Comprehending Monads

Philip Wadler
University of Glasgow*

Abstract

Category theorists invented *monads* in the 1960's to concisely express certain aspects of universal algebra. Functional programmers invented *list comprehensions* in the 1970's to concisely express certain programs involving lists. This paper shows how list comprehensions may be generalised to an arbitrary monad, and how the resulting programming feature can concisely express in a pure functional language some programs that manipulate state, handle exceptions, parse text, or invoke continuations. A new solution to the old problem of destructive array update is also presented. No knowledge of category theory is assumed.

1 Introduction

Is there a way to combine the indulgences of impurity with the benefits of purity?

Impure, strict functional languages such as Standard ML [Mil84, HMT88] and Scheme [RC86] support a wide variety of features, such as assigning to state, handling exceptions, and invoking continuations. Pure, lazy functional languages such as Haskell [HW90] or Miranda[1] [Tur85] eschew such features, because they are incompatible with the advantages of lazy evaluation and equational reasoning, advantages that have been described at length elsewhere [Hug89, BW88].

Purity has its regrets, and all programmers in pure functional languages will recall some moment when an impure feature has tempted them. For instance, if a counter is required to generate unique names, then an assignable variable seems just the ticket. In such cases it is always possible to mimic the required impure feature by straightforward though tedious means. For instance, a counter can be simulated by modifying the relevant functions to accept an additional parameter (the counter's current value) and return an additional result (the counter's updated value).

This paper describes a new method for structuring pure programs that mimic impure features. This method does not completely eliminate the tension between purity and impurity, but it does relax it a little bit. It increases the readability of the resulting programs, and it eliminates the possibility of certain silly errors that might otherwise arise (such as accidentally passing the wrong value for the counter parameter).

The inspiration for this technique comes from the work of Eugenio Moggi [Mog89a, Mog89b]. His goal was to provide a way of structuring the semantic description of features such as state, exceptions, and continuations. His discovery was that the notion of a *monad* from category theory suits this purpose. By defining an interpretation of λ-calculus in an arbitrary monad he provided a framework that could describe all these features and more.

It is relatively straightforward to adopt Moggi's technique of structuring denotational specifications into a technique for structuring functional programs. This paper presents a simplified version of Moggi's ideas, framed in a way better suited to functional programmers rather than semanticists; in particular, no knowledge of category theory is assumed.

The paper contains two significant new contributions.

The first contribution is a new language feature, the *monad comprehension*. These generalise the familiar notion of list comprehension [Wad87], due originally to Burstall and Darlington, and found in KRC [Tur82], Miranda, Haskell and other languages. Monad comprehensions are not essential to the structuring technique described here, but they do provide a pleasant syntax for expressing programs structured in this way.

The second contribution is a new solution to the old problem of destructive array update. The solution consists of two abstract data types with ten operations between them. The usual typing discipline (e.g., Hindley-Milner extended with abstract data types) is sufficient to guarantee that array update may safely be imple-

*Author's address: Department of Computing Science, University of Glasgow, G12 8QQ, Scotland. Electronic mail: wadler@cs.glasgow.ac.uk.

[1] Miranda is a trademark of Research Software Limited.

mented by overwriting. To my knowledge, this solution has never been proposed before, and its existence is quite a surprise considering the plethora of more elaborate solutions that have been proposed: these include syntactic restrictions [Sch85], run-time checks [Hol83], abstract interpretation [Hud86a, Hud86b, Blo89], and exotic type systems [GH90, Wad90]. That monads led to the discovery of this solution must be counted a point in their favour.

Why has this solution not been discovered before? One likely reason is that the data types involve higher-order functions in an essential way. The usual axiomatisation of arrays involves only first-order functions (*index*, *update*, and *newarray*, as described in Section 4.3), and so, apparently, it did not occur to anyone to search for an abstract data type based on higher-order functions. Incidentally, the higher-order nature of the solution means that it cannot be applied in first-order languages such as Prolog or OBJ. It also casts doubt on Goguen's thesis that first-order languages are sufficient for most purposes [Gog88].

Monads and monad comprehensions help to clarify to unify some previous proposals for incorporating various features into functional languages: exceptions [Wad85, Spi89], parsers [Wad85, Fai87, FL89], and non-determinism [HO89]. (Spivey's work [Spi89] is notable for pointing out, independently of Moggi, that monads provide a framework for exception handling.)

There is a translation scheme from λ-calculus into an arbitrary monad, indeed, there are two schemes, one yielding call-by-value semantics and one yielding call-by-name. These can be used to systematically transform languages with state, exceptions, continuations, or other features, into a pure functional language. Two applications are given. One is to derive call-by-value and call-by-name interpretations for a simple non-deterministic language. The other is to apply the call-by-value scheme in the monad of continuations: the result is the familiar continuation-passing style transformation.

A key feature of the monad approach is the use of types to indicate what parts of a program may have what sorts of effects. In this, it is similar in spirit to Gifford and Lucassen's *effect systems* [GL88].

The examples in this paper are based on Haskell [HW90], though any lazy functional language incorporating the Hindley/Milner type system would work as well.

The remainder of this paper is organised as follows. Section 2 uses list comprehensions to motivate the concept of a monad, and introduces monad comprehensions. Section 3 shows that variable binding (as in "where" terms) and control of evaluation order can be modelled by two trivial monads. Section 4 explores the use of monads to structure programs that manipulate state, and presents the new solution to the array update problem. Two examples are considered: renaming bound variables, and interpreting a simple imperative language. Section 5 extends monad comprehensions to include filters. Section 6 introduces the concept of monad morphism and gives a simple proof of the equivalence of two programs. Section 7 catalogues three more monads: parsers, exceptions, and continuations. Section 8 gives the translation schemes for interpreting λ-calculus in an arbitrary monad. Two examples are considered: giving a semantics to a non-deterministic language, and deriving continuation-passing style.

2 Comprehensions and monads

2.1 Lists

Let us write $M\ x$ for the data type of lists with elements of type x. (In Haskell, $M\ x$ is written $[x]$.) For example, $[1,2,3] :: M\ Int$ and $[\text{'a'}, \text{'b'}, \text{'c'}] :: M\ Char$. We write *map* for the higher-order function that applies a function to each elment of a list:

$$map :: (x \to y) \to (M\ x \to M\ y).$$

(In Haskell, type variables are written with small letters, e.g., x and y, and type constructors are written with capital letters, e.g., M.) For example, if $code : Char \to Int$ maps a character to its ASCII code, then $map\ code\ [\text{'a'}, \text{'b'}, \text{'c'}] = [97, 98, 99]$; Observe that

$$\begin{array}{lrcl} (i) & map\ id & = & id, \\ (ii) & map\ (g \cdot f) & = & map\ g \cdot map\ f. \end{array}$$

Here id is the identity function, $id\ x = x$, and $g \cdot f$ is function composition, $(g \cdot f)\ x = g\ (f\ x)$.

In categorical terms, any operator M on types combined with an operator map taking functions $x \to y$ into functions $M\ x \to M\ y$ and satisfying (i) and (ii) is called a *functor*. Categorists prefer to use the same symbol for both the type operator and the function operator, so would write $M\ f$ where we write $map\ f$.

Two useful functions are one that converts a value into a singleton list and one that concatenates a list of lists into a list:

$$\begin{array}{l} unit :: x \to M\ x, \\ join :: M\ (M\ x) \to M\ x. \end{array}$$

For example, $unit\ 3 = [3]$ and $join\ [[1,2],[3]] = [1,2,3]$. Observe that

$$\begin{array}{lrcl} (iii) & map\ f \cdot unit & = & unit \cdot f, \\ (iv) & map\ f \cdot join & = & join \cdot map\ (map\ f). \end{array}$$

Indeed, laws (*iii*) and (*iv*) follow immediately from the types of *unit* and *join*; see [Rey83, Wad89] for an explanation.

In categorical terms, *unit* and *join* are *natural transformations.* Rather than treating *unit* as a single function with a polymorphic type, categorists treat it as a family of functions, $unit_x :: x \to M\,x$ satisfying $map\,f \cdot unit_x = unit_y \cdot f$ for every $f :: x \to y$, where x and y are any types; and they treat *join* similarly. Natural transformation is a simpler concept than polymorphic type, but this paper will stick with polymorphic types since they are more familiar (to functional programmers).

2.2 Comprehensions

Many functional languages provide a form of *list comprehension* analogous to set comprehension. For example,

$$\begin{aligned}&[\,(x,y) \mid x \leftarrow [1,2]; y \leftarrow [3,4]\,]\\ &\quad = [(1,3),(1,4),(2,3),(2,4)].\end{aligned}$$

In general, a comprehension has the form $[\,t \mid q\,]$, where t is a term and q is a qualifier. We will use the letters t, u, v to range over terms, and p, q, r to range over qualifiers. A qualifier is either empty, Λ; or a generator, $x \leftarrow u$, where x is a variable and u is a list-valued term; or a composition of qualifiers, $(p;q)$. Comprehensions are defined by the following rules:

$$\begin{array}{lrcl}(1) & [\,t \mid \Lambda\,] & = & unit\; t\\ (2) & [\,t \mid x \leftarrow u\,] & = & map\,(\lambda x \to t)\; u\\ (3) & [\,t \mid (p;q)\,] & = & join\,[\,[t|q] \mid p\,]\end{array}$$

(In Haskell, λ-terms are written $(\lambda x \to t)$ rather than the more common $(\lambda x.\, t)$.) Note the reversal of qualifiers in rule (3): nesting q inside p on the right-hand side means that, as we expect, variables bound in p may be used in q.

For those familiar with list comprehensions, the empty qualifier and the parentheses around qualifier compositions will appear strange. This is because they are not needed; qualifier composition is associative and has the empty qualifier as a unit. This will be proved shortly. Indeed, the only reason for here for including parentheses is to facilitate the proof that they are not required!

Most languages that include list comprehensions also allow another form of qualifier, known as a filter. Filters will be ignored for the time being, but returned to in Section 5.

(A footnote for categorists: Using λ-terms means that we are working within a cartesian closed category. Thus, for each pair of objects (types) x and y in the category, there is an object (type) $x \Rightarrow y$ representing the arrows from x to y. Since M (or, in out notation, *map*) may appear in the body of a λ-term, it must itself correspond to an arrow of the category, $M : (x \Rightarrow y) \to (M\,x \Rightarrow M\,y)$. Such an M is called a *strong functor.*)

As a simple example, we have:

$$\begin{array}{cl} & [\,sqr\,x \mid x \leftarrow [1,2,3]\,]\\ = & \{\text{by } (2)\}\\ & map\,(\lambda x \to sqr\,x)\,[1,2,3]\\ = & \{\text{reducing } map\}\\ & [1,4,9]\end{array}$$

The comprehension in the initial example is computed as:

$$\begin{array}{cl} & [\,(x,y) \mid x \leftarrow [1,2]; y \leftarrow [3,4]\,]\\ = & \{\text{by } (3)\}\\ & join\,[\,[\,(x,y) \mid y \leftarrow [3,4]\,] \mid x \leftarrow [1,2]\,]\\ = & \{\text{by } (2)\}\\ & join\,[\,map\,(\lambda y \to (x,y))\,[3,4] \mid x \leftarrow [1,2]\,]\\ = & \{\text{by } (2)\}\\ & join\,(map\,(\lambda x \to map\,(\lambda y \to (x,y))\,[3,4])\,[1,2])\\ = & \{\text{reducing } map\}\\ & join\,(map\,(\lambda x \to [(x,3),(x,4)])\,[1,2])\\ = & \{\text{reducing } map\}\\ & join\,[[(1,3),(1,4)],[(2,3),(2,4)]]\\ = & \{\text{reducing } join\}\\ & [(1,3),(1,4),(2,3),(2,4)]\end{array}$$

A useful property satisfied by list comprehensions is

$$(4) \qquad [\,f\,t \mid q\,] \;=\; map\,f\,[\,t \mid q\,],$$

for all functions f, terms t, and qualifiers q, so long as f contains no free variables bound by q. For example, $[\,sqr\,(x+y) \mid x \leftarrow [1,2];\; y \leftarrow [3,4]\,] = map\;sqr\,[\,x+y \mid x \leftarrow [1,2];\; y \leftarrow [3,4]\,]$. The proof is by induction on the structure of qualifiers, and uses laws (*ii*)–(*iv*). We also have that

$$(5) \qquad [\,x \mid x \leftarrow u\,] \;=\; u.$$

which follows from law (*i*).

2.3 Monads

Parentheses in qualifiers are not required, because qualifier composition is associative and has the empty qualifier as a unit:

$$\begin{array}{lrcl}(I') & [\,t \mid \Lambda; q\,] & = & [\,t \mid q\,],\\ (II') & [\,t \mid q; \Lambda\,] & = & [\,t \mid q\,],\\ (III') & [\,t \mid (p;q); r\,] & = & [\,t \mid p; (q;r)\,].\end{array}$$

For example, the left-hand side of rule (II') simplifies to

$$\begin{aligned} & [\,t \mid q; \Lambda\,] \\ = \; & \{\text{by } (3)\} \\ & join\,[\,[\,t \mid \Lambda\,] \mid q\,] \\ = \; & \{\text{by } (1)\} \\ & join\,[\,unit\;t \mid q\,] \\ = \; & \{\text{by } (4)\} \\ & join\,(map\;unit\,[\,t \mid q\,]) \end{aligned}$$

Taking $[\,t \mid q\,]$ to be $[\,x \mid x \leftarrow u\,]$ and applying (5), it follows that (II') is equivalent to $join \cdot map\;unit = id$. Using a similar procedure for the other two rules, we have that (I')–(III') are equivalent, respectively, to

$$\begin{array}{lrcl} (I) & join \cdot unit & = & id, \\ (II) & join \cdot map\;unit & = & id, \\ (III) & join \cdot join & = & join \cdot map\;join. \end{array}$$

Laws (I)–(III) do indeed hold for lists. For example:

$$\begin{aligned} join\,(unit\,[1,2]) &= join\,[[1,2]] = [1,2], \\ join\,(map\;unit\,[1,2]) &= join\,[[1],[2]] = [1,2], \\ join\,(join\,[[[1],[2]],[[3]]]) &= join\,[[1],[2],[3]] = [1,2,3], \\ join\,(map\;join\,[[[1],[2]],[[3]]]) &= join\,[[1,2],[3]] = [1,2,3]. \end{aligned}$$

Hence, it is sensible to omit parentheses around qualifiers in list comprehensions. For instance, $[\,t \mid p; q; r\,]$ stands for either side of equation (III'). It is safe to drop empty qualifiers, so the only remaining comprehension containing an empty qualifier is $[\,t \mid \Lambda\,]$, which we will abbreviate as $[\,t\,]$.

Obviously, the comprehension notation is sensible for *any* operator M on types together with functions *map*, *unit*, and *join* of the appropriate types satisfying laws (i)–(iv) and (I)–(III). Such a triple $(map, unit, join)$ is called a ***monad*** by category theorists [Mac71, LS86]. (Less imaginatively, it has also been called a "triple" [BW85].)

In the following, we will often write the type M alone to stand for the monad $(map, unit, join)$ where the functions can be understood from context. In particular, we will write *List* to stand for the monad of lists as described above. We will also write $[\,t \mid q\,]^M$ to indicate in which monad a comprehension is to be interpreted.

There is exactly one clause in the definition of comprehension for each component of a monad: rule (1) corresponds to *unit*, rule (2) to *map*, and rule (3) to *join*. Further, *unit*, *map*, and *join* can be expressed in terms of M-comprehensions:

$$\begin{array}{lcl} unit\;x & = & [\,x\,]^M \\ map\;f\;\overline{x} & = & [\,f\;x \mid x \leftarrow \overline{x}\,]^M \\ join\;\overline{\overline{x}} & = & [\,x \mid \overline{x} \leftarrow \overline{\overline{x}}; x \leftarrow \overline{x}\,]^M. \end{array}$$

Here and in what follows, we adopt the convention that if x has type x, then $\overline{x}$ has type $M\;x$ and $\overline{\overline{x}}$ has type $M\,(M\;x)$.

(Moggi's work assumes not only a monad, but a "strong monad" with some additional structure. As already noted, M is a strong functor, and it follows from this that all the monads we consider are strong monads. In particular the tensorial strength, $t :: (x, M\;y) \rightarrow M\,(x,y)$, is defined by $t\;x\;\overline{y} = [\,(x,y) \mid y \leftarrow \overline{y}\,]^M$.)

We conclude with one final law of comprehensions, the ***comprehension substitution*** law. Let t, u be terms, p, q, r be qualifiers, and x a variable. Then

$$(6) \qquad [\,t \mid p;\; x \leftarrow [u|q]^M;\; r\,]^M \;=\; [\,t^u_x \mid p; q; r^u_x\,]^M$$

where t^u_x and r^u_x are t and r with each instance of x replaced by u. This law follows from laws (1)–(4). Alternatively, if we take laws (5) and (6) as given, and define *unit*, *map*, and *join* in terms of M-comprehensions as shown above, then laws (i)–(iv), (1)–(4), and (I)–(III) all follow. Thus, one may choose whether to take monads as primitive (axiomatised by laws (i)–(iv) and (I)–(III)) and define comprehensions in terms of monads, or to take comprehensions as primitive (axiomatised by laws (5) and (6)) and define monads in terms of comprehensions.

Monads were conceived in the 1960's, list comprehensions in the 1970's. They have quite independent origins, but fit with each other remarkably well. As often happens, a common truth may underlie apparently disparate phenomena, and it may take a decade or more before this underlying commonality is unearthed.

3 Two trivial monads

3.1 The identity monad

The identity monad is the trivial monad specified by

$$\begin{array}{lcl} \text{type } Id\;x & = & x \\ map^{Id}\,f\;x & = & f\;x \\ unit^{Id}\,x & = & x \\ join^{Id}\,x & = & x, \end{array}$$

so map^{Id}, $unit^{Id}$, and $bind^{id}$ are all just the identity function. A comprehension in the identity monad is like a "where" term:

$$\begin{aligned} & [\,t \mid x \leftarrow u\,]^{Id} \\ = \; & ((\lambda x \rightarrow t)\;u) \\ = \; & (t \text{ where } x = u). \end{aligned}$$

Similarly, a sequence of qualifiers corresponds to a sequence of nested "where" terms:

$$\begin{aligned} & [\,t \mid x \leftarrow u; y \leftarrow v\,]^{Id} \\ = \; & ((t \text{ where } y = v) \text{ where } x = u). \end{aligned}$$

Since y is bound after x it appears in the inner "where" term. In the following, comprehensions in the identity monad will be written in preference to "where" terms, as the two are equivalent.

(In the Hindley-Milner type system, λ-terms and "where" terms differ in that the latter may introduce polymorphism. The key factor allowing "where" terms to play this role is that the syntax pairs each bound variable with its binding term. Since monad comprehensions have a similar property, it seems reasonable that they, too, could be used to introduce polymorphism. However, the following does not require comprehensions that introduce polymorphism, so we leave exploration of this issue for the future.)

3.2 The strictness monad

Sometimes it is necessary to control order of evaluation in a lazy functional program. This is usually achieved with the computable function *strict*, defined by

$$\mathit{strict}\,f\,x \quad = \quad \text{if } x \neq \bot \text{ then } f\,x \text{ else } \bot.$$

Operationally, *strict f x* is reduced by first reducing x to weak head normal form (WHNF) and then reducing the application $f\,x$. Alternatively, it is safe to reduce x and $f\,x$ in parallel, but not allow access to the result until x is in WHNF.

We can use this function as the basis of a monad:

$$\begin{array}{lcl} \text{type } \mathit{Str}\,x & = & x \\ \mathit{map}^{\mathit{Str}}\,f\,x & = & \mathit{strict}\,f\,x \\ \mathit{unit}^{\mathit{Str}}\,x & = & x \\ \mathit{join}^{\mathit{Str}}\,x & = & x. \end{array}$$

This is identical to the identity monad, except for the definition of $\mathit{map}^{\mathit{Str}}$. It is easy to verify that this does indeed satisfy the monad laws, (*i*)–(*iv*) and (*I*)–(*III*).

The corresponding monad comprehension provides a simple way to control the evaluation order of lazy programs, which we will make use of later. For instance, the operational interpretation of

$$[\,t \mid x \leftarrow u;\; y \leftarrow v\,]^{\mathit{Str}}$$

is as follows: reduce u to WHNF, bind x to the value of u, reduce v to WHNF, bind y to value of v, then reduce t. Alternatively, it is safe to reduce t, u, and v in parallel, but not allow access to the result until both u and v are in WHNF.

4 Manipulating state

Procedural programming languages operate by assigning to a state; this is also possible in impure functional languages such as Standard ML. In pure functional languages, assignment may be simulated by passing around a value representing the current state. This section shows how the monad of state transformers and the corresponding comprehension can be used to structure programs written in this style.

4.1 State transformers

Fix a type S of states. The monad of state transformers ST is defined by

$$\begin{array}{lcl} \text{type } ST\,x & = & S \rightarrow (x, S) \\ \mathit{map}^{ST}\,f\,\bar{x} & = & \lambda s \rightarrow [\,(f\,x, s') \mid (x, s') \leftarrow \bar{x}\,s\,]^{Id} \\ \mathit{unit}^{ST}\,x & = & \lambda s \rightarrow (x, s) \\ \mathit{join}^{ST}\,\bar{\bar{x}} & = & \lambda s \rightarrow [\,(x, s'') \mid (\bar{x}, s') \leftarrow \bar{\bar{x}}\,s; \\ & & \qquad\qquad (x, s'') \leftarrow \bar{x}\,s'\,]^{Id}. \end{array}$$

(Recall the equivalence of *Id*-comprehensions and "where" terms as explained in Section 3.1.) A state transformer of type x takes a state and returns a value (of type x) and a (new) state. The unit takes the value x into the state transformer $\lambda s \rightarrow (x, s)$ that returns x and leaves the state unchanged. It follows from these definitions that

$$\begin{array}{l} [\,(x, y) \mid x \leftarrow \bar{x};\; y \leftarrow \bar{y}\,]^{ST} \\ \quad = \lambda s \rightarrow [\,((x, y), s'') \mid (x, s') \leftarrow \bar{x}\,s; \\ \qquad\qquad\qquad\qquad (y, s'') \leftarrow \bar{y}\,s'\,]^{Id}. \end{array}$$

This applies the state transformer $\bar{x}$ to the state s, yielding the value x and the new state s'; it then applies a second transformer $\bar{y}$ to the state s' yielding the value y and the newer state s''; it finally returns a value consisting of x paired with y and the final state s''.

Two useful operations in this monad are

$$\begin{array}{lcl} \mathit{fetch} & :: & ST\,S \\ \mathit{fetch} & = & \lambda s \rightarrow (s, s) \\ \\ \mathit{assign} & :: & S \rightarrow ST\,() \\ \mathit{assign}\,s' & = & \lambda s \rightarrow ((), s'). \end{array}$$

The first of these fetches the current value of the state (leaving the state unchanged); the second discards the old state, assigning the new state to be the given value. Here () is the type that contains only the value ().

A third useful operation is

$$\begin{array}{lcl} \mathit{init} & :: & S \rightarrow ST\,x \rightarrow x \\ \mathit{init}\,s\,\bar{x} & = & [\,x \mid (x, s') \leftarrow \bar{x}\,s\,]^{Id}. \end{array}$$

This applies the state transformer $\bar{x}$ to a given initial state s; it returns the value computed by the state transformer while discarding the final state.

4.2 Example: Renaming

Say we wish to rename all bound variables in a lambda term. A suitable data type *Term* for representing lambda terms is defined in Figure 1 (in Standard ML) and Figure 2 (in Haskell). New names are to be generated by counting; we assume there is a function

$$mkname :: Int \rightarrow Name$$

that given an integer computes a name. We also assume a function

$$subst :: Name \rightarrow Name \rightarrow Term \rightarrow Term$$

such that $subst\ x'\ x\ t$ substitutes x' for each occurrence of x in t.

A solution to this problem in an impure functional language is shown in Figure 1. This uses a reference N to an assignable storage location containing an integer, the current state. The "functions" and their types are:

$$\begin{array}{lll} newname & :: & () \rightarrow Name, \\ renamer & :: & Term \rightarrow Term, \\ rename & :: & Term \rightarrow Term. \end{array}$$

Note that *newname* and *renamer* are not true functions as they depend on the state. In particular, *newname* () returns a different name each time it is called, and hence requires a dummy parameter, (). However, *rename* is a true function (it always generates new names starting from 0). Understanding the program requires a knowledge of which "functions" affect the state and which do not. This is not always easy to see – *renamer* is not a true function, even though it does not contain any direct reference to the state N, because it does contain an indirect reference through *newname*; but *rename* is a true function, even though it references *renamer*.

An equivalent solution in a pure functional language is shown in Figure 2. This explicitly passes around an integer that is used to generate new names. The functions and their types are:

$$\begin{array}{lll} newname & :: & Int \rightarrow (Name, Int), \\ renamer & :: & Term \rightarrow Int \rightarrow (Term, Int), \\ rename & :: & Term \rightarrow Term. \end{array}$$

The function *newname* generates a new name from the integer and returns an incremented integer; the function *renamer* takes a term and an integer and returns a renamed term (with names generated from the given integer) paired with the final integer generated. The function *rename* takes a term and returns a renamed term (with names generated from 0). This program is straightforward, but can be difficult to read because it contains a great deal of "plumbing" to pass around the state. It is relatively easy to introduce errors into such programs, by writing n where n' is intended or the like. This "plumbing problem" can be more severe in a program of greater complexity.

Finally, a solution of this problem using the monad of state transformers is shown in Figure 3. The state is taken as $S = Int$. The functions and their types are now:

$$\begin{array}{lll} newname & :: & ST\ Name, \\ renamer & :: & Term \rightarrow ST\ Name, \\ rename & :: & Term \rightarrow Term. \end{array}$$

The monadic program is simply a different way of writing the pure program. Types in the monadic program can be seen to correspond directly to the types in the impure program: an impure "function" of type $x \rightarrow y$ that affects the state corresponds to a pure function of type $x \rightarrow ST\,y$. Thus, *renamer* has type $Term \rightarrow Term$ in the impure program, and type $Term \rightarrow ST\ Term$ in the monadic program; and *newname* has type $() \rightarrow Name$ in the impure program, and type $ST\ Name$, which is isomorphic to $() \rightarrow ST\ Name$, in the pure program. Unlike the impure program, types in the monadic program make it manifest where the state is affected (and so do the ST-comprehensions).

The "plumbing" is now handled implicitly by the state transformer rather than explicitly. Various kinds of errors that are possible in the pure program (such as accidentally writing n in place of n') are impossible in the monadic program. Further, the type system ensures that plumbing is handled in an appropriate way. For example, one might be tempted to write, say, $App\,(renamer\,t)\,(renamer\,u)$ for the right-hand side of the last equation defining *renamer*, but this would be detected as a type error.

Safety can be further ensured by making ST into an abstract data type on which map^{ST}, $unit^{ST}$, $join^{ST}$, *fetch*, *assign*, and *init* are the only operations. This guarantees that one cannot mix the state transformer abstraction with other functions which handle the state inappropriately. This idea will be pursued in the next section.

Impure functional languages (such as Standard ML) are restricted to using a strict (or call-by-value) order of evaluation, because otherwise the effect of the assignments becomes very difficult to predict. Programs using the monad of state transformers can be written in languages using either a strict (call-by-value) or lazy (call-by-name) order of evaluation. The state-transformer comprehensions make clear exactly the order in which the assignments take effect, regardless of the order of evaluation used.

Reasoning about programs in impure functional languages is problematic (although not impossible – see [MT89] for one approach). In contrast, programs written using monads, like all pure programs, can be reasoned about in the usual way, substituting equals for equals. They also satisfy additional laws, such as the

```
datatype Term = Var of Name | Lam of Name * Term | App of Term * Term;
fun rename t = let
                   val N = ref 0;
                   fun newname () = let n = !N; () = (N := n + 1) in mkname n;
                   fun renamer (Var x)       = Var x
                   |   renamer (Lam (x, t)) = let x' = newname () in
                                                  Lam (x', subst x' x (renamer t))
                   |   renamer (App (t, u)) = App (renamer t, renamer u)
               in
                 renamer t;
```

Figure 1: Renaming in impure functional language (Standard ML)

```
data Term            =  Var Name | Lam Name Term | App Term Term

newname              ::  Int → (Name, Int)
newname n            =   (mkname n, n + 1)

renamer              ::  Term → Int → (Term, Int)
renamer (Var x) n    =   (Var x, n)
renamer (Lam x t) n  =   (Lam x' (subst x' x t'), n'') where
                            (x', n') = newname n
                            (t', n'') = renamer t n'
renamer (App t u) n  =   (App t' u', n'') where
                            (t', n') = renamer t n
                            (u', n'') = renamer u n'

rename               ::  Term → Term
rename t             =   t' where (t', n') = renamer t 0
```

Figure 2: Renaming in pure functional language (Haskell)

```
data Term          =  Var Name | Lam Name Term | App Term Term

newname            ::  ST Name
newname            =   [mkname n | n ← fetch; () ← assign (n + 1)]^ST

renamer            ::  Term → ST Term
renamer (Var x)    =   [Var x]^ST
renamer (Lam x t)  =   [Lam x' (subst x' x t')) | x' ← newname; t' ← renamer t]^ST
renamer (App t u)  =   [App t' u' | t' ← renamer t; u' ← renamer u]^ST

rename             ::  Term → Term
rename t           =   init 0 (renamer t)
```

Figure 3: Renaming with the monad of state transformers

following laws on qualifiers:

$$\begin{aligned} x \leftarrow fetch;\ y \leftarrow fetch &= x \leftarrow fetch;\ y \leftarrow [x]^{ST}, \\ () \leftarrow assign\ x;\ y \leftarrow fetch &= () \leftarrow assign\ x;\ y \leftarrow [x]^{ST}, \\ () \leftarrow assign\ x;\ () \leftarrow assign\ y &= () \leftarrow assign\ y, \end{aligned}$$

and on terms:

$$\begin{aligned} init\, x\, [\, t\,]^{ST} &= t, \\ init\, x\, [\, t \mid q\,]^{ST} &= init\, x\, [\, t \mid () \leftarrow assign\ x;\ q\,]^{ST}, \\ init\, x\, [\, t \mid q;\ () \leftarrow assign\ y\,] &= init\, x\, [\, t \mid q\,]. \end{aligned}$$

These, combined with the monad laws (1)–(6), allow one to use equational reasoning to prove properties of programs that manipulate state.

4.3 Array update

Let *Arr* be the type of arrays taking indexes of type *Ix* and yielding values of type *Val*. The key operations on this type are

$$\begin{aligned} newarray &:: Val \to Arr, \\ index &:: Ix \to Arr \to Val, \\ update &:: Ix \to Val \to Arr \to Arr. \end{aligned}$$

Here *newarray v* returns an array with all entries set to v; and *index i a* returns the value at index i in array a; and *update i v a* returns an array where index i has value v and the remainder is identical to a. In equations,

$$\begin{aligned} index\ i\ (newarray\ v) &= v, \\ index\ i\ (update\ i\ v\ a) &= v, \\ index\ i\ (update\ i'\ v\ a) &= index\ i\ a, \quad \text{if } i \neq i'. \end{aligned}$$

The efficient way to implement the update operation is to overwrite the specified entry of the array, but in a pure functional language this is only safe if there are no other pointers to the array extant when the update operation is performed.

Now consider the monad of state transformers taking the state type $S = Arr$, so that

$$\text{type } ST\, x = Arr \to (x, Arr).$$

Variants of the *fetch* and *assign* operations can be defined to act on an array entry specified by a given index, and a variant of *init* can be defined to initialise all entries in an array to a given value:

$$\begin{aligned} fetch &:: Ix \to ST\ Val \\ fetch\ i &= \lambda a \to [(v, a) \mid v \leftarrow index\ i\ a]^{Str} \\[1ex] assign &:: Ix \to Val \to ST\, () \\ assign\ i\ v &= \lambda a \to ((), update\ i\ v\ a) \\[1ex] init &:: Val \to ST\, x \to x \\ init\ v\, \overline{x} &= [x \mid (x, a) \leftarrow \overline{x}\, (newarray\ v)]^{Id}. \end{aligned}$$

A *Str*-comprehension is used in *fetch* to force the entry from a to be fetched before a is made available for further access; this is essential in order for it to be safe to update a by overwriting.

Now, say we make ST into an abstract data type such that the only operations on values of type ST are map^{ST}, $unit^{ST}$, $join^{ST}$, *fetch*, *assign*, and *init*. It is straightforward to show that each of these operations, when passed the sole pointer to an array, returns as its second component the sole pointer to an array. Since these are the only operations that may be used to build a term of type ST, this guarantees that it is safe to implement the *assign* operation by overwriting the specified array entry.

The key idea here is the use of the abstract data type. Monad comprehensions are not essential for this to work, they merely provide a desirable syntax.

4.4 Example: Interpreter

Consider building an interpreter for a simple imperative language. The store of this imperative language will be modelled by a state of type *Arr*, so we will take *Ix* to be the type of variable names, and *Val* to be the type of values stored in variables. The abstract syntax for this language is represented by the data types shown in Figure 4. The language consists of expressions, commands, and programs:

$$\begin{aligned} \text{data } Exp &= Var\ Ix \mid Const\ Val \mid Plus\ Exp\ Exp \\ \text{data } Com &= Asgn\ Ix\ Exp \mid Seq\ Com\ Com \\ &\quad \mid If\ Exp\ Com\ Com \\ \text{data } Prog &= Prog\ Com\ Exp. \end{aligned}$$

An expression is a variable, a constant, or the sum of two expressions; a command is an assignment, a sequence of two commands, or a conditional; and a program consists of a command followed by an expression.

A version of the interpreter in a pure functional language is shown in Figure 4. The interpreter can be read as a denotational semantics for the language, with three semantic functions:

$$\begin{aligned} exp &:: Exp \to Arr \to Val, \\ com &:: Com \to Arr \to Arr, \\ prog &:: Prog \to Val. \end{aligned}$$

The semantics of an expression takes a store into a value; the semantics of a command takes a store into a store; and the semantics of a program is a value. A program consists of a command followed by an expression; its value is determined by applying the command to an initial store where all variables have the value 0, and then evaluating the expression in the context of the resulting store.

$$
\begin{aligned}
&\mathit{exp} && :: && \mathit{Exp} \rightarrow \mathit{Arr} \rightarrow \mathit{Val}\\
&\mathit{exp}\,(\mathit{Var}\;i)\;a && = && \mathit{lookup}\;i\;a\\
&\mathit{exp}\,(\mathit{Const}\;v)\;a && = && v\\
&\mathit{exp}\,(\mathit{Plus}\;e_1\;e_2)\;a && = && \mathit{exp}\;e_1\;a + \mathit{exp}\;e_2\;a\\
\\
&\mathit{com} && :: && \mathit{Com} \rightarrow \mathit{Arr} \rightarrow \mathit{Arr}\\
&\mathit{com}\,(\mathit{Asgn}\;i\;e)\;a && = && \mathit{update}\;i\,(\mathit{exp}\;e\;a)\;a\\
&\mathit{com}\,(\mathit{Seq}\;c_1\;c_2)\;a && = && \mathit{com}\;c_2\,(\mathit{com}\;c_1\;a)\\
&\mathit{com}\,(\mathit{If}\;e\;c_1\;c_2)\;a && = && \text{if } \mathit{exp}\;e\;a = 0 \text{ then } \mathit{com}\;c_1\;a \text{ else } \mathit{com}\;c_2\;a\\
\\
&\mathit{prog} && :: && \mathit{Prog} \rightarrow \mathit{Val}\\
&\mathit{prog}\,(\mathit{Prog}\;c\;e) && = && \mathit{exp}\;e\,(\mathit{com}\;c\,(\mathit{newarray}\;0))
\end{aligned}
$$

Figure 4: Interpreter in a pure functional language

$$
\begin{aligned}
&\mathit{exp} && :: && \mathit{Exp} \rightarrow \mathit{ST}\;\mathit{Val}\\
&\mathit{exp}\,(\mathit{Var}\;i) && = && [\,v \mid v \leftarrow \mathit{fetch}\;i\,]^{ST}\\
&\mathit{exp}\,(\mathit{Const}\;v) && = && [\,v\,]^{ST}\\
&\mathit{exp}\,(\mathit{Plus}\;e_1\;e_2) && = && [\,v_1 + v_2 \mid v_1 \leftarrow \mathit{exp}\;e_1;\; v_2 \leftarrow \mathit{exp}\;e_2\,]^{ST}\\
\\
&\mathit{com} && :: && \mathit{Com} \rightarrow \mathit{ST}\,()\\
&\mathit{com}\,(\mathit{Asgn}\;i\;e) && = && [\,() \mid v \leftarrow \mathit{exp}\;e;\; () \leftarrow \mathit{assign}\;i\;v\,]^{ST}\\
&\mathit{com}\,(\mathit{Seq}\;c_1\;c_2) && = && [\,() \mid () \leftarrow \mathit{com}\;c_1;\; () \leftarrow \mathit{com}\;c_2\,]^{ST}\\
&\mathit{com}\,(\mathit{If}\;e\;c_1\;c_2) && = && [\,() \mid v \leftarrow \mathit{exp}\;e;\; () \leftarrow \text{if } v = 0 \text{ then } \mathit{com}\;c_1 \text{ else } \mathit{com}\;c_2\,]^{ST}\\
\\
&\mathit{prog} && :: && \mathit{Prog} \rightarrow \mathit{Val}\\
&\mathit{prog}\,(\mathit{Prog}\;c\;e) && = && \mathit{init}\,0\,[\,v \mid () \leftarrow \mathit{com}\;c;\; v \leftarrow \mathit{exp}\;e\,]^{ST}
\end{aligned}
$$

Figure 5: Interpreter with state transformers

$$
\begin{aligned}
&\mathit{exp} && :: && \mathit{Exp} \rightarrow \mathit{SR}\;\mathit{Val}\\
&\mathit{exp}\,(\mathit{Var}\;i) && = && [\,v \mid v \leftarrow \mathit{fetch}\;i\,]^{SR}\\
&\mathit{exp}\,(\mathit{Const}\;v) && = && [\,v\,]^{ST}\\
&\mathit{exp}\,(\mathit{Plus}\;e_1\;e_2) && = && [\,v_1 + v_2 \mid v_1 \leftarrow \mathit{exp}\;e_1;\; v_2 \leftarrow \mathit{exp}\;e_2\,]^{SR}\\
\\
&\mathit{com} && :: && \mathit{Com} \rightarrow \mathit{ST}\,()\\
&\mathit{com}\,(\mathit{Asgn}\;i\;e) && = && [\,() \mid v \leftarrow \mathit{ro}\,(\mathit{exp}\;e);\; () \leftarrow \mathit{assign}\;i\;v\,]^{ST}\\
&\mathit{com}\,(\mathit{Seq}\;c_1\;c_2) && = && [\,() \mid () \leftarrow \mathit{com}\;c_1;\; () \leftarrow \mathit{com}\;c_2\,]^{ST}\\
&\mathit{com}\,(\mathit{If}\;e\;c_1\;c_2) && = && [\,() \mid v \leftarrow \mathit{ro}\,(\mathit{exp}\;e);\; () \leftarrow \text{if } v = 0 \text{ then } \mathit{com}\;c_1 \text{ else } \mathit{com}\;c_2\,]^{ST}\\
\\
&\mathit{prog} && :: && \mathit{Prog} \rightarrow \mathit{Val}\\
&\mathit{prog}\,(\mathit{Prog}\;c\;e) && = && \mathit{init}\,0\,[\,v \mid () \leftarrow \mathit{com}\;c;\; v \leftarrow \mathit{ro}\,(\mathit{exp}\;e)\,]^{ST}
\end{aligned}
$$

Figure 6: Interpreter with state transformers and readers

The interpreter uses the array operations *newarray*, *index*, and *update*. As it happens, it is safe to perform the updates in place for this program, but to discover this requires using one of the special analysis techniques cited in the introduction.

The same interpreter has been rewritten in Figure 5 using state transformers. The semantic functions now have the types:

$$\begin{array}{lcl} exp & :: & Exp \rightarrow ST\ Val, \\ com & :: & Com \rightarrow ST\ (), \\ prog & :: & Prog \rightarrow Val. \end{array}$$

The semantics of an expression depends on the state and returns a value; the semantics of a command transforms the state only; the semantics of a program, as before, is just a value. According to the types, the semantics of an expression might alter the state, although in fact expressions depend the state but do not change it - we will return to this problem shortly.

The abstract data type for ST guarantees that it is safe to perform updates (indicated by *assign*) in place - no special analysis technique is required. It is easy to see how the monad interpreter can be derived from the original, and (using the definitions given earlier) the proof of their equivalence is straightforward.

The program written using state transformers has a simple imperative reading. For instance, the line

$$com\,(Seq\; c_1\; c_2) \;=\; [\,()\mid () \leftarrow com\; c_1;\; () \leftarrow com\; c_2\,]^{ST}$$

can be read "to evaluate the command $Seq\; c_1\; c_2$, first evaluate c_1 and then evaluate c_2". The types and the use of the ST comprehension make clear that these operations transform the state; further, that the values returned are of type () makes it clear that only the effect on the state is of interest here.

One drawback of this program is that it introduces too much sequencing. The line

$$\begin{array}{l} exp\,(Plus\; e_1\; e_2) \\ \quad = \;[\,v_1 + v_2 \mid v_1 \leftarrow exp\; e_1;\; v_2 \leftarrow exp\; e_2\,]^{ST} \end{array}$$

can be read "to evaluate $Plus\; e_1\; e_2$, first evaluate e_1 yielding the value v_1, then evaluate e_2 yielding the value v_2, then add v_1 and v_2". This is unfortunate: it imposes a spurious ordering on the evaluation of e_1 and e_2 (the original program implies no such ordering). The order does not matter because although *exp* depends on the state, it does not change it. But, as already noted, there is no way to express this using just the monad of state transformers. To remedy this we will introduce a second monad, that of state readers.

4.5 State readers

Recall that the monad of state transformers, for a fixed type S of states, is given by

$$\text{type}\; ST\; x \;=\; S \rightarrow (x, S)$$

The monad of state readers, for the same type S of states, is given by

$$\begin{array}{lcl} \text{type}\; SR\; x & = & S \rightarrow (x, S) \\ map^{SR} f\; \hat{x} & = & \lambda s \rightarrow [\,f\; x \mid x \leftarrow \hat{x}\; s\,]^{Id} \\ unit^{SR}\; x & = & \lambda s \rightarrow x \\ join^{SR}\; \hat{\hat{x}} & = & \lambda s \rightarrow [\,x \mid \hat{x} \leftarrow \hat{\hat{x}}\; s;\; x \leftarrow \hat{x}\; s\,]^{Id}. \end{array}$$

Here $\hat{x}$ is a variable of type $SR\; x$, just as $\bar{x}$ is a variable of type $ST\; x$. A state reader of type x takes a state and returns a value (of type x), but no new state. The unit takes the value x into the state transformer $\lambda s \rightarrow x$ that ignores the state and returns x. It follows from these definitions that

$$\begin{array}{l} [\,(x,y) \mid x \leftarrow \hat{x};\; y \leftarrow \hat{y}\,]^{SR} \\ \quad = \lambda s \rightarrow [\,(x,y) \mid x \leftarrow \hat{x}\; s;\; y \leftarrow \hat{y}\; s\,]^{Id} \end{array}$$

This applies the state readers $\hat{x}$ and $\hat{y}$ to the state s, yielding the values x and y, which are returned in a pair.

It is easy to see for this monad that

$$\begin{array}{l} [\,(x,y) \mid x \leftarrow \hat{x};\; y \leftarrow \hat{y}\,]^{SR} \\ \quad = \;[\,(x,y) \mid y \leftarrow \hat{y};\; x \leftarrow \hat{x}\,]^{SR}, \end{array}$$

so that the order in which $\hat{x}$ and $\hat{y}$ are computed is irrelevant. A monad with this property is called *commutative*, since it follows from this that

$$[\,t \mid p;\; q\,]^{SR} \;=\; [\,t \mid q;\; p\,]^{SR}$$

for any term t, and any qualifiers p and q such that p binds no free variables of q and vice-versa. Thus state readers capture the notion of order independence that we desire for expression evaluation in the interpreter example.

Two useful operations in this monad are

$$\begin{array}{lcl} fetch & :: & SR\; S \\ fetch & = & \lambda s \rightarrow s \\ & & \\ ro & :: & SR\; x \rightarrow ST\; x \\ ro\; \hat{x} & = & \lambda s \rightarrow [\,(x,s) \mid x \leftarrow \hat{x}\; s\,]^{Id}. \end{array}$$

The first is the equivalent of the previous *fetch*, but now expressed as a state reader rather than a state transformer. The second converts a state reader into the corresponding state transformer: one that returns the same value as the state reader, and leaves the state unchanged. (The name *ro* abbreviates "read only".)

In the specific case where S is the array type Arr, we define

$$\begin{array}{lcl} fetch & :: & Ix \to SR\ Val \\ fetch\ i & = & \lambda a \to index\ i\ a. \end{array}$$

In order to guarantee the safety of update by overwriting, it is necessary to modify two of the other definitions to use Str-comprehensions rather than Id-comprehensions:

$$\begin{array}{lcl} map^{SR} f\ \hat{x} & = & \lambda a \to [\, f\ x \mid x \leftarrow \hat{x}\ a\,]^{Str} \\ ro\ \hat{x} & = & \lambda a \to [\,(x, a) \mid x \leftarrow \hat{x}\ a\,]^{Str} \end{array}$$

These correspond to the use of an Str-comprehension in the ST version of *fetch*.

Thus, for arrays, the complete collection of operations on state transformers and state readers consists of

$$\begin{array}{lcl} fetch & :: & Ix \to SR\ Val, \\ assign & :: & Ix \to Val \to ST\,(), \\ ro & :: & SR\,x \to ST\,x, \\ init & :: & Val \to ST\,x \to x, \end{array}$$

together with map^{SR}, $unit^{SR}$, $join^{SR}$ and map^{ST}, $unit^{ST}$, $join^{ST}$. These ten operations should be defined together and constitute all the ways of manipulating the two mutually defined abstract data types $SR\,x$ and $ST\,x$. It is straightforward to show that each operation of type SR, when passed an array, returns a value that contains no pointer to that array once it has been reduced to weak head normal form (WHNF); and that each operations of type ST, when passed the sole pointer to an array, returns as its second component the sole pointer to an array. Since these are the only operations that may be used to build values of types SR and ST, this guarantees that it is safe to implement the *assign* operation by overwriting the specified array entry. (The reader may check that the Str-comprehensions in map^{SR} and ro are essential to guarantee this property.)

Returning to the interpreter example, we get the new version shown in Figure 6. The only difference from the previous version is that some occurrences of ST have changed to SR and that ro has been inserted in a few places. The new typing

$$exp :: Exp \to SR\ Val$$

makes it clear that exp depends on the state but does not alter it. A proof that the programs in Figures 5 and 6 are equivalent is given in Section 6.

The excessive sequencing of the previous version has been eliminated. The line

$$\begin{array}{l} exp\,(Plus\ e_1\ e_2) \\ \quad = [\, v_1 + v_2 \mid v_1 \leftarrow exp\ e_1;\ v_2 \leftarrow exp\ e_2\,]^{SR} \end{array}$$

can now be read "to evaluate $Plus\ e_1\ e_2$, evaluate e_1 yielding the value v_1 and evaluate e_2 yielding the value v_2, then add v_1 and v_2". The order of qualifiers in an SR-comprehension is irrelevant, and so it is perfectly permissible to evaluate e_1 and e_2 in any order, or even concurrently.

The interpreter derived here is similar in structure to one in [Wad90], which uses a type system based on linear logic to guarantee safe destructive update of arrays. (A similar type system is discussed in [GH90].) However, the linear type system uses a "let!" construct that suffers from some unnatural restrictions: it requires hyperstrict evaluation, and it prohibits certain types involving functions. By contrast, the monad approach requires only strict evaluation, and it places no restriction on the types. This suggests that a careful study of the monad approach may lead to an improved understanding of linear types and the "let!" construct.

5 Filters

So far, we have ignored another form of qualifier found in list comprehensions, the *filter*. For list comprehensions, we can define filters by

$$[\, t \mid b\,] \;=\; \text{if } b \text{ then } [t] \text{ else } [\,],$$

where b is a boolean-valued term. For example,

$$\begin{array}{cl} & [\, x \mid x \leftarrow [1,2,3];\ odd\ x\,] \\ = & join\,[\,[\, x \mid odd\ x\,] \mid x \leftarrow [1,2,3]\,] \\ = & join\,[\,[\,1 \mid odd\ 1], [\,2 \mid odd\ 2], [\,3 \mid odd\ 3\,]\,] \\ = & join\,[\,[1], [\,], [2]\,] \\ = & [1,3]. \end{array}$$

Can we define filters in general for comprehensions in an arbitrary monad M? The answer is yes, if we can define $[\,]$ for M. Not all monads admit a useful definition of $[\,]$, but many do.

Recall that M-comprehensions of the form $[t]$ are defined in terms of the qualifier Λ, by taking $[\,t\,] = [\,t \mid \Lambda\,]$, and that Λ is a unit for qualifier composition,

$$[\, t \mid \Lambda;\ q\,] \;=\; [\, t \mid q\,] \;=\; [\, t \mid q;\ \Lambda\,].$$

Similarly, we will define M-comprehensions of the form $[\,]$ in terms of a new qualifier, $\emptyset$, by taking $[\,] = [\, t \mid \emptyset\,]$, and we will require that $\emptyset$ is a zero for qualifier composition,

$$[\, t \mid \emptyset;\ q\,] \;=\; [\, t \mid \emptyset\,] \;=\; [\, t \mid q;\ \emptyset\,].$$

Unlike with $[t|\Lambda]$, the value of $[t|\emptyset]$ is independent of t!

To define Λ we introduced a function $unit :: x \to M\ x$ satisfying the laws

$$\begin{array}{lrcl} (iii) & map\,f \cdot unit & = & unit, \\ (I) & join \cdot unit & = & id, \\ (II) & join \cdot map\ unit & = & id, \end{array}$$

and then taking $[\, t \mid \Lambda \,] = unit\, t$. Similarly, to define $\emptyset$ we introduce a function

$$zero :: y \rightarrow M\, x,$$

satisfying the laws

$$\begin{array}{lrcl} (v) & map\, f \cdot zero & = & zero \cdot g, \\ (IV) & join \cdot zero & = & zero, \\ (V) & join \cdot map\, zero & = & zero. \end{array}$$

Law (v) specifies that the result of *zero* is independent of its argument; it follows immediately from the type of *zero* (again, see [Rey83, Wad89] for the reason). In the case of lists, setting $zero\, y = [\,]$ makes the last two laws hold, because $join\, [\,] = [\,]$ and $join\, [[\,], \ldots, [\,]] = [\,]$. (This ignores what happens when *zero* is applied to $\bot$, which will be considered below.)

In general, if any monad $(map, unit, join)$ also possesses a *zero* satisfying the three laws above, then we can define $[\,]$ in this monad by taking $[\,] = zero\, t$ for any t. Moreover, we can extend comprehensions in this monad to contain a new form of qualifier, the filter, defined by

$$(7) \qquad [\, t \mid b \,] \;=\; \text{if } b \text{ then } unit\, t \text{ else } zero\, t,$$

where b is any boolean-valued term. For this extended definition of qualifiers, we can show that laws (4) and (6) still hold. We also have the new laws

$$\begin{array}{lrcl} (8) & [\, t \mid b;\, c \,] & = & [\, t \mid (b \wedge c) \,], \\ (9) & [\, t \mid q;\, b \,] & = & [\, t \mid b;\, q \,], \end{array}$$

where b and c are boolean-valued terms, and where q is any qualifier not binding variables used in b.

When dealing with $\bot$ as a potential value, more care is required. In a strict language, where all functions (including *zero*) are strict, there is no problem. But in a lazy language, in the case of lists, laws (v) and (IV) hold, but law (V) is an inequality, $join \cdot map\, zero \sqsubseteq zero$, since $join\, (map\, zero)\, \bot = \bot$ but $zero\, \bot = [\,]$. In this case, laws (1)–(8) are still valid, but law (9) holds only if $[\, t \mid q \,] \neq \bot$. In the case that $[\, t \mid q \,] = \bot$, law (9) becomes an inequality, $[\, t \mid q;\, b \,] \sqsubseteq [\, t \mid b;\, q \,]$.

As a second example of a monad with a zero, consider the strictness monad *Str* defined in Section 3.2. For this monad, a zero may be defined by $zero^{Str}\, y = \bot$. It is easy to verify that the required laws hold; unlike with lists, the laws hold even when *zero* is applied to $\bot$. For example, $[\, x - 1 \mid x \geq 1 \,]^{Str}$ returns one less than x if x is positive, and $\bot$ otherwise.

6 Monad morphisms

If M and N are two monads, then $h :: M\, x \rightarrow N\, x$ is a *monad morphism* from M to N if it preserves the monad operations:

$$\begin{array}{rcl} h \cdot map^M f & = & map^N f \cdot h, \\ h \cdot unit^M & = & unit^N, \\ h \cdot join^M & = & join^N \cdot h^2, \end{array}$$

where $h^2 = h \cdot map^M\, h = map^N\, h \cdot h$ (the two composites are equal by the first equation).

Define the effect of a monad morphism on qualifiers as follows:

$$\begin{array}{lcl} h\, (\Lambda) & = & \Lambda, \\ h\, (x \leftarrow u) & = & x \leftarrow (h\, u), \\ h\, (p;\, q) & = & (h\, p);\, (h\, q). \end{array}$$

It follows that if h is a monad morphism from M to N then

$$(*) \qquad h\, [\, t \mid q \,]^M \;=\; [\, t \mid (h\, q) \,]^N$$

for all terms t and qualifiers q. The proof is a simple induction on the form of qualifiers.

As an example, it is easy to check that $unit^M :: x \rightarrow M\, x$ is a monad morphism from *Id* to M. It follows that

$$[[\, t \mid x \leftarrow u \,]^{Id}]^M \;=\; [\, t \mid x \leftarrow [\, u \,]^M \,]^M.$$

This explains a trick occasionally used by functional programmers, where one writes the qualifier $x \leftarrow [\, u \,]$ inside a list comprehension to bind x to the value of u, that is, to achieve the same effect as the qualifier $x \leftarrow u$ in an *Id* comprehension.

As a second example, the function *ro* from Section 4.5 is a monad morphism from *SR* to *ST*. This can be used to prove the equivalence of the two interpreters in Figures 5 and 6. Write $exp^{ST} :: Exp \rightarrow ST\ Val$ and $exp^{SR} :: Exp \rightarrow SR\ Val$ for the versions in the two figures. The equivalence of the two versions is clear if we can show that

$$ro \cdot exp^{SR} \;=\; exp^{ST}.$$

The proof is a simple induction on the structure of expressions. In the case that the expression has the form $(Plus\ e_1\ e_2)$, we have that

$$\begin{array}{cl} & ro\, (exp^{SR}\, (Plus\ e_1\ e_2)) \\ = & \{\text{unfolding } exp^{SR}\} \\ & ro\, [\, v_1 + v_2 \mid v_1 \leftarrow exp^{SR}\, e_1;\, v_2 \leftarrow exp^{SR}\, e_2 \,]^{SR} \\ = & \{\text{by } (*)\} \\ & [\, v_1 + v_2 \mid v_1 \leftarrow ro\, (exp^{SR}\, e_1);\, v_2 \leftarrow ro\, (exp^{SR}\, e_2) \,]^{ST} \\ = & \{\text{hypothesis}\} \\ & [\, v_1 + v_2 \mid v_1 \leftarrow exp^{ST}\, e_1;\, v_2 \leftarrow exp^{ST}\, e_2 \,]^{ST} \\ = & \{\text{folding } exp^{ST}\} \\ & exp^{ST}\, (Plus\ e_1\ e_2). \end{array}$$

The other two cases are equally simple.

All of this extends straightforwardly to monads with zero. In this case we also require that $h \cdot zero^M = zero^N$, define the action of a morphism on a filter by $h\, b = b$, and observe that $(*)$ holds even when q contains filters.

7 More monads

This section describes three more monads: parsers, expressions, and continuations. The basic techniques are not new (parsers are discussed in [Wad85, Fai87, FL89], and exceptions are discussed in [Wad85, Spi89]), but monads and monad comprehensions provide a convenient framework for their expression.

7.1 Parsers

The monad of parsers is given by

$$\begin{aligned}
\text{type } \mathit{Parse}\, x &= \mathit{String} \to \mathit{List}\,(x, \mathit{String}) \\
\mathit{map}^{\mathit{Parse}}\, f\, \bar{x} &= \lambda i \to [(f\, x, i') \mid (x, i') \leftarrow \bar{x}\, i]^{\mathit{List}} \\
\mathit{unit}^{\mathit{Parse}}\, x &= \lambda i \to [(x, i)]^{\mathit{List}} \\
\mathit{join}^{\mathit{Parse}}\, \bar{\bar{x}} &= \lambda i \to [(x, i'') \mid (\bar{x}, i') \leftarrow \bar{\bar{x}}\, i; \\
&\qquad\qquad (x, i'') \leftarrow \bar{x}\, i']^{\mathit{List}}.
\end{aligned}$$

Here *String* is the type of lists of *Char*. Thus, a parser accepts an input string and returns a list of pairs. The list contains one pair for each succesful parse, consisting of the value parsed and the remaining unparsed input. An empty list denotes a failure to parse the input. It follows from these definitions that

$$\begin{aligned}
&[(x, y) \mid x \leftarrow \bar{x};\ y \leftarrow \bar{y}]^{\mathit{Parse}} \\
&= \lambda i \to [((x, y), i'') \mid (x, i') \leftarrow \bar{x}\, i; \\
&\qquad\qquad (y, i'') \leftarrow \bar{y}\, i']^{\mathit{List}}.
\end{aligned}$$

This applies the first parser to the input, binds x to the value parsed, then applies the second parser to the remaining input, binds y to the value parsed, then returns the pair (x, y) as the value together with input yet to be parsed. If either $\bar{x}$ or $\bar{y}$ fails to parse its input (returning an empty list) then the combined parser will fail as well.

There is also a suitable zero for this monad, given by

$$\mathit{zero}^{\mathit{Parse}}\, y \;=\; \lambda i \to []^{\mathit{List}}.$$

Thus, $[]^{\mathit{Parse}}$ is the parser that always fails to parse the input. It follows that we may use filters in *Parse*-comprehensions as well as in *List*-comprehensions.

The alternation operator combines two parsers:

$$\begin{aligned}
(\,[\!]\,) &:: \mathit{Parse}\, x \to \mathit{Parse}\, x \to \mathit{Parse}\, x \\
\bar{x} \,[\!]\, \bar{y} &= \lambda i \to (\bar{x}\, i) +\!\!+ (\bar{y}\, i).
\end{aligned}$$

(Here $+\!\!+$ is the operator that concatenates two lists.) It returns all parses found by the first argument followed by all parses found by the second.

The simplest parser is one that parses a single character:

$$\begin{aligned}
\mathit{next} &:: \mathit{Parse}\ \mathit{Char} \\
\mathit{next} &= \lambda i \to [(\mathit{head}\, i, \mathit{tail}\, i) \mid \mathit{not}\,(\mathit{null}\, i)]^{\mathit{List}}.
\end{aligned}$$

Here we have a *List*-comprehension with a filter. The parser *next* succeeds only if the input is non-empty, in which case it returns the next character. Using this, we may define a parser to recognise a literal:

$$\begin{aligned}
\mathit{lit} &:: \mathit{Char} \to \mathit{Parse}\,() \\
\mathit{lit}\, c &= [() \mid c' \leftarrow \mathit{next};\ c == c']^{\mathit{Parse}}.
\end{aligned}$$

Now we have a *Parse*-comprehension with a filter. The parser *lit c* succeeds only if the next character in the input is c.

As an example, a parser for fully parenthesised lambda terms, yielding values of the type *Term* described previously, can be written as follows:

$$\begin{aligned}
\mathit{term} &:: \mathit{Parse}\ \mathit{Term} \\
\mathit{term} &= [\,\mathit{Var}\, x \mid x \leftarrow \mathit{name}]^{\mathit{Parse}} \\
&\quad [\!]\ [\,\mathit{Lam}\, x\, t \mid () \leftarrow \mathit{lit}\,\text{`('};\ () \leftarrow \mathit{lit}\,\text{`}\lambda\text{'}; \\
&\qquad\qquad x \leftarrow \mathit{name};\ () \leftarrow \mathit{lit}\,\text{`}\to\text{'}; \\
&\qquad\qquad t \leftarrow \mathit{term};\ () \leftarrow \mathit{lit}\,\text{`)'}]^{\mathit{Parse}} \\
&\quad [\!]\ [\,\mathit{App}\, t\, u \mid () \leftarrow \mathit{lit}\,\text{`('};\ t \leftarrow \mathit{term}; \\
&\qquad\qquad u \leftarrow \mathit{term};\ () \leftarrow \mathit{lit}\,\text{`)'}]^{\mathit{Parse}}
\end{aligned}$$

$$\begin{aligned}
\mathit{name} &:: \mathit{Parse}\ \mathit{Name} \\
\mathit{name} &= [\,c \mid c \leftarrow \mathit{next};\ \text{`a'} \le c;\ c \le \text{`z'}]^{\mathit{Parse}}.
\end{aligned}$$

Here, for simplicity, it has been assumed that names consist of a single lower-case letter, so *Name* = *Char*; and that λ and $\to$ are both characters.

7.2 Exceptions

The type *Maybe x* consists of either a value of x, written *Just x*, or an exceptional value, written *Nothing*:

$$\text{data } \mathit{Maybe}\, x \;=\; \mathit{Just}\, x \mid \mathit{Nothing}.$$

(The names are due to Spivey [Spi89].) The following operations make this into a monad:

$$\begin{aligned}
\mathit{map}^{\mathit{Maybe}}\, f\,(\mathit{Just}\, x) &= \mathit{Just}\,(f\, x) \\
\mathit{map}^{\mathit{Maybe}}\, f\ \mathit{Nothing} &= \mathit{Nothing} \\[1ex]
\mathit{unit}^{\mathit{Maybe}}\, x &= \mathit{Just}\, x \\[1ex]
\mathit{join}^{\mathit{Maybe}}\,(\mathit{Just}\,(\mathit{Just}\, x)) &= \mathit{Just}\, x \\
\mathit{join}^{\mathit{Maybe}}\,(\mathit{Just}\ \mathit{Nothing}) &= \mathit{Nothing} \\
\mathit{join}^{\mathit{Maybe}}\ \mathit{Nothing} &= \mathit{Nothing}.
\end{aligned}$$

It follows from these definitions that

$$[(x, y) \mid x \leftarrow \bar{x};\ y \leftarrow \bar{y}]^{\mathit{Maybe}}$$

returns $\mathit{Just}\,(x, y)$ if $\bar{x}$ is *Just x* and $\bar{y}$ is *Just y*, and otherwise returns *Nothing*.

There is also a suitable zero for this monad, given by

$$\mathit{zero}^{\mathit{Maybe}}\, y \;=\; \mathit{Nothing}.$$

Hence $[\,]^{Maybe} = Nothing$ and $[x]^{Maybe} = Just\,x$. For example, $[x-1 \mid x \geq 1]^{Maybe}$ returns one less than x if x is positive, and *Nothing* otherwise.

Two useful operations test whether an argument corresponds to a value and, if so, return that value:

$$\begin{array}{lcl} exists & :: & Maybe\,x \rightarrow Bool \\ exists\,(Just\,x) & = & True \\ exists\,Nothing & = & False \end{array}$$

$$\begin{array}{lcl} the & :: & Maybe\,x \rightarrow x \\ the\,(Just\,x) & = & x. \end{array}$$

Observe that

$$[the\,\overline{x} \mid exists\,\overline{x}]^{Maybe} = \overline{x}$$

for all $\overline{x} :: Maybe\,x$. If we assume that $(the\ Nothing) = \perp$, it is easily checked that *the* is a monad morphism from *Maybe* to *Str*, and corresponds to the usual trick of considering error values and $\perp$ to be identical. We have that

$$the\,[x-1 \mid x \geq 1]^{Maybe} = [x-1 \mid x \geq 1]^{Str}$$

as an immediate consequence of the monad morphism law.

The biased-choice operator chooses the first of two possible values that is well defined:

$$\begin{array}{lcl} (?) & :: & Maybe\,x \rightarrow Maybe\,x \rightarrow Maybe\,x \\ \overline{x}\,?\,\overline{y} & = & \text{if } exists\,\overline{x} \text{ then } \overline{x} \text{ else } \overline{y}. \end{array}$$

The ? operation is associative and has *Nothing* as a unit. It appeared in early versions of ML [GMW79], and similar operators appear in other languages. As an example of its use, the term

$$the\,([x-1 \mid x \geq 1]^{Maybe}\ ?\ [0]^{Maybe})$$

returns the predecessor of x if it is non-negative, and zero otherwise.

In [Wad85] it was proposed to use lists to represent exceptions, encoding a value x by the unit list, and an exception by the empty list. This corresponds to the map

$$\begin{array}{lcl} maybe & :: & Maybe\,x \rightarrow List\,x \\ maybe\,(Just\,x) & = & [x]^{List} \\ maybe\,Nothing & = & [\,]^{List} \end{array}$$

which is a monad morphism from *Maybe* to *List*. We have that

$$maybe\,(\overline{x}\,?\,\overline{y}) \subseteq (maybe\,\overline{x}) +\!\!+ (maybe\,\overline{y}),$$

where $\subseteq$ is the sublist relation. Thus, exception comprehensions can be represented by list comprehensions, and biased choice can be represented by list concatenation. The argument in [Wad85] that list comprehensions provide a convenient notation for manipulating exceptions can be mapped, via this morphism, into an argument in favour of exception comprehensions!

7.3 Continuations

Fix a type R of results. The monad of continuations is given by

$$\begin{array}{lcl} \text{type } Cont\,x & = & (x \rightarrow R) \rightarrow R \\ map^{Cont}\,f\,\overline{x} & = & \lambda k \rightarrow \overline{x}\,(\lambda x \rightarrow k\,(f\,x)) \\ unit^{Cont}\,x & = & \lambda k \rightarrow k\,x \\ join^{Cont}\,\overline{\overline{x}} & = & \lambda k \rightarrow \overline{\overline{x}}\,(\lambda \overline{x} \rightarrow \overline{x}\,(\lambda x \rightarrow k\,x)). \end{array}$$

A continuation of type x takes a continuation function $k :: x \rightarrow R$, which specifies how to take a value of type x into a result of type R, and returns a result of type R. The unit takes a value x into the continuation $\lambda k \rightarrow k\,x$ that applies the continuation function to the given value. It follows from these definitions that

$$\begin{array}{l} [(x,y) \mid x \leftarrow \overline{x};\ y \leftarrow \overline{y}]^{Cont} \\ \quad = \lambda k \rightarrow \overline{x}\,(\lambda x \rightarrow \overline{y}\,(\lambda y \rightarrow k\,(x,y))). \end{array}$$

This can be read as follows: evaluate $\overline{x}$, bind x to the result, then evaluate $\overline{y}$, bind y to the result, then return the pair (x,y).

A useful operation in this monad is

$$\begin{array}{lcl} callcc & :: & ((x \rightarrow Cont\,y) \rightarrow Cont\,x) \rightarrow Cont\,x \\ callcc\,g & = & \lambda k \rightarrow g\,(\lambda x \rightarrow \lambda k' \rightarrow k\,x)\,k. \end{array}$$

This mimics the "call with current continuation" (or *call-cc*) operation popular from Scheme [RC86]. For example, the Scheme program

```
(call-cc (lambda (esc)
            (/ x (if (= y 0) (esc 42) y))))
```

translates to the equivalent program

$$\begin{array}{l} callcc\,(\lambda esc \rightarrow \\ \qquad [x/z \mid z \leftarrow \text{if } y == 0 \text{ then } esc\,42 \text{ else } y\,]^{Cont}. \end{array}$$

Both of these programs bind *esc* to an escape function that returns its argument as the value of the entire *callcc* expression. They then return the value of x divided by y, or return 42 in the case that y is zero.

8 Translation

In Section 4, we saw that a function of type $x \rightarrow y$ in an (impure) functional language that manipulates state corresponds to a function of type $x \rightarrow ST\,y$ in a (pure) functional program. The correspondence was drawn in an informal way, so we might ask, what assurance is there that *every* program can be translated in a similar way? This section provides that assurance, in the form of a translation of lambda calculus into an arbitrary monad. This allows us to translate not only programs that manipulate state, but also programs that raise exceptions, call continuations, and so on. Indeed, we shall see that there are two translations, one call-by-value and one call-by-name. The target language of both translations is a pure, non-strict λ-calculus, augmented with M-comprehensions.

We will perform our translations on a simple typed lambda calculus. We will use T, U, V to range over types, and K to range over base types. A type is either a base type, function type, or product type:

$$T, U, V \ ::= \ K \mid (U \rightarrow V) \mid (U, V).$$

We will use t, u, v to range over terms, and x to range over variables. A term is either a variable, an abstraction, an application, a pair, or a selection:

$$t, u, v \ ::= \ x \mid (\lambda x \rightarrow v) \mid (t\ u) \mid (u, v) \mid (\mathit{fst}\ t) \mid (\mathit{snd}\ t).$$

We will use A to range over assumptions, which are lists associating variables with types:

$$A \ ::= \ x_1 :: T_1, \ldots, x_n :: T_n.$$

We write the typing $A \vdash t :: T$ to indicate that under assumption A the term t has type T. The inference rules for well-typings in this calculus are well known, so they are omitted to save space.

The call-by-value translation of *lambda*-calculus into a monad M is given in Figure 7. The translation of the type T is written T^* and the translation of the term t is written t^*. The rule for translating function types,

$$(U \rightarrow V)^* \ = \ U^* \rightarrow M\ V^*,$$

can be read "a call-by-value function takes as its argument a *value* of type U and returns a *computation* of type V." This corresponds to the translation in Section 4, where a function of type $x \rightarrow y$ in the (impure) source language is translated to a function of type $x \rightarrow M\ y$ in the (pure) target language. Each of the rules for translating terms has a straightforward computational reading. For example, the rule for applications,

$$(t\ u)^* \ = \ [\, z \mid x \leftarrow t^*;\ y \leftarrow u^*;\ z \leftarrow (x\ y)\,]^M,$$

can be read "to apply t to u, first evaluate t (call the result x), then evaluate u (call the result y), then apply x to y (call the result z) and return z." This is what one would expect in a call-by-value language – the argument is evaluated before the function is applied. If

$$x_1 :: T_1, \ldots, x_n :: T_n \vdash t :: T$$

is a well-typing in the source language, then its translation

$$x_1 :: T_1^*, \ldots, x_n :: T_n^* \vdash t^* :: M\ T^*$$

Types

$$\begin{aligned} K^* &= K \\ (U \rightarrow V)^* &= (U^* \rightarrow M\ V^*) \\ (U, V)^* &= (U^*, V^*) \end{aligned}$$

Terms

$$\begin{aligned} x^* &= [\, x \,]^M \\ (\lambda x \rightarrow v)^* &= [\, (\lambda x \rightarrow v^*) \,]^M \\ (t\ u)^* &= [\, z \mid x \leftarrow t^*;\ y \leftarrow u^*;\ z \leftarrow (x\ y) \,]^M \\ (u, v)^* &= [\, (x, y) \mid x \leftarrow u^*;\ y \leftarrow v^* \,]^M \\ (\mathit{fst}\ t)^* &= [\, (\mathit{fst}\ x) \mid x \leftarrow t^* \,]^M \end{aligned}$$

Environments

$$\begin{aligned} &(x_1 :: T_1, \ldots, x_n :: T_n)^* \\ &\qquad = \ x_1 :: T_1^*, \ldots, x_n :: T_n^* \end{aligned}$$

Typings

$$(A \vdash t :: T)^* \ = \ A^* \vdash t^* :: M\ T^*$$

Figure 7: Call-by-value translation.

is a well-typing in the target language. Like the arguments of a function, the free variables correspond to *values*, while, like the result of a function, the term corresponds to a *computation*.

The call-by-name translation of λ-calculus into a monad M is given in Figure 8. Now the translation of the type T is written $T^\dagger$ and the translation of the term t is written $t^\dagger$. The rule for translating function types,

$$(U \rightarrow V)^\dagger \ = \ M\ U^\dagger \rightarrow M\ V^\dagger,$$

can be read "a call-by-name function takes as its argument a *computation* of type U and returns a *computation* of type V." The rule for applications,

$$(t\ u)^\dagger \ = \ [\, y \mid x \leftarrow t^\dagger;\ y \leftarrow (x\ u^\dagger)\,]^M,$$

can be read "to apply t to u, first evaluate t (call the result x), then apply x to the term u (call the result y) and return y." This is what one would expect in a call-by-name language – the argument u is passed unevaluated, and is evaluated each time it is used. The well-typing in the source language given previously now translates to

$$x_1 :: M\ T_1^\dagger, \ldots, x_n :: M\ T_n^\dagger \vdash t^\dagger :: M\ T^\dagger,$$

which is again a well-typing in the target language. This time both the free variables and the term correspond to

Types

$$\begin{aligned}
K^\dagger &= K\\
(U \to V)^\dagger &= (M\ U^\dagger \to M\ V^\dagger)\\
(U, V)^\dagger &= (M\ U^\dagger,\ M\ V^\dagger)
\end{aligned}$$

Terms

$$\begin{aligned}
x^\dagger &= x\\
(\lambda x \to v)^\dagger &= [\,(\lambda x \to v^\dagger)\,]^M\\
(t\ u)^\dagger &= [\,y \mid x \leftarrow t^\dagger;\ y \leftarrow (x\ u^\dagger)\,]^M\\
(u, v)^\dagger &= [\,(u^\dagger, v^\dagger)\,]^M\\
(\mathit{fst}\ t)^\dagger &= [\,y \mid x \leftarrow t^\dagger;\ y \leftarrow (\mathit{fst}\ x)\,]^M
\end{aligned}$$

Environments

$$\begin{aligned}
&(x_1 :: T_1, \ldots, x_n :: T_n)^\dagger\\
&\quad = x_1 :: M\ T_1^\dagger, \ldots, x_n :: M\ T_n^\dagger
\end{aligned}$$

Typings

$$(A \vdash t :: T)^\dagger = A^\dagger \vdash t^\dagger :: M\ T^\dagger$$

Figure 8: Call-by-name translation.

computations, reflecting that in a call-by-value language the free variables correspond to computations (or closures) that must be evaluated each time they are used.

In particular, the call-by-value intrepretation in the strictness monad *Str* of Section 3.2 yields the usual strict semantics of λ-calculus, whereas the call-by-name interpretation in the same monad yields the usual lazy semantics.

If we use the monad of, say, state transformers then the call-by-value interpretation yields the usual semantics of a λ-calculus with assignment. The call-by-name interpretation yields a semantics where the state transformation specified by a variable occurs *each* time the variable is accessed. This explains why the second translation is titled call-by-name rather than call-by-need. Of course, since the target of both translations is a pure, non-strict λ-calculus, there is no problem with executing translated programs under a lazy (i.e., call-by-need) implementation.

8.1 Example: Non-determinism

As a more detailed example of the application of the translation schemes, consider a small non-deterministic language. This consists of the λ-calculus as defined above with its syntax extended to include a non-deterministic choice operator ($\sqcup$) and simple arithmetic:

$$t, u, v \ ::= \ \cdots \mid (t \sqcup u) \mid n \mid (t + u),$$

where n ranges over integer constants. This language is typed just as for lambda calculus. We assume a base type *Int*, and the additional constructs typed as follows: for any type T, if $t :: T$ and $u :: T$ then $(t \sqcup u) :: T$; and $n :: \mathit{Int}$; and if $t :: \mathit{Int}$ and $u :: \mathit{Int}$ then $(t + u) :: \mathit{Int}$. For example, the term

$$((\lambda a \to a + a)\,(1 \sqcup 2))$$

has the type *Int*. Under a call-by-value interpretation we would expect this to return either 2 or 4 (i.e., $1 + 1$ or $2 + 2$), whereas under a call-by-name interpretation we would expect this to return 2 or 3 or 4 (i.e., $1 + 1$ or $1 + 2$ or $2 + 1$ or $2 + 2$).

We will give the semantics of this language by interpreting the λ-calculus in the set monad. This is specified, as one would expect, by

$$\begin{aligned}
\mathit{map}^{\mathit{Set}} f\,\overline{x} &= \{\, f\,x \mid x \in \overline{x} \,\}\\
\mathit{unit}^{\mathit{Set}}\,x &= \{\, x \,\}\\
\mathit{join}^{\mathit{Set}}\,\overline{\overline{x}} &= \bigcup \overline{\overline{x}}.
\end{aligned}$$

It is easy to check that the resulting *Set*-comprehensions correspond to the traditional usage, and in what follows we will write $\{\, t \mid q \,\}$ in preference to the more cumbersome $[\, t \mid q \,]^{\mathit{Set}}$.

The call-by-value interpretation for this language is provided by the rules in Figure 7, choosing M to be the monad *Set*, together with the rules:

$$\begin{aligned}
t \sqcup u^* &= t^* \cup u^*\\
n^* &= \{n\}\\
t + u^* &= \{\, x + y \mid x \leftarrow t^*;\ y \leftarrow u^* \,\}.
\end{aligned}$$

These rules translate a term of type T in the non-deterministic language into a term of type *Set* T in the pure functional language (which includes monad comprehensions). For example, the term above translates to

$$\begin{aligned}
\{\, z \mid\ & x \leftarrow \{\, (\lambda a \to \{\, x' + y' \mid x' \leftarrow \{a\};\ y' \leftarrow \{a\} \,\}) \,\};\\
& y \leftarrow \{1\} \cup \{2\};\\
& z \leftarrow (x\ y) \,\}
\end{aligned}$$

which has the value $\{2, 4\}$, as expected.

The call-by-name translation of the same language is provided by the rules in Figure 8. The rules for $(t \sqcup u)$, n, and $(t + u)$ are the same as the call-by-value rules, replacing replacing $(-)^*$ with $(-)^\dagger$. Now the same term translates to

$$\begin{aligned}
\{\, y \mid\ & x \leftarrow \{\, (\lambda a \to \{\, x' + y' \mid x' \leftarrow a;\ y' \leftarrow a \,\}) \,\};\\
& y \leftarrow x\,(\{1\} \cup \{2\}) \,\}
\end{aligned}$$

$$\begin{aligned}
x^* &= \lambda k \rightarrow k\, x \\
(\lambda x \rightarrow v)^* &= \lambda k \rightarrow k\, (\lambda x \rightarrow v^*) \\
(t\, u)^* &= \lambda k \rightarrow t^*\, (\lambda x \rightarrow u^*\, (\lambda y \rightarrow x\, y\, k)) \\
(u, v)^* &= \lambda k \rightarrow u^*\, (\lambda x \rightarrow v^*\, (\lambda y \rightarrow k\, (u, v))) \\
(\mathit{fst}\, t)^* &= \lambda k \rightarrow t^*\, (\lambda x \rightarrow k\, (\mathit{fst}\, x))
\end{aligned}$$

Figure 9: Continuation-passing style

which has the value $\{2,3,4\}$, as expected.

A similar approach to non-deteminism is taken by Hughes and O'Donnell [HO89]. They suggest adding a set type to a lazy functional language where a set is actually represented by a non-deterministic choice of one of the elements of the set. The primitive operations they provide on sets are just *map*, *unit*, and *join* of the set monad, plus set union ($\cup$) to represent non-deterministic choice. They address the issue of how such sets should behave with respect to $\perp$, and present an elegant derivation of a non-deterministic, parallel, tree search algorithm. However, they provide no argument that any program in a traditional non-deterministic functional language can be encoded in their approach. Such an argument is provided by the translation scheme above.

8.2 Example: Continuations

As a final example, consider the call-by-value interpretation under the monad of continuations, *Cont*, given in Section 7.3. Applying straightforward calculation to simplify the *Cont*-comprehensions yields the translation scheme given in Figure 9, which is simply the continuation-passing style beloved by many theorists and compiler writers. It is left to the reader to perform a similar calculation on the other translation scheme to yield a call-by-value version of continuation-passing style (which is less well known, but can be found in [Rey74]).

Acknowledgements

This work owes a great deal to Eugenio Moggi: I thank him for his ideas, and for the time he took to explain them to me. I thank John Launchbury for his enthusiasm and suggestions, and for helpful comments I thank Arvind, Olivier Danvy, Kevin Hammond, John Hughes, Karsten Kehler Holst, Austin Melton, Nikhil, Simon Peyton Jones, Andy Pitts, Andre Scedrov, Carolyn Talcott, Phil Trinder, and attenders of the 1989 Glasgow Summer School on Category Theory and Constructive Logic. And special thanks to Catherine Lyons for added commas and hyphens.

References

[Blo89] A. Bloss, Update analysis and the efficient implementation of functional aggregates. In *4'th Symposium on Functional Programming Languages and Computer Architecture*, ACM, London, September 1989.

[BW85] M. Barr and C. Wells, *Toposes, Triples, and Theories*. Springer Verlag, 1985.

[BW88] R. Bird and P. Wadler, *Introduction to Functional Programming*. Prentice Hall, 1988.

[Fai87] J. Fairbairn, Form follows function. *Software - Practice and Experience*, 17(6):379–386, June 1987.

[FL89] R. Frost and J. Launchbury, Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, **32**(2):108–121, April 1989.

[Gog88] J. A. Goguen, Higher order functions considered unnecessary for higher order programming. Technical report SRI-CSL-88-1, SRI International, January 1988.

[GL88] D. K. Gifford and J. M. Lucassen, Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, pp. 28–39, Cambridge, Massachusetts, August 1986.

[GH90] J. Guzmán and P. Hudak, Single-threaded polymorphic lambda calculus. In *IEEE Symposium on Logic in Computer Science*, Philadelphia, June 1990.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF*. LNCS 78, Springer-Verlag, 1979.

[Hol83] S. Holmström, A simple and efficient way to handle large data structures in applicative languges. In *Proceedings SERC/Chalmers Workshop on Declarative Programming*, University College London, 1983.

[Hud86a] P. Hudak, A semantic model of reference counting and its abstraction (detailed summary). In *ACM Conference on Lisp and*

Functional Programming, pp. 351–363, Cambridge, Massachusetts, August 1986.

[HMT88] R. Harper, R. Milner, and M. Tofte, The definition of Standard ML, version 2. Report ECS-LFCS-88-62, Edinburgh University, Computer Science Dept., 1988.

[Hud86b] P. Hudak, Arrays, non-determinism, side-effects, and parallelism: a functional perspective. In J. H. Fasel and R. M. Keller, editors, *Workshop on Graph Reduction*, Santa Fe, New Mexico, September–October 1986. LNCS 279, Springer-Verlag, 1986.

[Hug89] J. Hughes, Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[HO89] J. Hughes and J. O'Donnell, Expressing and reasoning about non-deterministic functional programs. Glasgow Workshop on Functional Programming, Fraserburgh, August 1989.

[HW90] P. Hudak and P. Wadler, editors, *Report on the Programming Language Haskell.* Technical report, Yale University and Glasgow University, April 1990.

[LS86] J. Lambek and P. Scott, *Introduction to Higher Order Categorical Logic*, Cambridge University Press, 1986.

[Mac71] S. Mac Lane, *Categories for the Working Mathematician*, Springer-Verlag, 1971.

[Mil84] R. Milner, A proposal for Standard ML. In *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[MT89] I. Mason and C. Talcott, Axiomatising operational equivalence in the presence of side effects. In *IEEE Symposium on Logic in Computer Science*, Asilomar, California, June 1989.

[Mog89a] E. Moggi, Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, Asilomar, California, June 1989. (A longer version is available as a technical report from the University of Edinburgh.)

[Mog89b] E. Moggi, An abstract view of programming languges. Course notes, University of Edinburgh.

[RC86] J. Rees and W. Clinger (eds.), The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79 (1986).

[Rey74] J. C. Reynolds, On the relation between direct and continuation semantics. In *Colloquium on Automata, Languages and Programming*, Saarbrücken, July–August 1974, LNCS 14, Springer-Verlag, 1974.

[Rey83] J. C. Reynolds, Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, 513–523, North-Holland, Amsterdam.

[Sch85] D. A. Schmidt, Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7:299–310, 1985.

[Spi89] M. Spivey, Term rewriting without exceptions, *Science of Computer Programming*, 1989.

[Tur82] D. A. Turner, Recursion equations as a programming language. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, Cambridge University Press, 1982.

[Tur85] D. A. Turner, Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2'nd International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. LNCS 201, Springer Verlag, 1985.

[Wad85] P. Wadler, How to replace failure by a list of successes. In *2'nd Symposium on Functional Programming Languages and Computer Architecture*, Nancy, September 1985. LNCS 273, Springer-Verlag, 1985.

[Wad87] P. Wadler, List comprehensions. In S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[Wad89] P. Wadler, Theorems for free! In *4'th Symposium on Functional Programming Languages and Computer Architecture*, ACM, London, September 1989.

[Wad90] P. Wadler, Linear types can change the world! In *IFIP Working Conference on Programming Concepts and Methods*, Sea of Gallilee, Israel, April 1990.