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 blessings 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 [HPW91] 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).

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

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

This paper appeared in *Mathematical Structures in Computer Science* volume 2, pp. 461–493, 1992; copyright Cambridge University Press. This version corrects a few small errors in the published version. An earlier version appeared in *ACM Conference on Lisp and Functional Programming*, Nice, June 1990.

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 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*. This generalises 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. Under this approach, the usual typing discipline (e.g., Hindley-Milner extended with abstract data types) is sufficient to guarantee that array update may safely be implemented by overwriting. To my knowledge, this solution has never been proposed before, and its discovery comes as 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, Wad91]. That monads led to the discovery of this solution must count as 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 and unify some previous proposals for incorporating various features into functional languages: exceptions [Wad85, Spi90], parsers [Wad85, Fai87, FL89], and non-determinism [HO89]. In particular, Spivey's work [Spi90] 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: this fits the work of Hughes and O'Donnell [HO89] into the more general framework given here. The other is to apply the call-by-value scheme in the monad of continuations: the result is the familiar continuation-passing style transformation. It remains an open question whether there is a translation scheme that corresponds to call-by-need as opposed to call-by-name.

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 [HPW91], 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 "let" 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, this is usually 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 category theory, the notions of *type* and *function* are generalised to *object* and *arrow*. An operator M taking each object x into an object $M\,x$, combined with an operator *map* taking each arrow $f :: x \to y$ into an arrow $map\,f :: M\,x \to M\,y$, and satisfying (i) and (ii), is called a *functor*. Categorists prefer to use the same symbol for both operators, and so would write $M\,f$ where we write $map\,f$.

The function *unit* converts a value into a singleton lists, and the function *join* concatenates a list of lists into a list:

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

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

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

Laws (iii) and (iv) may be derived by a systematic transformation of the polymorphic types of *unit* and *join*. The idea of deriving laws from types goes by the slogan "theorems for free" [Wad89] and is a consequence of Reynolds' abstraction theorem for polymorphic lambda calculus [Rey83].

In categorical terms, *unit* and *join* are *natural transformations*. Rather than treat *unit* as a single function with a polymorphic type, categorists treat it as a family of arrows, $unit_x :: x \to M\,x$, one for each object x, satisfying $map\,f \cdot unit_x = unit_y \cdot f$ for any objects x and y and any arrow $f :: x \to y$ between them. They treat *join* similarly. Natural transformation is a simpler concept than polymorphic function, but we will stick with polymorphism since it's a more familiar concept to functional programmers.

2.2 Comprehensions

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

$$
[\,(x,y) \mid x \leftarrow [1,2],\ y \leftarrow [3,4]\,] \quad = \quad [(1,3),(1,4),(2,3),(2,4)].
$$

In general, a comprehension has the form $[\,t \mid q\,]$, where t is a term and q is a qualifier. We 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:

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

(In Haskell, λ-terms are written $(\lambda x \rightarrow t)$ 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 but not vice-versa.

For those familiar with list comprehensions, the empty qualifier and the parentheses in qualifier compositions will appear strange. This is because they are not needed. We will shortly prove that qualifier composition is associative and has the empty qualifier as unit. Thus we need not write parentheses in qualifier compositions, since $((p, q), r)$ and $(p, (q, r))$ are equivalent, and we need not write (q, Λ) or (Λ, q) because both are equivalent to the simpler q. The only remaining use of Λ is to write $[\, t \mid \Lambda\,]$, which we abbreviate $[\, t\,]$.

Most languages that include list comprehensions also allow another form of qualifier, known as a *filter*, the treatment of which is postponed until Section 5.

As a simple example, we have:

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

The comprehension in the initial example is computed as:

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

From (i)–(iv) and (1)–(3) we may derive further laws:

$$
\begin{array}{llcl}
(4) & [\, f\ t \mid q\,] & = & map\, f\, [\, t \mid q\,], \\
(5) & [\, x \mid x \leftarrow u\,] & = & u, \\
(6) & [\, t \mid p,\ x \leftarrow [u|q],\ r\,] & = & [\, t^u_x \mid p,\ q,\ r^u_x\,].
\end{array}
$$

In (4) function f must contain no free occurrences of variables bound by qualifier q, and in (6) the term t^u_x stands for term t with term u substituted for each free occurrence of variable x, and similarly for the qualifier r^u_x. Law (4) is proved by induction over the structure of qualifiers; the proof uses laws (*ii*)–(*iv*) and (*1*)–(*3*). Law (*5*) is an immediate consequence of laws (*i*) and (*2*). Law (*6*) is again proved by induction over the structure of qualifiers, and the proof uses laws (*1*)–(*4*).

As promised, we now show that qualifier composition is associative and has the empty qualifier as a unit:

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

First, observe that (I')–(III') are equivalent, respectively, to the following:

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

To see that (II') and (II) are equivalent, start with the left side of (II') and simplify:

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

That (II) implies (II') is immediate. For the converse, take $[\,t \mid q\,]$ to be $[\,x \mid x \leftarrow u\,]$ and apply (*5*). The other two equivalences are seen similarly.

Second, observe that laws (I)–(III) do indeed hold. For example:

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

Use induction over lists to prove (I) and (II), and over list of lists to prove (III).

2.3 Monads

The comprehension notation suits data structures other than lists. Sets and bags are obvious examples, and we shall encounter many others. Inspection of the foregoing lets us isolate the conditions under which a comprehension notation is sensible.

For our purposes, a *monad* is an operator M on types together with a triple of functions

$$
\begin{array}{lcl}
\mathit{map} & :: & (x \to y) \to (M\; x \to M\; y), \\
\mathit{unit} & :: & x \to M\; x, \\
\mathit{join} & :: & M\,(M\; x) \to M\; x,
\end{array}
$$

satisfying laws (*i*)–(*iv*) and (*I*)–(*III*).

Every monad gives rise to a notion of comprehension via laws (*1*)–(*3*). The three laws establish a correspondence between the three components of a monad and the three forms of qualifier: (*1*) associates *unit* with the empty qualifier, (*2*) associates *map* with generators, and (*3*) associates *join* with qualifier composition. The resulting notion of comprehension is guaranteed to be sensible in that it necessarily satisfies laws (*4*)–(*6*) and (*I′*)–(*III′*).

In what follows, we will need to distinguish many monads. We write M alone to stand for the monad, leaving the triple $(map^M, unit^M, join^M)$ implicit, and we write $[\,t \mid q\,]^M$ to indicate in which monad a comprehension is to be interpreted. The monad of lists as described above will be written *List*.

As an example, take *Set* to be the set type constructor, map^{Set} to be the image of a set under a function, $unit^{Set}$ to be the function that takes an element into a singleton set, and $join^{Set}$ to be the union of a set of sets:

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

The resulting comprehension notation is the familiar one for sets. For instance, $[\,(x, y) \mid x \leftarrow \overline{x},\ y \leftarrow \overline{y}\,]^{Set}$ specifies the cartesian product of sets $\overline{x}$ and $\overline{y}$.

We can recover *unit*, *map*, and *join* from the comprehension notation:

$$\begin{array}{llcl} (1') & unit\, x & = & [\, x\,] \\ (2') & map\, f\, \overline{x} & = & [\, f\, x \mid x \leftarrow \overline{x}\,] \\ (3') & join\, \overline{\overline{x}} & = & [\, x \mid \overline{x} \leftarrow \overline{\overline{x}},\ x \leftarrow \overline{x}\,]. \end{array}$$

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

Thus not only can we derive comprehensions from monads, but we can also derive monads from comprehensions. Define a *comprehension structure* to be any interpretation of the syntax of comprehensions that satisfies laws (*5*)–(*6*) and (*I′*)–(*III′*). Any monad gives rise to a comprehension structure, via laws (*1*)–(*3*); as we have seen, these imply (*4*)–(*6*) and (*I′*)–(*III′*). Conversely, any comprehension structure gives rise to a monad structure, via laws (*1′*)–(*3′*); it is easy to verify that these imply (*i*)–(*iv*) and (*1*)–(*4*), and hence (*I*)–(*III*).

The concept we arrived at by generalising list comprehensions, mathematicians arrived at by a rather different route. It first arose in homological algebra in the 1950's with the undistinguished name "standard construction" (sort of a mathematical equivalent of "hey you"). The next name, "triple", was not much of an improvement. Finally it was baptised a "monad". Nowadays it can be found in any standard text on category theory [Mac71, BW85, LS86].

The concept we call a monad is slightly stronger than what a categorist means by that name: we are using what a categorist would call a *strong monad* in a *cartesian closed* *category*. Rougly speaking, a category is cartesian closed if it has enough structure to interpret λ-calculus. In particular, associated with any pair of objects (types) x and y there is an object $[x \rightarrow y]$ representing the space of all arrows (functions) from x to y. Recall that M is a functor if for any arrow $f :: x \rightarrow y$ there is an arrow $map\, f :: M\, x \rightarrow M\, y$ satisfying (i) and (ii). This functor is strong if it is itself represented by a single arrow $map :: [x \rightarrow y] \rightarrow [M\, x \rightarrow M\, y]$. This is all second nature to a generous functional programmer, but a stingy categorist provides such structure only when it is needed.

It is needed here, as evidenced by Moggi's requirement that a computational monad have a *strength*, a function $t :: (x, M\, y) \rightarrow M\,(x, y)$ satisfying certain laws [Mog89a]. In a cartesian closed category, a monad with a strength is equivalent to a monad with a strong functor as described above. In our framework, the strength is defined by $t\,(x, \overline{y}) = [\,(x, y) \mid y \leftarrow \overline{y}\,]$. (Following Haskell, we write (x, y) for pairs and also (x, y) for the corresponding product type.)

Monads were conceived in the 1950'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 "let" term:

$$\begin{array}{cl}
 & [\,t \mid x \leftarrow u\,]^{Id} \\
= & ((\lambda x \rightarrow t)\, u) \\
= & (\text{let } x = u \text{ in } t).
\end{array}$$

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

$$[\,t \mid x \leftarrow u,\ y \leftarrow v\,]^{Id} \;=\; (\text{let } x = u \text{ in } (\text{let } y = v \text{ in } t)).$$

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

In the Hindley-Milner type system, λ-terms and "let" terms differ in that the latter may introduce polymorphism. The key factor allowing "let" 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, $\mathit{strict}\, f\, x$ 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 the same as the identity monad, except for the definition of $\mathit{map}^{\mathit{Str}}$. Monad laws (*i*), (*iii*)–(*iv*), and (*I*)–(*III*) are satisfied, but law (*ii*) becomes an inequality,

$$\mathit{map}^{\mathit{Str}}\, g \cdot \mathit{map}^{\mathit{Str}}\, f \sqsubseteq \mathit{map}^{\mathit{Str}}\, (g \cdot f).$$

So *Str* is not quite a monad; categorists might call it a *lax monad*. Comprehensions for lax monads are defined by laws (*1*)–(*3*), just as for monads. Law (*5*) remains valid, but laws (*4*) and (*6*) become inequalities.

We will use *Str*-comprehensions to control the evaluation order of lazy programs. 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 to 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

$$\begin{aligned}
\text{type } ST\, x &= S \to (x, S) \\
map^{ST} f\, \overline{x} &= \lambda s \to [\,(f\, x, s') \mid (x, s') \leftarrow \overline{x}\, s\,]^{Id} \\
unit^{ST}\, x &= \lambda s \to (x, s) \\
join^{ST}\, \overline{\overline{x}} &= \lambda s \to [\,(x, s'') \mid (\overline{x}, s') \leftarrow \overline{\overline{x}}\, s,\ (x, s'') \leftarrow \overline{x}\, s'\,]^{Id}.
\end{aligned}$$

(Recall the equivalence of Id-comprehensions and "let" 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 $\lambda s \to (x, s)$ that returns x and leaves the state unchanged. We have that

$$[\,(x, y) \mid x \leftarrow \overline{x},\ y \leftarrow \overline{y}\,]^{ST} \;=\; \lambda s \to [\,((x, y), s'') \mid (x, s') \leftarrow \overline{x}\, s,\ (y, s'') \leftarrow \overline{y}\, s'\,]^{Id}.$$

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

Two useful operations in this monad are

$$\begin{aligned}
fetch &:: ST\, S \\
fetch &= \lambda s \to (s, s) \\[1ex]
assign &:: S \to ST\, () \\
assign\, s' &= \lambda s \to ((), s').
\end{aligned}$$

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

$$\begin{aligned}
init &:: S \to ST\, x \to x \\
init\, s\, \overline{x} &= [\, x \mid (x, s') \leftarrow \overline{x}\, s\,]^{Id}.
\end{aligned}$$

This applies the state transformer $\overline{x}$ 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

$$mkname :: Int \to Name$$

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

$$subst :: Name \to Name \to Term \to Term$$

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

A solution to this problem in the impure functional language Standard ML is shown in Figure 1. The impure feature we are concerned with here is state: the solution uses a reference N to an assignable location containing an integer. The "functions" and their types are:

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

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 so requires the dummy parameter () to give it the form of a "function". However, *rename* is a true function, since 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:

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

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:

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

The monadic program is simply a different way of writing the pure program: expanding the monad comprehensions in Figure 3 and simplifying would yield the program in Figure 2.

Types in the monadic program can be seen to correspond directly to the types in the impure program: an impure "function" of type $U \to V$ that affects the state corresponds to a pure function of type $U \to ST\ V$. Thus, *renamer* has type $Term \to Term$ in the impure program, and type $Term \to ST\ Term$ in the monadic program; and *newname* has type $() \to Name$ in the impure program, and type $ST\ Name$, which is isomorphic to $() \to ST\ Name$, in the pure program. Unlike the impure program, types in the monadic program make 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 following laws on qualifiers:

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

and on terms:

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

These, together with the comprehension laws (5), (6), and $(I')--(III')$, 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

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

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,

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

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:

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

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 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 following data types:

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

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:

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

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.

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

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

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

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

Here $\widehat{x}$ is a variable of type $SR\; x$, just as $\overline{x}$ 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. We have that

$$[\,(x, y) \mid x \leftarrow \widehat{x},\; y \leftarrow \widehat{y}\,]^{SR} \;=\; \lambda s \rightarrow [\,(x, y) \mid x \leftarrow \widehat{x}\; s,\; y \leftarrow \widehat{y}\; s\,]^{Id}.$$

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

It is easy to see that

$$[\,(x, y) \mid x \leftarrow \widehat{x},\; y \leftarrow \widehat{y}\,]^{SR} \;=\; [\,(x, y) \mid y \leftarrow \widehat{y},\; x \leftarrow \widehat{x}\,]^{SR},$$

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

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

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

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

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:

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

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

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

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 two versions are equivalent appears in Section 6.

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

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

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. (Related type systems are discussed in [GH90, Wad91].) 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,

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

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 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 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!

Recall that for Λ we introduced a function $unit :: x \to M\, x$ satisfying the laws

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

and then defined $[\,t \mid \Lambda\,] = unit\; t$.

Similarly, for $\emptyset$ we introduce a function

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

satisfying the laws

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

and define

$$(7) \qquad [\,t \mid \emptyset\,] \;=\; zero\; t.$$

Law (v) specifies that the result of $zero$ is independent of its argument, and can be derived from the type of $zero$ (again, see [Rey83, Wad89]). In the case of lists, setting $zero\; y = [\,]$ makes laws (IV) and (V) hold, since $join\,[\,] = [\,]$ and $join\,[[\,], \ldots, [\,]] = [\,]$. (This ignores what happens when $zero$ is applied to $\bot$, which will be considered below.)

Now, for a monad with zero we can extend comprehensions to contain a new form of qualifier, the filter, defined by

$$(8) \qquad [\,t \mid b\,] \;=\; \text{if } b \text{ then } [t] \text{ else } [\,],$$

where b is any boolean-valued term. Recall that laws (4) and (6) were proved by induction on the form of qualifiers; we can show that for the new forms of qualifiers, defined by (7) and (8), they still hold. We also have new laws

$$\begin{array}{lrcl} (9) & [\,t \mid b,\, c\,] & = & [\,t \mid (b \wedge c)\,], \\ (10) & [\,t \mid q,\, b\,] & = & [\,t \mid b,\, q\,], \end{array}$$

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

When dealing with $\perp$ 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 $join\,(map\ zero)\,\perp\ =\ \perp$ but $zero\,\perp\ =\ [\,]$. In this case, laws (1)–(9) are still valid, but law (10) holds only if $[\,t \mid q\,] \neq \perp$. In the case that $[\,t \mid q\,] = \perp$, law (10) 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 $zero^{Str}\ y = \perp$. It is easy to verify that the required laws hold; unlike with lists, the laws hold even when *zero* is applied to $\perp$. For example, $[\,x - 1 \mid x \geq 1\,]^{Str}$ returns one less than x if x is positive, and $\perp$ otherwise.

6 Monad morphisms

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

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

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:

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

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

$$(11) \qquad h\,[\,t \mid q\,]^M \;=\; [\,t \mid (h\ q)\,]^N$$

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 $unit^M :: x \to M\ x$ 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 $exp^{ST} :: Exp \to ST\ Val$ and $exp^{SR} :: Exp \to SR\ Val$ 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. If the expression has the form $(Plus\ e_1\ e_2)$, we have that

$$
\begin{array}{ll}
 & ro\,(exp^{SR}\,(Plus\ e_1\ e_2)) \\
= & \{\text{unfolding } exp^{SR}\} \\
 & ro\,[\,v_1 + v_2 \mid v_1 \leftarrow exp^{SR}\ e_1,\ v_2 \leftarrow exp^{SR}\ e_2\,]^{SR} \\
= & \{\text{by } (11)\} \\
 & [\,v_1 + v_2 \mid v_1 \leftarrow ro\,(exp^{SR}\ e_1),\ v_2 \leftarrow ro\,(exp^{SR}\ e_2)\,]^{ST} \\
= & \{\text{hypothesis}\} \\
 & [\,v_1 + v_2 \mid v_1 \leftarrow exp^{ST}\ e_1,\ v_2 \leftarrow exp^{ST}\ e_2\,]^{ST} \\
= & \{\text{folding } exp^{ST}\} \\
 & exp^{ST}\,(Plus\ e_1\ e_2).
\end{array}
$$

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 (11) holds even when q contains filters.

7 More monads

This section describes four more monads: parsers, expressions, input-output, and continuations. The basic techniques are not new (parsers are discussed in [Wad85, Fai87, FL89], and exceptions are discussed in [Wad85, Spi90]), but monads and monad comprehensions provide a convenient framework for their expression.

7.1 Parsers

The monad of parsers is given by

$$
\begin{array}{lcl}
\text{type } Parse\ x & = & String \to List\,(x, String) \\
map^{Parse}\,f\,\overline{x} & = & \lambda i \to [\,(f\ x, i') \mid (x, i') \leftarrow \overline{x}\ i\,]^{List} \\
unit^{Parse}\ x & = & \lambda i \to [\,(x, i)\,]^{List} \\
join^{Parse}\,\overline{\overline{x}} & = & \lambda i \to [\,(x, i'') \mid (\overline{x}, i') \leftarrow \overline{\overline{x}}\ i,\ (x, i'') \leftarrow \overline{x}\ i'\,]^{List}.
\end{array}
$$

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 successful parse, consisting of the value parsed and the remaining unparsed input. An empty list denotes a failure to parse the input. We have that

$$
[\,(x, y) \mid x \leftarrow \overline{x},\ y \leftarrow \overline{y}\,]^{Parse} \;=\; \lambda i \to [\,((x, y), i'') \mid (x, i') \leftarrow \overline{x}\ i,\ (y, i'') \leftarrow \overline{y}\ i'\,]^{List}.
$$

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 $\overline{x}$ or $\overline{y}$ 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

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

Thus, $[\,]^{Parse}$ 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:

$$\begin{array}{lcl} (\mathbin{[\!]}) & :: & Parse\ x \to Parse\ x \to Parse\ x \\ \overline{x} \mathbin{[\!]} \overline{y} & = & \lambda i \to (\overline{x}\ i) +\!\!+ (\overline{y}\ i). \end{array}$$

(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:

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

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:

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

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:

$$\begin{array}{lcl} term & :: & Parse\ Term \\ term & = & \quad [\,Var\ x \mid x \leftarrow name\,]^{Parse} \\ & & \mathbin{[\!]}\ [\,Lam\ x\ t \mid () \leftarrow lit\,\text{`('},\ () \leftarrow lit\,\text{`}\lambda\text{'},\ x \leftarrow name,\ () \leftarrow lit\,\text{`}\!\to\!\text{'}, \\ & & \qquad\qquad\quad t \leftarrow term,\ () \leftarrow lit\,\text{`)'}\,]^{Parse} \\ & & \mathbin{[\!]}\ [\,App\ t\ u \mid () \leftarrow lit\,\text{`('},\ t \leftarrow term,\ u \leftarrow term,\ () \leftarrow lit\,\text{`)'}\,]^{Parse} \\ \\ name & :: & Parse\ Name \\ name & = & [\,c \mid c \leftarrow next,\ \text{`a'} \leq c,\ c \leq \text{`z'}\,]^{Parse}. \end{array}$$

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 type *x*, written *Just x*, or an exceptional value, written *Nothing*:

$$\text{data } Maybe\ x \quad = \quad Just\ x \mid Nothing.$$

(The names are due to Spivey [Spi90].) The following operations yield a monad:

$$
\begin{array}{lcl}
map^{Maybe}\, f\, (Just\, x) & = & Just\, (f\, x) \\
map^{Maybe}\, f\, Nothing & = & Nothing \\
\\
unit^{Maybe}\, x & = & Just\, x \\
\\
join^{Maybe}\, (Just\, (Just\, x)) & = & Just\, x \\
join^{Maybe}\, (Just\, Nothing) & = & Nothing \\
join^{Maybe}\, Nothing & = & Nothing.
\end{array}
$$

We have that

$$[\, (x, y) \mid x \leftarrow \overline{x},\ y \leftarrow \overline{y}\,]^{Maybe}$$

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

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

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

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:

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

Observe that

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

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. We have that

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

as an immediate consequence of the monad morphism law. This mapping embodies the common simplification of considering error values and $\perp$ to be identical.

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 mapping

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

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

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

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 Input and output

Fix the input and output of a program to be strings (e.g., the input is a sequence of characters from a keyboard, and the output is a sequence of characters to appear on a screen). The input and output monads are given by:

$$\begin{array}{lcl} \text{type } \mathit{In}\ x & = & \mathit{String} \to (x, \mathit{String}) \\ \text{type } \mathit{Out}\ x & = & (x, \mathit{String} \to \mathit{String}). \end{array}$$

The input monad is a function from a string (the input to the program) and to a pair of a value and a string (the input to the rest of the program). The output monad is a pair of a value and a function from a string (the output of the rest of the program) to a string (the output of the program).

The input monad is identical to the monad of state transformers, fixing the state to be a string; and the operations *map*, *unit*, and *join* are identical to those in the state-transformer monad. Two useful operations in the input monad are

$$\begin{array}{lcl} \mathit{eof} & :: & \mathit{In}\ \mathit{Bool} \\ \mathit{eof} & = & \lambda i \to (\mathit{null}\ i, i) \\ \\ \mathit{read} & :: & \mathit{In}\ \mathit{Char} \\ \mathit{read} & = & \lambda i \to (\mathit{head}\ i, \mathit{tail}\ i). \end{array}$$

The first returns true if there is more input to be read, the second reads the next input character.

The output monad is given by

$$\begin{array}{lcl} \mathit{map}^{\mathit{Out}}\ f\ \hat{x} & = & [\,(f\ x, \mathit{ot}) \mid (x, \mathit{ot}) \leftarrow \hat{x}\,]^{\mathit{Id}} \\ \mathit{unit}^{\mathit{Out}}\ x & = & (x, \lambda o \to o) \\ \mathit{join}^{\mathit{Out}}\ \hat{\hat{x}} & = & [\,(x, \mathit{ot} \cdot \mathit{ot}') \mid (\hat{x}, \mathit{ot}) \leftarrow \hat{\hat{x}},\ (x, \mathit{ot}') \leftarrow \hat{x}\,]^{\mathit{Id}}. \end{array}$$

The second component of the pair is an output transformer, which given the output of the rest of the program produces the output of this part. The unit produces no output of its own, so its output transformer is the identity function. The join operation composes two output transformers. A useful operation in the output monad is

$$\begin{array}{lcl} \mathit{write} & :: & \mathit{Char} \to \mathit{Out}\,() \\ \mathit{write}\; c & = & ((), \lambda o \to c : o). \end{array}$$

This adds the character to be written onto the head of the output list.

Alternative definitions of the output monad are possible, but these do not behave as well as the formulation given above. One alternative treats output as a state transformer,

$$\text{type } \mathit{Out}'\, x \;=\; \mathit{String} \to (x, \mathit{String}),$$

taking *map*, *unit*, and *join* as in the state transformer monad. The write operation is now

$$\begin{array}{lcl} \mathit{write} & :: & \mathit{Char} \to \mathit{Out}'\,() \\ \mathit{write}\; c & = & \lambda o \to ((), c : o). \end{array}$$

This formulation is not so good, because it is too strict: output will not appear until the program terminates. Another alternative is

$$\begin{array}{lcl} \text{type } \mathit{Out}''\, x & = & (x, \mathit{String}) \\ \mathit{map}^{\mathit{Out}''}\, f\, \widehat{x} & = & [\,(f\; x, o) \mid (x, o) \leftarrow \widehat{x}\,]^{\mathit{Id}} \\ \mathit{unit}^{\mathit{Out}''}\, x & = & (x, [\,]) \\ \mathit{join}^{\mathit{Out}''}\, \widehat{\widehat{x}} & = & [\,(x, o +\!\!+\, o') \mid (\widehat{x}, o) \leftarrow \widehat{\widehat{x}},\, (x, o') \leftarrow \widehat{x}\,]^{\mathit{Id}} \\ \mathit{write}\; c & = & ((), [c]). \end{array}$$

This formulation is also not so good, because the time to perform the concatenation ($+\!\!+$) operations is quadratic in the size of the output in the worst case.

Finally, the output and input monads can be combined into a single monad:

$$\text{type } \mathit{InOut}\, x \;=\; \mathit{String} \to (x, \mathit{String}, \mathit{String} \to \mathit{String}).$$

Suitable definitions of *map*, *unit*, and *join* are left to the reader. Useful operations on this monad are:

$$\begin{array}{lcl} \mathit{in} & :: & \mathit{In}\, x \to \mathit{InOut}\, x \\ \mathit{in}\, \overline{x} & = & \lambda i \to [\,(x, i', \lambda o \to o) \mid (x, i') \leftarrow \overline{x}\; i\,]^{\mathit{Id}} \\ \\ \mathit{out} & :: & \mathit{Out}\, x \to \mathit{InOut}\, x \\ \mathit{out}\, \widehat{x} & = & \lambda i \to [\,(x, i, \mathit{ot}) \mid (x, \mathit{ot}) \leftarrow \widehat{x}\,]^{\mathit{Id}} \\ \\ \mathit{fun} & :: & \mathit{InOut}\,() \to (\mathit{String} \to \mathit{String}) \\ \mathit{fun}\, \tilde{x} & = & \lambda i \to [\,\mathit{ot}\,[\,] \mid ((), i', \mathit{ot}) \leftarrow \tilde{x}\; i\,]^{\mathit{Id}}. \end{array}$$

The first two are monad morphisms from *In* and *Out* to *InOut*; they take input-only and output-only operations into the input-output monad. The last takes a value into the input-output monad into a function from the input to the output.

7.4 Continuations

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

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

A continuation of type x takes a continuation function $k :: x \to R$, 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 $\lambda k \to k\ x$ that applies the continuation function to the given value. We have that

$$[\, (x, y) \mid x \leftarrow \overline{x},\ y \leftarrow \overline{y}\,]^{Cont} \;=\; \lambda k \to \overline{x}\, (\lambda x \to \overline{y}\, (\lambda y \to k\, (x, y))).$$

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

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

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

$$(\mathit{call/cc}\ (\mathit{lambda}\ (\mathit{esc})(/\ x\ (\mathit{if}\ (=\ \ y\ \mathit{0})\ (\mathit{esc}\ \mathit{42})\ y))))$$

translates to the equivalent program

$$callcc\, (\lambda esc \to [\, x/z \mid z \leftarrow \text{if } y = \mathit{0} \text{ then } esc\ \mathit{42} \text{ else } [\, y\,]^{Cont}\,]^{Cont}).$$

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 if y is zero.

8 Translation

In Section 4, we saw that a function of type $U \to V$ in an impure functional language that manipulates state corresponds to a function of type $U \to ST\ V$ in a pure functional language. 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 λ-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:

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

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:

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

In the following, we usually give the case for $(\mathit{fst}\ t)$ but omit that for $(\mathit{snd}\ t)$, since the two are nearly identical. 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, and can be seen on the left hand sides of Figures 8 and 10.

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,

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

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 $U \to V$ in the (impure) source language is translated to a function of type $U \to M\ V$ in the (pure) target language. Each of the rules for translating terms has a straightforward computational reading. For example, the rule for applications,

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

can be read "to apply t to u, first evaluate t (call the result f), then evaluate u (call the result x), then apply f to x (call the result y) and return y." 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

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

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*. Figure 8 demonstrates that the call-by-value translation preserves well-typings:

a term that is well-typed in the source language translates to one that is well-typed in the target language.

The call-by-name translation of λ-calculus into a monad M is given in Figure 9. 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,

$$(U \to V)^\dagger \;=\; M\ U^\dagger \to M\ V^\dagger,$$

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,

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

can be read "to apply t to u, first evaluate t (call the result f), then apply f 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 *computations*, reflecting that in a call-by-name language the free variables correspond to computations (or closures) that must be evaluated each time they are used. Figure 10 demonstrates that the call-by-name translation preserves well-typings.

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 the call-by-value and call-by-name translations is a pure, non-strict λ-calculus, there is no problem with executing programs translated by either scheme in 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:

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

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

$$((\lambda a \rightarrow a + a)\,(\mathit{1} \sqcup \mathit{2}))$$

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

We will give the semantics of this language by interpreting the λ-calculus in the set monad, as specified in Section 2.3. In what follows we will write $\{\, t \mid q \,\}$ in preference to the more cumbersome $[\, t \mid q \,]^{Set}$.

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:

$$\begin{array}{lcl}
(u \sqcup v)^* & = & u^* \cup v^* \\
n^* & = & \{n\} \\
(u + v)^* & = & \{\, x + y \mid x \leftarrow u^*,\ y \leftarrow v^* \,\}.
\end{array}$$

These rules translate a term of type T in the non-deterministic language into a term of type $Set\ T$ in a pure functional language augmented with set comprehensions. For example, the term above translates to

$$\begin{array}{l}
\{\, y \mid f \leftarrow \{\, (\lambda a \rightarrow \{\, x' + y' \mid x' \leftarrow \{a\},\ y' \leftarrow \{a\} \,\}) \,\}, \\
\qquad x \leftarrow \{\mathit{1}\} \cup \{\mathit{2}\}, \\
\qquad y \leftarrow (f\ x) \,\}
\end{array}$$

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

The call-by-name translation of the same language is provided by the rules in Figure 9. The rules for $(u \sqcup v)$, n, and $(u + v)$ are the same as the call-by-value rules, replacing $(-)^*$ with $(-)^\dagger$. Now the same term translates to

$$\begin{array}{l}
\{\, y \mid f \leftarrow \{\, (\lambda a \rightarrow \{\, x' + y' \mid x' \leftarrow a,\ y' \leftarrow a \,\}) \,\}, \\
\qquad y \leftarrow f\,(\{\mathit{1}\} \cup \{\mathit{2}\}) \,\}
\end{array}$$

which has the value $\{\mathit{2}, \mathit{3}, \mathit{4}\}$, 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 all programs 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 7.4. Applying straightforward calculation to simplify the *Cont*-comprehensions yields the translation scheme given in Figure 11, which is simply the continuation-passing style transformation beloved by many theorists and compiler writers [Plo75, AJ89].

Each of the rules retains its straightforward operational reading. For example, the rule for applications,

$$(t\ u)^* \;=\; \lambda k \to t^* \, (\lambda f \to u^* \, (\lambda x \to f\ x\ k)),$$

can still be read "to apply t to u, first evaluate t (call the result f), then evaluate u (call the result x), then apply f to x (call the result y) and return y."

A similar calculation on the other translation scheme yields a call-by-name version of continuation-passing style. This is less well known, but can be found in [Rey74, Plo75].

Acknowledgements

I thank Eugenio Moggi 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, Stephen Bevan, Olivier Danvy, Kevin Hammond, John Hughes, Karsten Kehler Holst, Michael Johnson, Austin Melton, Nikhil, Simon Peyton Jones, Andy Pitts, Andre Scedrov, Carolyn Talcott, Phil Trinder, attenders of the 1989 Glasgow Summer School on Category Theory and Constructive Logic, and an anonymous referee.

References

[AJ89] A. Appel and T. Jim, Contiuation-passing, closure-passing style. In *16'th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.

[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. In K. Davis and J. Hughes, editors, *Functional Programming, Glasgow 1989* (Glasgow workshop, Fraserburgh, August 1989), Workshops in Computing, Springer Verlag, 1990.

[HPW91] P. Hudak, S. Peyton Jones and P. Wadler, editors, *Report on the Programming Language Haskell: Version 1.1.* Technical report, Yale University and Glasgow University, August 1991.

[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.

[Plo75] G. Plotkin, Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[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.

[Spi90] M. Spivey, A functional theory of exceptions. *Science of Computer Programming*, **14**(1):25–42, June 1990.

[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 M. Broy and C. Jones, editors, *Programming Concepts and Methods* (IFIP Working Conference, Sea of Gallilee, Israel, April 1990), North Holland, 1990.

[Wad91] P. Wadler, Is there a use for linear logic? In *Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, ACM, New Haven, Connecticut, June 1991.

```
datatype Term  =  Var of Name | Lam of Name * Term | App of Term * Term;

fun rename t  =  let  val  N                      =  ref 0;

                      fun  newname ()             =  let  val  n = !N;
                                                          val  () = (N := n + 1);
                                                     in   mkname n
                                                     end;

                      fun  renamer (Var x)        =  Var x
                      |    renamer (Lam (x, t))   =  let  val  x' = newname ();
                                                     in   Lam (x', subst x' x (renamer t))
                                                     end
                      |    renamer (App (t, u))   =  App (renamer t, renamer u);

                 in   renamer t
                 end;
```

Figure 1: Renaming in an 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  =  let  (x', n') = newname n
                             (t', n'') = renamer t n'
                        in   (Lam x' (subst x' x t'), n'')
renamer (App t u) n  =  let  (t', n') = renamer t n
                             (u', n'') = renamer u n'
                        in   (App t' u', n'')

rename               ::  Term -> Term
rename t             =  let (t', n') = renamer t 0 in t'
```

Figure 2: Renaming in a pure functional language (Haskell)

$$
\begin{array}{lcl}
\text{data } \mathit{Term} & = & \mathit{Var\ Name} \mid \mathit{Lam\ Name\ Term} \mid \mathit{App\ Term\ Term} \\[1ex]
\mathit{newname} & :: & \mathit{ST\ Name} \\
\mathit{newname} & = & [\, \mathit{mkname}\ n \mid n \leftarrow \mathit{fetch},\ () \leftarrow \mathit{assign}\,(n+1)\,]^{ST} \\[1ex]
\mathit{renamer} & :: & \mathit{Term} \rightarrow \mathit{ST\ Term} \\
\mathit{renamer}\ (\mathit{Var}\ x) & = & [\, \mathit{Var}\ x\,]^{ST} \\
\mathit{renamer}\ (\mathit{Lam}\ x\ t) & = & [\, \mathit{Lam}\ x'\ (\mathit{subst}\ x'\ x\ t')) \mid x' \leftarrow \mathit{newname},\ t' \leftarrow \mathit{renamer}\ t\,]^{ST} \\
\mathit{renamer}\ (\mathit{App}\ t\ u) & = & [\, \mathit{App}\ t'\ u' \mid t' \leftarrow \mathit{renamer}\ t,\ u' \leftarrow \mathit{renamer}\ u\,]^{ST} \\[1ex]
\mathit{rename} & :: & \mathit{Term} \rightarrow \mathit{Term} \\
\mathit{rename}\ t & = & \mathit{init}\ 0\ (\mathit{renamer}\ t)
\end{array}
$$

Figure 3: Renaming with the monad of state transformers

$$
\begin{array}{lcl}
\mathit{exp} & :: & \mathit{Exp} \rightarrow \mathit{Arr} \rightarrow \mathit{Val} \\
\mathit{exp}\ (\mathit{Var}\ i)\ a & = & \mathit{index}\ i\ a \\
\mathit{exp}\ (\mathit{Const}\ v)\ a & = & v \\
\mathit{exp}\ (\mathit{Plus}\ e_1\ e_2)\ a & = & \mathit{exp}\ e_1\ a + \mathit{exp}\ e_2\ a \\[1ex]
\mathit{com} & :: & \mathit{Com} \rightarrow \mathit{Arr} \rightarrow \mathit{Arr} \\
\mathit{com}\ (\mathit{Asgn}\ i\ e)\ a & = & \mathit{update}\ i\ (\mathit{exp}\ e\ a)\ a \\
\mathit{com}\ (\mathit{Seq}\ c_1\ c_2)\ a & = & \mathit{com}\ c_2\ (\mathit{com}\ c_1\ a) \\
\mathit{com}\ (\mathit{If}\ e\ c_1\ c_2)\ a & = & \text{if } \mathit{exp}\ e\ a = 0 \text{ then } \mathit{com}\ c_1\ a \text{ else } \mathit{com}\ c_2\ a \\[1ex]
\mathit{prog} & :: & \mathit{Prog} \rightarrow \mathit{Val} \\
\mathit{prog}\ (\mathit{Prog}\ c\ e) & = & \mathit{exp}\ e\ (\mathit{com}\ c\ (\mathit{newarray}\ 0))
\end{array}
$$

Figure 4: Interpreter in a pure functional language

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

Figure 5: Interpreter with state transformers

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

Figure 6: Interpreter with state transformers and readers

Types

$$
\begin{array}{lcl}
K^* & = & K \\
(U \to V)^* & = & (U^* \to M\ V^*) \\
(U, V)^* & = & (U^*, V^*)
\end{array}
$$

Terms

$$
\begin{array}{lcl}
x^* & = & [\, x\,]^M \\
(\lambda x \to v)^* & = & [\, (\lambda x \to v^*)\,]^M \\
(t\ u)^* & = & [\, y \mid f \leftarrow t^*,\ x \leftarrow u^*,\ y \leftarrow (f\ x)\,]^M \\
(u, v)^* & = & [\, (x, y) \mid x \leftarrow u^*,\ y \leftarrow v^*\,]^M \\
(\mathit{fst}\ t)^* & = & [\, (\mathit{fst}\ z) \mid z \leftarrow t^*\,]^M
\end{array}
$$

Assumptions

$$
(x_1 :: T_1, \ldots, x_n :: T_n)^* \ = \ x_1 :: T_1^*, \ldots, x_n :: T_n^*
$$

Typings

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

Figure 7: Call-by-value translation.

$$
\begin{array}{lcl}
\hline
(A,\ x :: T \vdash x :: T)^* & = & A^*,\ x :: T^* \vdash [\, x\,]^M :: M\ T^* \\[2ex]
(A,\ x :: U \vdash v :: V)^* & = & A^*,\ x :: U^* \vdash v^* :: M\ V^* \\ \hline
(A \vdash (\lambda x \to v) :: (U \to V))^* & = & A^* \vdash [\, (\lambda x \to v^*)\,]^M :: M\ (U^* \to M\ V^*) \\[2ex]
(A \vdash t :: (U \to V))^* & = & A^* \vdash t^* :: M\ (U^* \to M\ V^*) \\
(A \vdash u :: U)^* & = & A^* \vdash u^* :: M\ U^* \\ \hline
(A \vdash (t\ u) :: V)^* & = & A^* \vdash [\, y \mid f \leftarrow t^*,\ x \leftarrow u^*,\ y \leftarrow (f\ x)\,]^M :: M\ V^* \\[2ex]
(A \vdash u :: U)^* & = & A^* \vdash u^* :: M\ U^* \\
(A \vdash v :: V)^* & = & A^* \vdash v^* :: M\ V^* \\ \hline
(A \vdash (u, v) :: (U, V))^* & = & A^* \vdash [\, (x, y) \mid x \leftarrow u^*,\ y \leftarrow v^*\,]^M :: M\ (U^*, V^*) \\[2ex]
(A \vdash t :: (U, V))^* & = & A^* \vdash t^* :: M\ (U^*, V^*) \\ \hline
(A \vdash (\mathit{fst}\ t) :: U)^* & = & A^* \vdash [\, (\mathit{fst}\ z) \mid z \leftarrow t^*\,]^M :: M\ U^*
\end{array}
$$

Figure 8: The call-by-value translation preserves well-typing.

Types

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

Terms

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

Assumptions

$$
(x_1 :: T_1, \ldots, x_n :: T_n)^\dagger \;=\; x_1 :: M\ T_1^\dagger, \ldots, x_n :: M\ T_n^\dagger
$$

Typings

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

Figure 9: Call-by-name translation.

$$
\frac{}{(A,\ x :: T \vdash x :: T)^\dagger} \;=\; \frac{}{A^\dagger,\ x :: M\ T^\dagger \vdash x :: M\ T^\dagger}
$$

$$
\frac{(A,\ x :: U \vdash v :: V)^\dagger}{(A \vdash (\lambda x \to v) :: (U \to V))^\dagger} \;=\; \frac{A^\dagger,\ x :: M\ U^\dagger \vdash v^\dagger :: M\ V^\dagger}{A^\dagger \vdash [\,(\lambda x \to v^\dagger)\,]^M :: M\,(M\ U^\dagger \to M\ V^\dagger)}
$$

$$
\frac{(A \vdash t :: (U \to V))^\dagger \quad (A \vdash u :: U)^\dagger}{(A \vdash (t\ u) :: V)^\dagger} \;=\; \frac{A^\dagger \vdash t^\dagger :: M\,(M\ U^\dagger \to M\ V^\dagger) \quad A^\dagger \vdash u^\dagger :: M\ U^\dagger}{A^\dagger \vdash [\,y \mid f \leftarrow t^\dagger,\ y \leftarrow (f\ u^\dagger)\,]^M :: M\ V^\dagger}
$$

$$
\frac{(A \vdash u :: U)^\dagger \quad (A \vdash v :: V)^\dagger}{(A \vdash (u, v) :: (U, V))^\dagger} \;=\; \frac{A^\dagger \vdash u^\dagger :: M\ U^\dagger \quad A^\dagger \vdash v^\dagger :: M\ V^\dagger}{A^\dagger \vdash [\,(u^\dagger, v^\dagger)\,]^M :: M\,(M\ U^\dagger, M\ V^\dagger)}
$$

$$
\frac{(A \vdash t :: (U, V))^\dagger}{(A \vdash (\mathit{fst}\ t) :: U)^\dagger} \;=\; \frac{A^\dagger \vdash t^\dagger :: M\,(M\ U^\dagger, M\ V^\dagger)}{A^\dagger \vdash [\,x \mid z \leftarrow t^\dagger,\ x \leftarrow (\mathit{fst}\ z)\,]^M :: M\ U^\dagger}
$$

Figure 10: The call-by-name translation preserves well-typing.

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

Figure 11: Call-by-value continuation-passing style transformation

$$
\begin{aligned}
x^\dagger &= x \\
(\lambda x \to v)^\dagger &= \lambda k \to k\,(\lambda x \to v^\dagger) \\
(t\ u)^\dagger &= \lambda k \to t^\dagger\,(\lambda f \to f\ u^\dagger\ k) \\
(u, v)^\dagger &= \lambda k \to k\,(u^\dagger, v^\dagger) \\
(\mathit{fst}\ t)^\dagger &= \lambda k \to t^\dagger\,(\lambda z \to \mathit{fst}\ z\ k)
\end{aligned}
$$

Figure 12: Call-by-name continuation-passing style transformation