LISP AND SYMBOLIC COMPUTATION: An International Journal, 7, 39–56, 1993

Monads and composable continuations

PHILIP WADLER (*wadler@dcs.glasgow.ac.uk*)
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, Scotland

Keywords: Monads, Continuations, Continuation-passing style, Types

Abstract. Moggi's use of monads to factor semantics is used to model the composable continuations of Danvy and Filinski. This yields some insights into the type systems proposed by Murthy and by Danvy and Filinski. Interestingly, modelling some aspects of composable continuations requires a structure that is almost, but not quite, a monad.

1. Introduction

Continuation-passing style was introduced to model one feature of programming languages – the jump – and to explicate the execution order of programs [14, 12]. Recently, Moggi has shown how monads, a notion from category theory, generalise the continuation-passing style transformation [9]. Monads can model a wide variety of features, including continuations, state, exceptions, input-output, non-determinism, and parallellism. Monads have also been applied both as a way of structuring functional programs [16, 17] and as a way of introducing new features into functional languages [11].

It begins to seem as if *any* feature of a programming language can be modelled by a monad. Let's take the name 'Moggi's Hypothesis' as a convenient label for this conjecture, although Moggi himself has never made such a rash claim. As we lack a uniform theory of programming language features, it is difficult to see how to verify such a hypothesis. However, it is easy to envision that it might be falsified by a counterexample.

Composable continuations appear to be provide such a counterexample. Usual continuations support the 'escape' operation, which is similar to Landin's J operator, Reynolds's escape operator, or 'call/cc' as found in Scheme [13] or SML/NJ [5]. This increases the programmer's power of expression greatly, though the resulting programs are sometimes rather devious. Composable continuations, as devised by Danvy and Filinski [2, 3, 4], support additional operations 'shift' and 'reset'. These even further increase the programmer's power of expression, and can result in programs of mind-boggling deviousness. (The operations 'shift' and 'reset' are similar to, but not the same as, the operations 'control' and 'prompt' of Felleisen [6].) Two different type systems for composable continuations have been devised, one by Danvy and Filinski [2], and the other by Murthy [10].

The purpose of this note is to explore the utility of monads for modelling composable continuations. A succession of models based on monads will be presented. One of these will correspond to the type system of Murthy, and another to the type system of Danvy and Filinski. The monad approach succeeds in providing a simple way to understand these type systems. On the other hand, some of the models will not quite be monads: one will have types *more general* than a monad, while another will have types *less general* than a monad. So Moggi's Hypothesis appears to be violated. However, the violation is fairly mild: all the models have the same basic structure as a monad, and satisfy the three monad laws; and monads provide a useful framework for parameterising the definition of two-level continuations.

Kieburtz, Agapiev, and Hook have also examined the use of monads to model composable continuations [8]. Their model also turns out to be not quite a monad, though for a very different reason than the model presented here. In particular, their model has the right type for a monad, but fails to satisfy one of the three monad laws. However, their model seems to contain a spurious complication. Removing the complication appears to yield the model in this paper, which does satisfy all the monad laws.

As a sidelight, the work presented here demonstrates the utility of a functional language as a 'power tool' for performing experiments in theoretical computer science. At various points we will need to know the most general type of a given form that can be assigned to a lambda expression. This was easily computed using an implementation of a functional language that encorporates the Hindley-Milner type reconstruction algorithm [7, 1]. The implementation used was Gofer, a dialect of Haskell implemented by Mark Jones; but any language based on Hindley-Milner types would be suitable.

This paper is aimed at readers familiar with monads and composable continuations, but for completeness summaries of both will be presented. The remainder of this paper is organised as follows. Section 2 reviews monads. Section 3 reviews composable continuations, and shows how they can be modeled with monads (and in one case, with something more than a monad). Section 4 adapts this model to two-level continuations (and along the way spots something that is less than a monad). Section 5 critiques the work of Kieburtz, Agapiev, and Hook. Section 6 concludes.

2. Monads

For our purposes, a *monad* consists of a type constructor M and two operations.

$$\begin{array}{lll} unit & :: & a \to M\ a \\ (\star) & :: & M\ a \to (a \to M\ b) \to M\ b \end{array}$$

(We write '::' for 'has type'.) If you are a functional programmer, think of M as a type constructor in a functional language, and *unit* and $\star$ as polymorphic functions. If you are a domain theorist, think of M as an operation on domains, and *unit* and $\star$ as operations parameterised on the domains a and b.

Roughly speaking, type $M\ a$ represents a computation that yields a value of type a. The purpose of *unit* is to take values into computations. If $v :: a$ is a value, then $unit\ v :: M\ a$ represents the computation that does nothing except yield the value v. The purpose of $\star$ is to combine two computations, where the second computation may depend on a value yielded by the first. If $m :: M\ a$ is a computation and $k :: a \to M\ b$ is a function from values to computations, then $m \star k :: M\ b$ represents the computation that performs computation m, applies k to the value yielded by the computation, and then performs the computation that results.

The monad operations must satisfy three laws.

$$\begin{array}{lrcl} \text{(left unit)} & unit\ v \star (\lambda w.\, k\ w) & = & k\ v \\ \text{(right unit)} & m \star (\lambda v.\, unit\ v) & = & m \\ \text{(associative)} & (m \star (\lambda v.\, k\ v)) \star (\lambda w.\, h\ w) & = & m \star (\lambda v.\, (k\ v \star (\lambda w.\, h\ w))) \end{array}$$

The laws may be simplified by rewriting $(\lambda w.\, k\ w)$ as k, and so on. The form shown here was chosen because in practice one tends to write $\star$ followed by a lambda abstraction.

2.1. The translation

There is a standard call-by-value translation of lambda calculus into a monad.

$$\begin{array}{lcl} [\![x]\!]\rho & = & unit\ (\rho\ x) \\ [\![\lambda x.\, e]\!]\rho & = & unit\ (\lambda v.\, [\![e]\!]\rho[v/x]) \\ [\![d\ e]\!]\rho & = & [\![d]\!]\rho \star (\lambda k.\, [\![e]\!]\rho \star (\lambda v.\, k\ v)) \\ [\![n]\!]\rho & = & unit\ n \\ [\![d + e]\!]\rho & = & [\![d]\!]\rho \star (\lambda v.\, [\![e]\!]\rho \star (\lambda w.\, unit\ (v + w))) \\ [\![\text{if } c \text{ then } d \text{ else } e]\!]\rho & = & [\![c]\!]\rho \star (\lambda v.\, \text{if } v \text{ then } [\![d]\!]\rho \text{ else } [\![e]\!]\rho) \\ [\![\text{let } x = d \text{ in } e]\!]\rho & = & [\![d]\!]\rho \star (\lambda v.\, [\![e]\!]\rho[v/x]) \\ [\![\text{letrec } f = (\lambda x.\, d) \text{ in } e]\!]\rho & = & \text{letrec } k = (\lambda v.\, [\![d]\!]\rho[k/f, v/x]) \\ & & \text{in } [\![e]\!]\rho[k/f] \end{array}$$

Here c, d, e range over terms in the source language, x, f range over variables in the source language, n ranges over constants, v, w, k range over values (or variables in the target language), and ρ ranges over environments mapping source variables to values (or to target variables). Since this is a call-by-value translation, 'letrec' expressions are valid only if the recursive variable is bound to a function definition. A representative sampling of expressions has been given; it is straightforward to add others.

This translation can be viewed in two ways. If the right-hand side is taken as meta-syntax, with v, w, k as values, then the result is an *interpreter*, or a denotational semantics. If the right-hand side is taken as a target language, with v, w, k as variables in the target language, then the result is a *compiler*. The source and target languages are both lambda calculus, but the target language contains *unit* and $\star$ as additional constructs (or as predefined constants, if you prefer).

2.2. Types

If viewed as a compiler, the translation takes simply-typed lambda calculus into simply-typed lambda calculus. The translation on types is as follows.

$$\begin{array}{lcl} [\![I]\!] & = & I \\ [\![a \to b]\!] & = & [\![a]\!] \to M\,[\![b]\!] \end{array}$$

Here I ranges over base types such as *Int* or *Bool*, and a, b range over types. As with terms, the source and target languages contain the same types, except the target language contains M as an additional type constructor. The typed source term

$$x_1 :: a_1, \ldots, x_n :: a_n \vdash e :: b$$

translates into the typed target term

$$v_1 :: [\![a_1]\!], \ldots, v_n :: [\![a_n]\!] \vdash [\![e]\!]\rho :: M\,[\![b]\!]$$

where $\rho = [v_1/x_1, \ldots, v_n/x_n]$.

There is a problem if the source language includes 'let' polymorphism, as found in the Hindley-Milner type system: translating 'let' into $\star$ loses polymorphism. Thus we will limit our attention, as do Danvy and Filinski and Murthy, to a source language without 'let' polymorphism. This is a serious restriction; an alternative approach might be based on stronger type systems such as polymorphic lambda calculus [15].

2.3. For categorists only

Categorists will recognize M as a monad with *unit* as its unit and $\star$ as its Kleisli star. The free use of lambda calculus means that we are working in a cartesian closed category. It is assumed that $\star$ is itself represented as an arrow in that category, rather than just an operator on arrows. That is, we assume for each object a and b an arrow

$$(a \Rightarrow M\ b) \longrightarrow (M\ a \Rightarrow M\ b)$$

where $\longrightarrow$ denotes an arrow in the category, and $\Rightarrow$ denotes exponentiation. This is stronger than the assumptions made by Moggi. When we use 'letrec', we also assume for each object a and b an arrow

$$((a \Rightarrow M\ b) \Rightarrow (a \Rightarrow M\ b)) \longrightarrow (a \Rightarrow M\ b)$$

to model fixpoints.

3. Continuations

The usual continuation passing style arises as a special case of the monad translation. We define our monad as follows.

$$\begin{array}{lcl}
\text{type}\ M\ a & = & (a \to O) \to O \\[1ex]
\mathit{unit} & :: & a \to M\ a \\
(\star) & :: & M\ a \to (a \to M\ b) \to M\ b \\
\mathit{eval} & :: & M\ O \to O \\[1ex]
\mathit{unit}\ v & = & \lambda c.\ c\ v \\
m \star k & = & \lambda c.\ m\ (\lambda v.\ k\ v\ c) \\
\mathit{eval}\ m & = & m\ \mathit{id}
\end{array}$$

Here O is the type of answers, and $\mathit{id} = (\lambda v.\ v)$ is the identity function. We have added one extra operation, *eval*, to the monad, which can be used to extract the answer from a 'top-level' computation.

Substituting these defintions of *unit* and $\star$ into the call-by-value monad translation and simplifying yields the usual call-by-value continuation-passing style translation. A pleasant property of this translation is that a source program is always given the same call-by-value semantics, regardless of whether the target program is given a call-by-value or call-by-name semantics.

Monads are in a sense just an abstraction of continuation passing style, and the second argument to $\star$ is very similar to a continuation. In a sense, we have continuations at two levels: at the meta-level, we have the continuation $k :: a \to M\ b$ and at the object level we have the continuation $c :: a \to O$. (Section 4 will introduce yet another level of continuation.)

3.1. Escape

We now extend our source language with an 'escape' operation, similar to Landin's J operator, Reynolds' escape operator, or 'call/cc' as found in Scheme or SML/NJ. First, we extend the translation.

$$[\![\text{escape } f.\, e]\!]\rho \quad = \quad \mathit{escape}\,(\lambda k.\, [\![e]\!]\rho[k/f])$$

Second, we add a corresponding new operation to the monad.

$$\begin{array}{lcl} \mathit{escape} & :: & ((a \to M\ b) \to M\ a) \to M\ a \\ \mathit{escape}\, h & = & \lambda c.\, h\,(\lambda v.\, \lambda c'.\, c\ v)\ c \end{array}$$

The 'escape' operation binds the specified variable to a function that, if called, returns its argument to the context surrounding the 'escape'.

For example, the term below has the value 101.

$$1 + (\text{escape } f.\,(10 + (f\ 100))) \quad = \quad 1 + 100$$

Here f is bound to a function that passes its argument to the context surrounding the 'escape'.

The type of *escape* says something about what type the 'escape' construct might have in a typed programming language. Using the type translation of Section 2.2, the type of 'escape' is the translation of the type $((a \to b) \to a) \to a$, so one might expect an 'escape' construct to have the latter type. And indeed, the type of 'escape' in SML/NJ is very similar to this. However, it is not identical, as extra cleverness is required to get maximum polymorphism; see the discussion by Duba, Harper, and MacQueen [5].

3.2. Shift and reset

We further extend our source language with the operations 'shift' and 'reset' defined by Danvy and Filinski. Again, first we extend the translation.

$$\begin{array}{lcl} [\![\text{shift } f.\, e]\!]\rho & = & \mathit{shift}\,(\lambda k.\, [\![e]\!]\rho[k/f]) \\ [\![\text{reset } e]\!]\rho & = & \mathit{reset}\,([\![e]\!]\rho) \end{array}$$

Second, we add corresponding operations to the monad.

$$\begin{array}{lcl} \mathit{shift} & :: & ((a \to M\ O) \to M\ O) \to M\ a \\ \mathit{reset} & :: & M\ O \to M\ O \\ \\ \mathit{shift}\, h & = & \lambda c.\, h\,(\lambda v.\, \lambda c'.\, c'\,(c\ v))\ \mathit{id} \\ \mathit{reset}\, m & = & \lambda c.\, c\,(m\ \mathit{id}) \end{array}$$

The 'shift' operation binds the specified variable to the context between the 'shift' and the nearest (dynamically) enclosing 'reset' (or 'shift'); the body of the 'shift' returns its value to the nearest enclosing 'reset'. Here are three examples.

Example 1. The three terms below have the values 121, 101, and 1121, respectively.

$$\begin{array}{lcl} 1 + (\text{reset } (10 + (\text{shift } f.\, (f\,(f\,100))))) & = & 1 + (10 + (10 + 100)) \\ 1 + (\text{reset } (10 + (\text{shift } f.\, 100))) & = & 1 + 100 \\ 1 + (\text{reset } (10 + (\text{shift } f.\, ((f\,100) + (f\,1000))))) & & \\ \quad = \; 1 + ((10 + 100) + (10 + 1000)) & & \end{array}$$

In each case, f is bound to a function that behaves as the context between the 'shift' and the enclosing 'reset'; that is, f adds 10 to its argument.

Example 2. Here is a very odd way of reversing a list.

$$\begin{array}{l} \text{letrec } \mathit{perverse} = (\lambda l.\, \text{if } \mathit{null}\ l \\ \qquad\qquad\qquad\qquad \text{then } [\,] \\ \qquad\qquad\qquad\qquad \text{else } (\text{shift } f.\, \mathit{head}\ l : f\ (\mathit{perverse}\ (\mathit{tail}\ l)))) \\ \text{in } (\text{reset } (\mathit{perverse}\,[1,2,3])) \end{array}$$

(Here *head*, *tail*, *null*, $[\,]$, and : (cons) are the usual operations on lists.) This returns $[3,2,1]$. For an explanation, consult Danvy and Filinski [2].

Example 3. Here is an even stranger program.

$$\begin{array}{l} \text{let } g = (\text{reset } (\text{if } (\text{shift } f.\, f) \text{ then } 2 \text{ else } 3)) \\ \text{in } (g\ \mathit{True}) + (g\ \mathit{False}) \end{array}$$

Here f (and hence g) is bound to the function that returns 2 if passed *True*, and 3 if passed *False*, hence the value of the given term is 5. Application of a similar idea to implement backtracking has been explored by Danvy and Filinski [3].

Despite the worryingly convoluted nature of the examples, it is this ability to express and encapsulate features such as backtracking that (may) make composable continuations worthy of study.

3.3. Laws of escape and shift

Various properties are satisfied by the 'escape' and 'shift' operators.

$$\begin{array}{rcll} [\![\text{escape } f.\, e]\!] & = & [\![e]\!], & f \text{ not free in } e \\ [\![\text{shift } f.\, f\ e]\!] & = & [\![e]\!], & f \text{ not free in } e \\ [\![\text{escape } f.\, e]\!] & = & [\![\text{shift } f'.\, f'\ (\text{let } f = (\lambda x.\, \text{shift } f''.\, f'\ x) \text{ in } e)]\!], & \\ & & & f' \text{ not free in } e \end{array}$$

The third equation acts as a definition of 'escape' in terms of 'shift'.

The above equations in the source language are equivalent to the following in the target language.

$$\begin{array}{rcl} \mathit{escape}\,(\lambda k.\, m) & = & m \\ \mathit{shift}\,(\lambda k.\, m \star k) & = & m \\ \mathit{escape}\, h & = & \mathit{shift}\,(\lambda k'.\, h\,(\lambda x.\, \mathit{shift}\,(\lambda k''.\, k'\, x)) \star k') \end{array}$$

Each of these can be shown straightforwardly from the definitions of *escape*, *shift*, and $\star$ together with the laws of lambda calculus.

3.4. What about the types?

Consider again the types given for *shift* and *reset*.

$$\begin{array}{lcl} \mathit{shift} & :: & ((a \to M\ O) \to M\ O) \to M\ a \\ \mathit{reset} & :: & M\ O \to M\ O \end{array}$$

These are the most general type that can be derived for the given definition of M.

Here the types are less satisfactory than with *escape*. A disturbing number of O's are creeping in. Recall that O is the type of answers returned at the 'top level'. It doesn't seem reasonable that uses of 'shift' and 'reset' be restricted to apply to terms that have the same type as the top-level context.

Nonetheless, these typings are suitable for building an interpreter (or equivalently, a denotational semantics). This works because in an interpreter there is only a single 'value' type (into which all others are embedded) and one takes O to be this type. In short, the interpreter works because it is essentially untyped.

We now look at better types for the monads, which correspond to the type systems proposed by Murthy [10] and Danvy and Filinski [2].

3.5. Murthy types

One way to generalise is to parameterise the monad M by the answer type. Here is the result, giving the most general types than can now be inferred for the previous definitions.

$$
\begin{array}{lcl}
\text{type } M\ o\ a & = & (a \to o) \to o \\
\mathit{unit} & :: & a \to M\ o\ a \\
(\star) & :: & M\ o\ a \to (a \to M\ o\ b) \to M\ o\ b \\
\mathit{eval} & :: & M\ o\ o \to o \\
\mathit{escape} & :: & ((a \to M\ o\ b) \to M\ o\ a) \to M\ o\ a \\
\mathit{shift} & :: & ((a \to M\ o\ p) \to M\ p\ p) \to M\ p\ a \\
\mathit{reset} & :: & M\ p\ p \to M\ o\ p
\end{array}
$$

Voila! The resulting types correspond to those used by Murthy [10, Section 4.2]. His type system in fact applies to a more general language: each 'shift' and 'reset' operation has a level n, where $0 < n < m$, and there are $m - 1$ possible levels. The system here arises in the special case where $m = 2$ and $n = 1$. For that case, his rules reduce to roughly the following, with some simplification and change of notation.

$$
\frac{\Gamma, x : a \to K_{s,o}[p] \vdash e : K_{s,p}[p]}{\Gamma \vdash \text{shift } x.\, e : K_{s,p}[a]}
\qquad\qquad
\frac{\Gamma \vdash e : K_{s,p}[p]}{\Gamma \vdash \text{reset } e : K_{s,o}[p]}
$$

Murthy's $K_{s,o}[a]$ corresponds to our type $M\ o\ a$, and his typing rules correspond directly to the typings given above for *shift* and *reset*. (We will see in Section 4.1 what it is that his s corresponds to.)

Examples 1 and 2 are well-typed in this system, but Example 3 is not.

3.6. Danvy and Filinski types

An even more general type system results if the monad is given *two* type parameters, replacing $(a \to o) \to o$ by the more general $(a \to p) \to o$. This makes sense, in that there is no reason why the type p returned by a composable continuation need be the same as the type o returned by the entire computation. Here is the new result, again inferring the most general types consistent with the definitions.

$$
\begin{array}{lcl}
\text{type } M\ o\ p\ a & = & (a \to p) \to o \\
\mathit{unit} & :: & a \to M\ o\ o\ a \\
(\star) & :: & M\ o\ p\ a \to (a \to M\ p\ q\ b) \to M\ o\ q\ b \\
\mathit{eval} & :: & M\ o\ p\ p \to o \\
\mathit{escape} & :: & ((a \to M\ p\ q\ b) \to M\ o\ p\ a) \to M\ o\ p\ a \\
\mathit{shift} & :: & ((a \to M\ o\ o\ p) \to M\ q\ r\ r) \to M\ q\ p\ a \\
\mathit{reset} & :: & M\ p\ q\ q \to M\ o\ o\ p
\end{array}
$$

Voila again! The new types correspond to those used by Danvy and Filinski [2]. That paper uses the notation $\Gamma, p \vdash e : a, o$ where Γ is an environment mapping variables to types, e is an expression, a is the type of e, and p and o are the 'old' and 'new' continuation types. Corresponding to this, we would say that the translation of e has type $M\ o\ p\ a$. That paper also writes the type of functions in the form $a/p \to b/o$, where a is the argument type, b is the result type, and again p and o are the 'old' and 'new' continuation types. Corresponding to this, we would say that the translation of the function has type $a \to M\ o\ p\ b$. The typing rules given in that paper can then be seen to correspond to the types of the monad operations given above.

The resulting type scheme is general enough so that Examples 1, 2, and 3 are all well typed.

3.7. It's not a monad!

Our model of Danvy and Filinski's type system is quite satisfactory. But it is *not* a monad. In a monad, $\star$ should have the type

$$(\star) \quad :: \quad M\ a \to (a \to M\ b) \to M\ b.$$

To model Murthy's type system, $\star$ has the type

$$(\star) \quad :: \quad M\ o\ a \to (a \to M\ o\ b) \to M\ o\ b,$$

and this still matches the monad pattern, where M in the former corresponds to $M\ o$ in the latter. But for our second attempt, $\star$ has the type

$$(\star) \quad :: \quad M\ o\ p\ a \to (a \to M\ p\ q\ b) \to M\ o\ q\ b,$$

and this is *too general* to be a monad: the three type constructors $M\ o\ p$, $M\ p\ q$, and $M\ o\ q$ are quite distinct.

So the monad methodology has led to a successful modeling of 'shift' and 'reset', but in the process we had to create something more general than a monad. Note that the generalisation is fairly mild. We still use a type constructor and two operations *unit* and $\star$, and one can verify that the three monad laws are still satisfied. It is just that the type of $\star$ is too general.

4. Two-level continuations

The semantics given for 'shift' and 'reset' allows applications of continuations to be nested, as in the phrase $c'\,(c\,v)$. As a result, one loses the pleasant property that a source program is always assigned the same semantics regardless of whether the target program is taken as call-by-value or call-by-need. This property can be regained by taking the definitions of 'shift' and 'reset' and translating *them* into continuation-passing style.

What is now required is two levels of continuations. The lower level, called L, is identical to the previous monad M.

$$
\begin{array}{lcl}
\text{type } L\,a & = & (a \to O) \to O \\
\\
unit_L & :: & a \to L\,a \\
(\star_L) & :: & L\,a \to (a \to L\,b) \to L\,a \\
eval_L & :: & L\,O \to O \\
\\
unit_L\,v & = & \lambda g.\, g\,v \\
l \star_L c & = & \lambda g.\, l\,(\lambda v.\, c\,v\,g) \\
eval_L\,l & = & l\,id
\end{array}
$$

This uses 'first-level' continuations, $g :: a \to O$.

The upper level is the new monad M. The definition of $M\ a$ has changed in that the old answer type O is replaced by the new answer type $L\,a$. The operations $unit_L$, $\star_L$, and $eval_L$ are used to manipulate values of type $L\,a$.

$$
\begin{array}{lcl}
\text{type } M\ a & = & (a \to L\,a) \to L\,a \\
\\
unit & :: & a \to M\ a \\
(\star) & :: & M\ a \to (a \to M\ a) \to M\ a \\
eval & :: & M\ O \to O \\
escape & :: & ((a \to M\ a) \to M\ a) \to M\ a \\
shift & :: & ((a \to M\ a) \to M\ a) \to M\ a \\
reset & :: & M\ a \to M\ a \\
\\
unit\ v & = & \lambda c.\, c\,v \\
m \star k & = & \lambda c.\, m\,(\lambda v.\, k\,v\,c) \\
eval\ m & = & eval_L\,(m\ unit_L) \\
escape\ h & = & \lambda c.\, h\,(\lambda v.\, \lambda c'.\, c\,v)\ c \\
shift\ h & = & \lambda c.\, h\,(\lambda v.\, \lambda c'.\, c\,v \star_L (\lambda v'.\, c'\,v'))\ unit_L \\
reset\ m & = & \lambda c.\, m\ unit_L \star_L (\lambda v.\, c\,v)
\end{array}
$$

This uses 'second-level' continuations, $c :: a \to L\,a$. The definitions of *unit*, $\star$, and *escape* remain unchanged, while *eval*, *shift*, and *reset* have been rewritten in terms of the continuation-passing operations of the lower level. For instance, the phrase $c'\,(c\ v)$ in the old definition of *shift* is replaced by $c\ v \star_L (\lambda v'.\, c'\,v')$.

One advantage of the monad approach is that the translation need not be changed at all, only the monad definitions. This contrasts with the previous attempts to explain the two-level style, where the two levels of continuations had to be interwoven with the translation scheme.

The type of M was chosen to correspond to that used by Danvy and Filinski, where a is their 'value' type and O is their 'answer' type. Expanding out the definitions of $unit_L$, $\star_L$, and $eval_L$ in the definitions of *eval*, *shift*, and *reset* yields definitions corresponding to those given by Danvy and Filinski.

$$\begin{array}{lcl} eval\, m & = & m\,(\lambda v.\,\lambda g.\,g\,v)\; id \\ shift\, h & = & \lambda c.\,\lambda g.\,h\,(\lambda v.\,\lambda c'.\,\lambda g'.\,c\,v\,(\lambda v'.\,c'\,v'\,g'))\,(\lambda v''.\,\lambda g''.\,g''\,v'')\,g \\ reset\, m & = & \lambda c.\,\lambda g.\,m\,(\lambda v'.\,\lambda g'.\,g'\,v')\,(\lambda v.\,c\,v\,g) \end{array}$$

Different definitions for the monad L would yield different definitions for the monad M. In particular, one might choose L to be the identity monad.

$$\begin{array}{lcl} \text{type } L\,a & = & a \\ & & \\ unit_L\, v & = & v \\ v \star_L c & = & c\, v \\ eval_L\, v & = & v \end{array}$$

Expanding then yields the original definitions of *eval*, *shift*, and *reset* given in Section 3.

The laws of Section 3.3 still hold with the new definitions. Furthermore, they hold regardless of the definition of L chosen, so long as it satisfies the right unit law for monads. An interesting question is whether there are other laws of 'escape' and 'shift' that depend upon the left unit or associative law of L.

The types shown above are the most general that can be inferred for the given definition of M. However the types are highly unsatisfactory in that $M\ a$ appears uniformly everywhere. In particular, recall that from the definition of a monad, the type of $\star$ should be

$$(\star) \quad :: \quad M\ a \to (a \to M\ b) \to M\ b.$$

But the type given above is

$$(\star) \quad :: \quad M\ a \to (a \to M\ a) \to M\ a.$$

So the structure above is *more specific* than a proper monad. Compare this with the situation in Section 3.6, where the structure was *more general* than a proper monad.

For the purposes of building an interpreter, this doesn't matter, as we are only interested in one type anyway (the 'value' type). Nonetheless, that our model does not form a proper monad is somewhat disturbing. Fortunately, it turns out that the various generalisations suggested previously fix this problem.

4.1. Murthy types

Recall that to model Murthy's type system, we used continuations parameterised on the answer type.

$$\text{type } M\ o\ a \quad = \quad (a \to o) \to o$$

The corresponding type for the two-level system is as follows.

$$\begin{array}{lcl} \text{type } L\ o\ a & = & (a \to o) \to o \\ \text{type } M\ s\ o\ a & = & (a \to L\ s\ o) \to L\ s\ o \end{array}$$

With these new type definitions, the two-level monad operations can be given exactly the same types as in Section 3.5 with M everywhere replaced by $M\ s$, except that now $\mathit{eval} :: M\ s\ s\ s \to s$. This explains where the extra parameter s in Murthy's notation comes from: we now have the more precise correspondence between $K_{s,o}[a]$ and $M\ s\ o\ a$.

The overspecialisation has been fixed: the type of $\star$ is now correct for this to be a monad.

4.2. Danvy and Filinski types

To model Danvy and Filinski's system, we used continuations parameterised over two types.

$$\text{type } M\ o\ p\ a \quad = \quad (a \to p) \to o$$

The corresponding type for the two-level system is as follows.

$$\begin{array}{lcl} \text{type } L\ o\ a & = & (a \to o) \to o \\ \text{type } M\ s\ o\ p\ a & = & (a \to L\ s\ p) \to L\ s\ o \end{array}$$

With these new type definitions, the two-level monad operations can be given exactly the same types as in Section 3.6 with M everywhere replaced by $M\ s$, except that now $\mathit{eval} :: M\ s\ p\ p \to s$.

The overspecialisation has been fixed too well: we have gone back to a model that is too general to be a monad.

4.3. Multiple levels

There are more general definitions of 'shift' and 'reset' that work for an arbitrary number of levels. It may be possible to model these by a hierarchy of monads.

$$\begin{array}{lcl} \text{type } M^0\ a & = & a \\ \text{type } M^1\ o_1\ a & = & (a \to M^0\ o_1) \to M^0\ o_1 \\ \text{type } M^2\ o_2\ o_1\ a & = & (a \to M^1\ o_2\ o_1) \to M^1\ o_2\ o_1 \\ & \vdots & \end{array}$$

Here M^0 is the identity monad, M^1 is the previous L, and M^2 is the previous M, in the Murthy type system. In general, the type $M^m\ o_m\ \ldots\ o_1\ a$ corresponds to Murthy's $K_{o_m \ldots o_1}[a]$. It remains to be seen whether the generalised definitions of 'shift' and 'reset', and the corresponding type system, fit into the monad framework.

5. Relation to Kieburtz, Agapiev, and Hook

Kieburtz, Agapiev, and Hook also model composable continuations using something that is almost, but not quite, a monad [8]. In their case, the reason for failure is entirely different. The types are all right, but one of the monad laws fails to hold.

Their model is based on a sequence of three type constructors, each accompanied by appropriate *unit* and $\star$ operations.

$$\begin{array}{lcl} \text{type } T\, a & = & (a \to O) \to O \\ \text{type } S\, a & = & (a \to O) \to a \\ \text{type } R\, a & = & T\,(S\, a) \end{array}$$

The first two of these are monads, but the third fails to satisfy one of the monad laws.

However, on close examination, their model appears to be overly complicated. The 'continuation' passed to computations of type $S\, a$ appears always to be ignored! It is therefore possible to simplify their system greatly, by eliding all the unexamined continuations. The result is that the type $S\, a$ becomes simply the type a, and the type $R\, a$ becomes identical to $T\, a$. The simplified definitions are essentially the same as those given in Section 3 of this paper. The failure to satisfy a monad law seems due to the introduction of a spurious continuation.

A closer comparison between the papers is clouded by two points. First, they make some strong assumptions about the answer type O in order to give *eval* the type $M\ a \to a$. The same effect seems to be achieved here by parameterising on the answer type, and giving *eval* the type $M\ o\ o \to o$. Second, they use a version of 'shift' that differs slightly from Danvy and Filinski's definition, though regretably their paper fails to make that clear.

6. Conclusion

We have succeeded in using monads to model composable continuations. Along the way, we have encountered some counter-examples to what might be called 'Moggi's hypothesis': the conjecture that every programming language feature can be modeled by a monad. One of our models had types too general to be a monad, and another had types too specific. Every concept is better understood by knowing its limits, so it is good to have at last found a place where monads don't (quite) reach.

Nonetheless, it seems fair to count this as a victory for the monad approach. All of our models were obtained by starting from the obvious monad model and modifying it in straightforward ways. This led directly to type systems already reported in the literature, and provided useful insight for understanding and relating these systems. Further, the factorisation of two-level continuations into two levels of monads may prove to be a useful generalisation.

This work was aided by using an implementation of a functional language as a 'power tool' for performing experiments. The various monads were implemented in Haskell, and the various types given given above were derived automatically using the Hindley-Milner algorithm embedded in the Haskell implementation. A simple compiler based on the monad translation was written in Haskell, and used to translate the examples given in Section 3.2 into Haskell. The resulting code was type-checked in conjunction with the various monads, thereby testing the power of the induced type systems.

Several questions remain for future consideration. What, if any, is the categorical nature of the various generalisations of monad discussed? Are there better type systems for composable continuations outside the strictures of the Hindley-Milner system?

One goal of composable continuations with multiple levels was to be able to factor different effects into different levels. Danvy and Filinski claim it is relatively easy to combine different effects uniformly in this way. Monads are also intended to factor effects in a way which eases their combination. However, there is no uniform rule for combining any two monads. This paper has used monads to shine some light on composable continuations. Will composable continuations shed light on the problem of combining monads?

Acknowledgements. Thanks for the Gofer system to Mark Jones, and for their comments to Olivier Danvy, Andrzej Filinski, James Hook, Dick Kieburtz, Chetan Murthy, Carolyn Talcott, and three anonymous and perceptive referees.

References

1. Damas, L. and Milner, R. Principal type-schemes for functional programs. In *9th Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, ACM (January 1982).

2. Danvy, O. and Filinski, A. *A functional abstraction of typed contexts.* Technical Report, Copenhagen University (August 1989).

3. Danvy, O. and Filinski, A. Abstracting control. In *Conference on Lisp and Functional Programming, Nice, France*, ACM (June 1990).

4. Danvy, O. and Filinski, A. Representing control. *Mathematical Structures in Computer Science*, 2, 4 (December 1992).

5. Duba, B., Harper, R., and MacQueen, D. Typing first-class continuations in ml. In *18'th Symposium on Principles of Programming Languages, Orlando, Florida*, ACM (January 1991).

6. Felleisen, M. The theory and practice of first-class prompts. In *15th Symposium on Principles of Programming Languages, San Diego, California*, ACM (January 1988).

7. Hindley, R. The principle type-scheme of an object in combinatory logic. *Trans. Am. Math. Soc.*, 146 (1969) 29–60.

8. Kieburtz, R., Agapiev, B., and Hook, J. Three monads for continuations. In *ACM SIGPLAN Workshop on Continuations* (June 1992). Stanford University Report STAN-CS-92-1426.

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

10. Murthy, C. Control operators, hierarchies, and pseudo-classical type systems. In *Proceedings of the ACM SIGPLAN Workshop on Continuations* (June 1992). Stanford University Report STAN-CS-92-1426.

11. Peyton Jones, S. and Wadler, P. Imperative functional programming. In *20th Symposium on Principles of Programming Languages, Charleston, South Carolina*, ACM (January 1993).

12. Plotkin, G. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1 (1975) 125–159.

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

14. Reynolds, J. C. Definitional interpreters for higher-order programming languages. In *25'th ACM National Conference* (1972) 717–740.

15. Reynolds, J. C. Three approaches to type structure. In *Mathematical Foundations of Software Development*, LNCS 185, Springer-Verlag (1985).

16. Wadler, P. Comprehending monads. *Mathematical Structures in Computer Science*, 2, 4 (December 1992). An earlier version appears in *Conference on Lisp and Functional Programming, Nice, France*, ACM, June 1990.

17. Wadler, P. The essence of functional programming. In *19th Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, ACM (January 1992).