This theorem is fairly well-known in the Constructive Algorithmics community, bearing the name 'The Third Homomorphism Theorem'. However, it has somewhat the status of a 'folk theorem' (Harel, 1980). It was conjectured by Richard Bird and proved by Lambert Meertens during a train journey across the Netherlands in 1989 (Meertens, 1995); the theorem has been published only in non-archival sources (Barnard *et al.*, 1991; Gibbons, 1993), and we feel that it deserves wider recognition.

In this paper we formalize and prove this theorem, and use it to derive 'mergesort' from 'insertsort'. The remainder of this paper is structured as follows. In Section 2, we introduce the necessary notation. In Section 3, we state the First and Second Homomorphism Theorems, for completeness' sake. Section 4 contains the main result of the paper, the Third Homomorphism Theorem. In Section 5, we use the theorem to derive mergesort from insertsort.

An earlier version of this paper appeared as (Gibbons, 1994).

2 Notation

In this section, we introduce the notation used in the rest of the paper.

Functions: Function application is denoted by juxtaposition, is tightest binding, and associates to the left. Function composition is written '$\circ$'.

Lists: For the purposes of this paper, lists are finite sequences of elements, all of the same type. A list is either empty, a singleton, or the concatenation of two other lists. We write '$[\,]$' for the empty list, '$[a]$' for the singleton list with element a (and '$[\cdot]$' for the function taking a to $[a]$), and '$x +\!\!+ y$' for the concatenation of x and y. Concatenation is associative, and $[\,]$ is its unit. For example, the term $[a_1] +\!\!+ [a_2] +\!\!+ [a_3]$ denotes a list with three elements, often written in the abbreviated form $[a_1, a_2, a_3]$. We also write '$a : x$' for $[a] +\!\!+ x$; the operator ':' associates to the right.

Homomorphisms: For a binary operator $\odot$, the list function h is $\odot$-*homomorphic* iff, for all lists x and y,

$$h\ (x +\!\!+ y) \quad = \quad h\ x\ \odot\ h\ y$$

For example, the functions *length* and *sum* are both $+$-homomorphic, since

$$\begin{array}{rcl} sum\ (x +\!\!+ y) & = & sum\ x\ +\ sum\ y \\ length\ (x +\!\!+ y) & = & length\ x\ +\ length\ y \end{array}$$

Note that $\odot$ is necessarily associative on the range of h, because $+\!\!+$ is associative. Moreover, $h\ [\,]$ is necessarily the unit of $\odot$ on the range of h (if it exists), because $[\,]$ is the unit of $+\!\!+$. If $\odot$ has no unit, then $h\ [\,]$ is not defined. For example, *head* is $\ll$-homomorphic where $a \ll b = a$, but because $\ll$ has no unit, *head* $[\,]$ is undefined.

For associative operator $\odot$ with unit e, we write '*hom* $(\odot)\ f\ e$' for the (unique) $\odot$-homomorphic function h for which $h \circ [\cdot] = f$. For example,

$$\begin{array}{rcl} sum & = & hom\ (+)\ id\ 0 \\ length & = & length = hom\ (+)\ one\ 0 \end{array}$$

where $one\ a = 1$ for all a.

Leftwards and rightwards functions: The list function h is $\oplus$-*leftwards* for binary operator $\oplus$ iff, for all elements a and lists y,

$$h\ ([a] +\!\!+ y) \quad = \quad a \oplus h\ y$$

Here, $\oplus$ need not be associative. The (unique) $\oplus$-leftwards function h for which $h\ [\,] = e$ is written '*foldr* $(\oplus)\ e$'. For example, the function *lsp* referred to earlier is $\oplus$-leftwards where

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

Since $lsp\ [\,] = [\,]$, we have $lsp = foldr\ (\oplus)\ [\,]$ with the above definition of $\oplus$. Expanding the definition of a leftwards function reveals the significance of the name. For example:

$$foldr\ (\oplus)\ e\ [a_1, a_2, a_3] \quad = \quad a_1 \oplus (a_2 \oplus (a_3 \oplus e))$$

and so its computation proceeds from right to left. In general:

$$foldr\ (\oplus)\ e\ (x +\!\!+ y) \quad = \quad foldr\ (\oplus)\ (foldr\ (\oplus)\ e\ y)\ x \qquad (1)$$

(The name '*foldr*' is unfortunate for a right-to-left computation, but it is well established.)

Symmetrically, the list function h is $\otimes$-*rightwards* for binary operator $\otimes$ iff, for all lists x and elements a,

$$h\ (x +\!\!+ [a]) \quad = \quad h\ x\ \otimes\ a$$

Again, the operator $\otimes$ need not be associative. We write '*foldl* $(\otimes)\ e$' for the

unique $\otimes$-rightwards function h for which $h\ [\,] = e$. Expanding the definition reveals a left-to-right pattern of computation. For example:

$$\mathit{foldl}\ (\otimes)\ e\ [a_1, a_2, a_3] \quad = \quad ((e \otimes a_1) \otimes a_2) \otimes a_3$$

and in general:

$$\mathit{foldl}\ (\otimes)\ e\ (x +\!\!+\ y) \quad = \quad \mathit{foldl}\ (\otimes)\ (\mathit{foldl}\ (\otimes)\ e\ x)\ y \tag{2}$$

3 The First and Second Homomorphism Theorems

For the sake of completeness, we state without proof the First and Second Homomorphism Theorems.

Definition 3.1
A function of the form $\mathit{hom}\ (\odot)\ \mathit{id}\ e$ for some $\odot$ is called a *reduction*.

Definition 3.2
For given f, the function $\mathit{hom}\ (+\!\!+)\ ([\cdot] \circ f)\ [\,]$ is written '$\mathit{map}\ f$' and called a *map*.

Theorem 3.3 (First Homomorphism Theorem)
Every homomorphism can be written as the composition of a reduction and a map:

$$\mathit{hom}\ (\odot)\ f\ e \quad = \quad \mathit{hom}\ (\odot)\ \mathit{id}\ e \circ \mathit{map}\ f$$

Conversely, every such composition is a homomorphism.

Theorem 3.4 (Second Homomorphism Theorem, or Specialization Theorem)
Every homomorphism is both a leftwards and a rightwards function. That is, if $\odot$ is associative, then

$$\begin{array}{rcll} \mathit{hom}\ (\odot)\ f\ e & = & \mathit{foldr}\ (\oplus)\ e & \text{where } a \oplus s = f\ a \odot s \\ & = & \mathit{foldl}\ (\otimes)\ e & \text{where } r \otimes a = r \odot f\ a \end{array}$$

4 The Third Homomorphism Theorem

This section contains the main result of the paper, the statement and proof of the Third Homomorphism Theorem.

Theorem 4.1 (Third Homomorphism Theorem)
If h is leftwards and rightwards, then h is a homomorphism.

In fact, we will show that h is $\odot$-homomorphic where

$$t \odot u \quad = \quad h\ (g\ t +\!\!+ g\ u)$$

for some g such that $h \circ g \circ h = h$. Such a g exists, as the following lemma shows.

Lemma 4.2
For every computable total function h with enumerable domain, there is a computable (but possibly partial) function g such that $h \circ g \circ h = h$.

Proof
Here is one suitable definition of g. To compute $g\ t$ for some t, simply enumerate the domain of h and return the first x such that $h\ x = t$. If t is in the range of h, then this process terminates. □

The proof of the Third Homomorphism Theorem relies on the following lemma:

Lemma 4.3
The list function h is a homomorphism iff the implication

$$h\ v = h\ x\ \wedge\ h\ w = h\ y \quad \Rightarrow \quad h\ (v +\!\!+ w) = h\ (x +\!\!+ y) \tag{3}$$

holds for all lists v, w, x, y.

(We note in passing an interesting corollary to Lemma 4.3: any injective function is homomorphic.)

Proof
The 'only if' is obvious: if h is a homomorphism, then there is a $\oplus$ such that $h\ (x +\!\!+ y) = h\ x \oplus h\ y$ for all lists x and y. Now consider the 'if' part.

Assume that h satisfies (3). Choose a g such that $h \circ g \circ h = h$, and define operator $\odot$ by the equation

$$t \odot u \quad = \quad h\ (g\ t +\!\!+ g\ u)$$

(as in the statement of the Third Homomorphism Theorem). We show that h is $\odot$-homomorphic.

Because of the way we chose g, $h\ x = h\ (g\ (h\ x))$ and $h\ y = h\ (g\ (h\ y))$, and so, by (3) (with $v = g\ (h\ x)$ and $w = g\ (h\ y)$), we have

$$\begin{aligned} h\ (x +\!\!+ y) \quad &= \quad h\ (g\ (h\ x) +\!\!+ g\ (h\ y)) \\ &= \quad h\ x \odot h\ y \end{aligned}$$

□

We now prove the Third Homomorphism Theorem.

Proof
We show that, if h is leftwards and rightwards, then h satisfies (3).

Suppose that $h = \mathit{foldr}\ (\oplus)\ e = \mathit{foldl}\ (\otimes)\ e$, and that $h\ v = h\ x$ and $h\ w = h\ y$. Then:

$$\begin{array}{ll} & h\ (v +\!\!+ w) \\ = & \quad \{ \text{ treating } h \text{ as a leftwards function } \} \\ & \mathit{foldr}\ (\oplus)\ e\ (v +\!\!+ w) \\ = & \quad \{ \ (1) \ \} \\ & \mathit{foldr}\ (\oplus)\ (\mathit{foldr}\ (\oplus)\ e\ w)\ v \\ = & \quad \{ \text{ since } h\ w = h\ y \ \} \\ & \mathit{foldr}\ (\oplus)\ (\mathit{foldr}\ (\oplus)\ e\ y)\ v \\ = & \quad \{ \ (1) \ \} \\ & \mathit{foldr}\ (\oplus)\ e\ (v +\!\!+ y) \\ = & \quad \{ \text{ treating } h \text{ as a leftwards function } \} \\ & h\ (v +\!\!+ y) \\ = & \quad \{ \text{ symmetrically, treating } h \text{ as a rightwards function } \} \\ & h\ (x +\!\!+ y) \end{array}$$

Hence, by Lemma 4.3, h is a homomorphism. □

5 Application: sorting

We now use the Third Homomorphism Theorem to derive the $O(n \log n)$ sorting algorithm 'mergesort' from the $O(n^2)$ 'insertsort'. (In fact, the Third Homomorphism Theorem yields only an inefficient homomorphic sorting algorithm; we have to do a little more work to derive mergesort itself.)

The function *sort*, which sorts a list, is leftwards, since it can be written

$$\mathit{sort} \ = \ \mathit{foldr}\ \mathit{ins}\ [\,]$$

where

$$\begin{array}{rcl} \mathit{ins}\ a\ [\,] & = & [a] \\ \mathit{ins}\ a\ (b : x) & = & \begin{cases} a : b : x, & \text{if } a \leq b \\ b : (\mathit{ins}\ a\ x), & \text{otherwise} \end{cases} \end{array}$$

This is just traditional 'insertsort', and takes $O(n^2)$ time to sort n elements.

The same function is also rightwards, since it can be written as a 'backwards insertsort':

$$\mathit{sort} \ = \ \mathit{foldl}\ \mathit{ins}'\ [\,] \tag{4}$$

where

$$\mathit{ins}'\ x\ a \ = \ \mathit{ins}\ a\ x$$

The Third Homomorphism Theorem concludes that *sort* is therefore homomorphic. The homomorphism constructed by the proof is $hom\ (\odot)\ [\cdot]\ []$ where

$$u \odot v \quad = \quad sort\ (unsort\ u +\!\!+ unsort\ v)$$

for some function *unsort* such that $sort \circ unsort \circ sort = sort$, that is, which permutes the elements of a list.

We pick $unsort = id$ for simplicity, giving

$$u \odot v \quad = \quad sort\ (u +\!\!+ v) \tag{5}$$

This gives us a homomorphic method of sorting, but clearly it is very inefficient. To sort $x +\!\!+ y$, we sort x and y (yielding u and v), concatenate u and v, and then (presumably using some other sorting method, such as insertsort) sort the result $u +\!\!+ v$. However, we can improve this algorithm, by capitalizing on the fact that—in the context of evaluating $hom\ (\odot)\ [\cdot]\ []$—the arguments u and v to $\odot$ will be sorted. This improvement takes us directly to the traditional 'mergesort' algorithm, which is $O(n \log n)$.

Suppose first that u is sorted, that is, that $u = sort\ u$. Then

$$\begin{array}{ll}
 & u \odot v \\
= & \quad \{\ (5)\ \} \\
 & sort\ (u +\!\!+ v) \\
= & \quad \{\ (4)\ \} \\
 & foldl\ ins'\ []\ (u +\!\!+ v) \\
= & \quad \{\ (2)\ \} \\
 & foldl\ ins'\ (foldl\ ins'\ []\ u)\ v \\
= & \quad \{\ (4)\ \} \\
 & foldl\ ins'\ (sort\ u)\ v \\
= & \quad \{\ u \text{ is sorted}\ \} \\
 & foldl\ ins'\ u\ v \\
= & \quad \{\ \text{let } merge = foldl\ ins'\ \} \\
 & merge\ u\ v
\end{array}$$

We have picked a suggestive name in the last step, but it is justified by the observation that

$$\begin{array}{rcl}
merge\ u\ [] & = & foldl\ ins'\ u\ [] \\
 & = & u
\end{array}$$

and

$$\begin{array}{rcl}
merge\ u\ (b : v) & = & foldl\ ins'\ u\ (b : v) \\
 & = & foldl\ ins'\ (ins'\ u\ b)\ v \\
 & = & merge\ (ins'\ u\ b)\ v
\end{array}$$

This is a straightforward method of merging two lists, the first one already sorted, to

produce a sorted list. Note, however, that it takes quadratic time, and so computing *hom merge* $[\cdot]$ $[]$ also takes quadratic time†. We can make a further improvement by assuming that the second argument to *merge* is also sorted.

We use the following lemma, which is easily proved by induction. We write '$a \leq v$' to denote that $a \leq b$ for every element b of list v.

Lemma 5.1
If $a \leq x$ and $a \leq y$, then

$$foldl\ ins'\ (a : x)\ y \quad = \quad a : foldl\ ins'\ x\ y$$

Suppose that v is sorted. Then

$$\begin{array}{ll} & merge\ []\ v \\ = & \qquad \{\ \text{definition of } merge\ \} \\ & foldl\ ins'\ []\ v \\ = & \qquad \{\ (4)\ \} \\ & sort\ v \\ = & \qquad \{\ v \text{ is sorted}\ \} \\ & v \end{array}$$

Now suppose that $a : u$ and $b : v$ are sorted. Then

$$\begin{array}{ll} & merge\ (a : u)\ (b : v) \\ = & \qquad \{\ \text{definition of } merge\ \} \\ & foldl\ ins'\ (a : u)\ (b : v) \\ = & \qquad \{\ \text{defining property of } foldl\ \} \\ & foldl\ ins'\ (ins'\ (a : u)\ b)\ v \end{array}$$

We now consider the cases $a < b$ and $a \geq b$ separately.

Case $a < b$**:** Since $a : u$ and $b : v$ are sorted, we have $a \leq u$ and $a \leq v$; hence $a \leq ins'\ u\ b$ also. Then

$$\begin{array}{ll} & foldl\ ins'\ (ins'\ (a : u)\ b)\ v \\ = & \qquad \{\ ins';\ a < b\ \} \\ & foldl\ ins'\ (a : ins'\ u\ b)\ v \\ = & \qquad \{\ \text{Lemma 5.1}\ \} \\ & a : foldl\ ins'\ (ins'\ u\ b)\ v \\ = & \qquad \{\ \text{defining property of } foldl\ \} \\ & a : foldl\ ins'\ u\ (b : v) \\ = & \qquad \{\ \text{definition of } merge\ \} \\ & a : merge\ u\ (b : v) \end{array}$$

† because $\sum_{i=0}^{\log n} 2^i (\frac{n}{2^i})^2 \simeq 2n^2$

Case $a \geq b$**:** Since $a : u$ and $b : v$ are sorted, we have $b \leq a : u$ and $b \leq v$. Then

$$\begin{array}{ll} & \mathit{foldl}\ \mathit{ins}'\ (\mathit{ins}'\ (a : u)\ b)\ v \\ = & \qquad \{\ \mathit{ins}';\ a \geq b\ \} \\ & \mathit{foldl}\ \mathit{ins}'\ (b : a : u)\ v \\ = & \qquad \{\ \text{Lemma 5.1}\ \} \\ & b : \mathit{foldl}\ \mathit{ins}'\ (a : u)\ v \\ = & \qquad \{\ \text{definition of } \mathit{merge}\ \} \\ & b : \mathit{merge}\ (a : u)\ v \end{array}$$

We have just derived the following characterization of *merge*, when both of its arguments are sorted:

$$\begin{array}{rcl} \mathit{merge}\ [\,]\ v & = & v \\ \mathit{merge}\ u\ [\,] & = & u \\ \mathit{merge}\ (a : u)\ (b : v) & = & \begin{cases} a : \mathit{merge}\ u\ (b : v), & \text{if } a < b \\ b : \mathit{merge}\ (a : u)\ v, & \text{otherwise} \end{cases} \end{array}$$

which is the standard way of merging two sorted lists (except that the comparison is usually '$a \leq b$' rather than '$a < b$'). This version of *merge* takes linear time, and yields the well-known mergesort algorithm, which is $O(n \log n)$ when the list is decomposed in a balanced fashion. Green and Barstow (1978) describe a similar derivation of *merge* and mergesort.

6 Conclusion

To summarize, we have presented and proved Bird and Meertens' Third Homomorphism Theorem, stating that any function on lists that can be computed both from left to right and from right to left is necessarily a list homomorphism. We gave an example of its use—deriving 'mergesort' from 'insertsort'—illustrating that the theorem does not usually give an efficient characterization of the homomorphism; further development must be done to produce this.

Further applications of the Third Homomorphism Theorem are given by Barnard *et al.* (1991), Gibbons (1993), and Gorlatch (1995).

Acknowledgements

Murray Cole, Rod Downey, Sergei Gorlatch, Lindsay Groves, Lambert Meertens, the participants of the *Computing—the Australian Theory Seminar* in Sydney in December 1994, and especially Richard Bird have all made comments to improve the presentation and content of this paper. Thanks are also due to Sue Gibbons, for her energetic red pen. The research reported here has been partially supported by University of Auckland Research Committee grant number 3414013.

References

Barnard, D. T., Schmeiser, J. P. and Skillicorn, D. B. 1991. Deriving associative operators for language recognition. *Bulletin of the European Association for Theoretical Computer Science*, 43: pp. 131–139.

Bird, R. S. 1987. An introduction to the theory of lists. In M. Broy (editor), *Logic of Programming and Calculi of Discrete Design*, pp. 3–42. Springer-Verlag. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.

Gibbons, J. 1993. Computing downwards accumulations on trees quickly. In G. Gupta, G. Mohay, and R. Topor (editors), *16th Australian Computer Science Conference*, pp. 685–691, Brisbane. Available by anonymous `ftp` as `out/jeremy/papers/quickly.ps.Z` on `ftp.cs.auckland.ac.nz`.

Gibbons, J. 1994. The Third Homomorphism Theorem. In C. Barry Jay (editor), *Computing: The Australian Theory Seminar*. University of Technology, Sydney.

Gorlatch, S. 1995. Constructing List Homomorphisms. Technical Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau.

Green, C. and Barstow, B. 1978. On program synthesis knowledge. *Artificial Intelligence*, 10: pp. 241–279.

Harel, D. 1980. On folk theorems. *Communications of the ACM*, 23(7): pp. 379–389.

Meertens, L. G. L. T. 1995. Personal communication.

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:

```
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:

$$1 + a + a^2 + a^3 + \ldots$$

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 ∞ 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:

$$I + M + M^2 + M^3 + \ldots$$

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 1×1 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 ∞, 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 ∞:

```
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
  LDistance x @+ LDistance y = LDistance (max x y)

  x @. LUnreachable = LUnreachable
  LUnreachable @. x = LUnreachable
  LInfinite @. _ = LInfinite
  _ @. LInfinite = LInfinite
  LDistance x @. LDistance y = LDistance (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 $x_1, \ldots, x_n$, where each x_i is defined by a linear combination of the variables and a constant term. That is, each equation is of the following form, where a_{ij} and b_i are givens:

$$x_i = a_{i1}x_1 + a_{i2}x_2 + \cdots + b_i$$

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 $A \to xB$, and for every accepting state q_A we have a production $A \to \varepsilon$.

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 $\mathbf{M}$ where $\mathbf{M}_{ij}$ is the symbol on the transition from the ith state to the jth, or 0 (the empty language) if there is no such transition. The vector $\mathbf{A}$ is constructed by setting $\mathbf{A}_i$ to 1 (the language containing only ε) when the ith 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 ith state. For our example machine, $\mathbf{M}$ and $\mathbf{A}$ are shown at the right of Figure 1.

Then, the regular grammar is described by:

$$\mathbf{L} = \mathbf{M} \cdot \mathbf{L} + \mathbf{A}$$

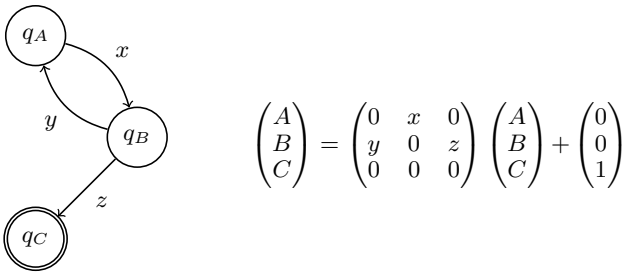


Figure 1. A finite state machine and its matrix representation

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 (IN_b) are those used by the basic block itself (USE_b) before their first definition, and those which are live at the end of the block (OUT_b) but not assigned a new value during the block (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:

$$\begin{aligned}\mathrm{IN}_b &= (\mathrm{OUT}_b \cap \overline{\mathrm{DEF}_b}) \cup \mathrm{USE}_b \\ \mathrm{OUT}_b &= \bigcup_{b' \in \mathrm{succ}(b)} \mathrm{IN}_{b'}\end{aligned}$$

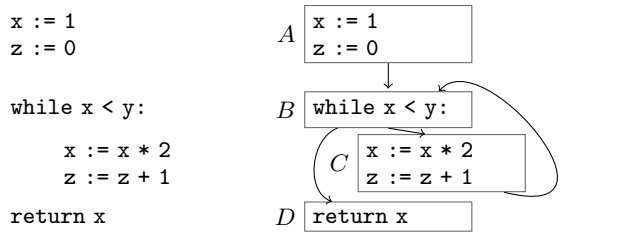


Figure 2. A simple imperative program to calculate the smallest power of two greater than the input `y`, and its control flow graph. The variable `z` does not affect the output.

An example program is given in Figure 2, where DEF and USE are as follows:

$$\begin{aligned}\mathrm{DEF}_A &= \{\mathtt{x}, \mathtt{z}\} & \mathrm{USE}_A &= \emptyset \\ \mathrm{DEF}_B &= \emptyset & \mathrm{USE}_B &= \{\mathtt{x}, \mathtt{y}\} \\ \mathrm{DEF}_C &= \{\mathtt{x}, \mathtt{z}\} & \mathrm{USE}_C &= \{\mathtt{x}, \mathtt{z}\} \\ \mathrm{DEF}_D &= \emptyset & \mathrm{USE}_D &= \{\mathtt{x}\}\end{aligned}$$

If we solve for IN_b and OUT_b, we find that `z` is not live upon entry to D (that is, $\mathtt{z} \notin \mathrm{IN}_D$). 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:

$$\begin{aligned}\mathrm{OUT}_b &= \sum_{b' \in \mathrm{succ}(b)} \mathrm{IN}_{b'} \\ &= \sum_{b' \in \mathrm{succ}(b)} \overline{\mathrm{DEF}_{b'}} \cdot \mathrm{OUT}_{b'} + \mathrm{USE}_{b'} \\ &= \left(\sum_{b' \in \mathrm{succ}(b)} \overline{\mathrm{DEF}_{b'}} \cdot \mathrm{OUT}_{b'} \right) + \left(\sum_{b' \in \mathrm{succ}(b)} \mathrm{USE}_{b'} \right)\end{aligned}$$

This is a system of affine equations over the variables OUT_b, with coefficients $\overline{\mathrm{DEF}_{b'}}$ and constant terms $\sum_{b' \in \mathrm{succ}(b)} \mathrm{USE}_{b'}$. 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 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 IN_b given OUT_b (the discussion applies equally well to forwards analyses, with suitable relabelling).

The equations have the following form:

$$\begin{aligned}\mathrm{IN}_b &= f_b(\mathrm{OUT}_b) \\ \mathrm{OUT}_b &= \operatorname*{join}_{b' \in \mathrm{succ}(b)} \mathrm{IN}_{b'}\end{aligned}$$

or, more compactly,

$$\mathrm{OUT}_b = \operatorname*{join}_{b' \in \mathrm{succ}(b)} f(\mathrm{OUT}_{b'})$$

For many standard analyses, we can define a semiring where f_b is linear, and 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 $\mathtt{x} \in \mathrm{IN}_C$ if $\mathtt{x} \in \mathrm{OUT}_C$ (since `x` is used to compute the new value of `x`), and similarly $\mathtt{z} \in \mathrm{IN}_C$ if $\mathtt{z} \in \mathrm{OUT}_C$.

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 CommutativeMonoid 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 ith 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 $1 + 2x = 1 + 2x + 0x^2 + 0x^3 + \ldots$).

```
  [] @+ 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 $1 + x + x^2 + x^3 \ldots$ as the sequence $1, 1, 1, \ldots$, 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 $w_1, \ldots, w_n$ and nonnegative integer values $v_1, \ldots, v_n$. 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:

$$t(w) = \max_{0 \le w_i \le w} (v_i + t(w - w_i))$$

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 ∞, 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:

$$V = x^3 + 5x^6 + 7x^8 + 10x^6$$

Since we are using the max-plus semiring, this is equivalent to:

$$V = x^3 + 10x^6 + 7x^8$$

Represented as a list, the wth 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 wth 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 $1 + x + x^2 + \ldots$ 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 nth term of the Fibonacci sequence is given by the sum of the previous two terms. We construct the formal power series F whose nth coefficient is the nth 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.

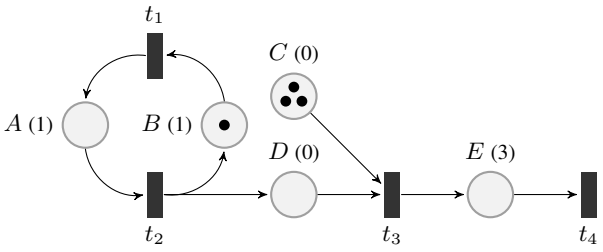


Figure 3. A timed event graph with four transitions t_1, t_2, t_3, t_4 and five places A, B, C, D, E with delays in parentheses, where all but two of the places are initially empty.

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 $1, 3, 5, \ldots$, 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 $\mathrm{IN}(p)$ and $\mathrm{OUT}(p)$. The ith element of $\mathrm{IN}(p)$ is the time at which a token arrives into p for the ith time. The ith element of $\mathrm{OUT}(p)$ is the time at which a token becomes available from p for the ith time, which may be some time before it actually leaves p.

In the example of Figure 3, we have:

$$\begin{aligned}
\mathrm{IN}(A) &= 0, 2, 4, 6 \ldots & \mathrm{OUT}(A) &= 1, 3, 5, 7 \ldots \\
\mathrm{IN}(B) &= 1, 3, 5, 7 \ldots & \mathrm{OUT}(B) &= 0, 2, 4, 6 \ldots \\
\mathrm{IN}(C) &= — & \mathrm{OUT}(C) &= 0, 0, 0 \\
\mathrm{IN}(D) &= 1, 3, 5, 7 \ldots & \mathrm{OUT}(D) &= 1, 3, 5, 7 \ldots \\
\mathrm{IN}(E) &= 1, 3, 5 & \mathrm{OUT}(E) &= 4, 6, 8
\end{aligned}$$

We say that a place p' is a predecessor of p (and write $p' \in \mathrm{pred}(p)$) 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.

$$\mathrm{IN}(p)_i = \max_{p' \in \mathrm{pred}(p)} \mathrm{OUT}(p')_i$$

Exactly when the ith token becomes available from a place p depends on the amount of time tokens spend processing at p, which we write as $\mathrm{delay}(p)$, and on the number of tokens initially in p, which we write as $\mathrm{nstart}(p)$. The times at which the first $\mathrm{nstart}(p)$ tokens become ready to leave p are given by the sequence $\mathrm{START}(p)$, which is nondecreasing and each element of which is less than $\mathrm{delay}(p)$. In the example we assume the initial tokens of B and C are immediately available, so we have $\mathrm{START}(B) = 0$ and $\mathrm{START}(C) = 0, 0, 0$.

Thus, the time that the ith token becomes available from p is given by:

$$\mathrm{OUT}(p)_i = \begin{cases} \mathrm{START(p)}_i & i < \mathrm{nstart}(p) \\ \mathrm{IN}(p)_{i-\mathrm{nstart}(p)} + \mathrm{delay}(p) & i \geq \mathrm{nstart}(p) \end{cases}$$

By adopting the convention that $\mathrm{IN}(p)_i$ is $-\infty$ when $i < 0$ and that $\mathrm{START}(p)_i$ is $-\infty$ when $i < 0$ or $i \geq \mathrm{nstart}(p)$, we can write the above more succinctly as:

$$\mathrm{OUT}(p)_i = \max(\mathrm{START(p)}_i, \mathrm{IN}(p)_{i-\mathrm{nstart}(p)} + \mathrm{delay}(p))$$

This gives a set of recurrences between the sequences: the value of $\mathrm{OUT}(p)$ depends on the previous values of $\mathrm{IN}(p)$.

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 ith element of the sequence is now the coefficient of x^i. With our semiring goggles on, the above equations now say:

$$\mathrm{IN}(p) = \sum_{p' \in \mathrm{pred}(p)} \mathrm{OUT}(p')$$

$$\mathrm{OUT}(p) = \mathrm{delay}(p) \cdot x^{\mathrm{nstart(p)}} \cdot \mathrm{IN}(p) + \mathrm{START}(p)$$

We can eliminate $\mathrm{IN}(p)$ by substituting its definition into the second equation:

$$\mathrm{OUT}(p) = \sum_{p' \in \mathrm{pred}(p)} \mathrm{delay}(p) \cdot x^{\mathrm{nstart(p)}} \cdot \mathrm{OUT}(p') + \mathrm{START}(p)$$

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 $\mathrm{delay}(p) \cdot x^{\mathrm{nstart}(p)}$ coefficients, and the column vector $\mathbf{S}$ containing all of the $\mathrm{START}(p)$ 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 $\mathbf{OUT}(p)$ 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 *dioids*. 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.