Representing Monads*

Andrzej Filinski

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
andrzej+@cs.cmu.edu

Abstract

We show that any monad whose unit and extension operations are expressible as purely functional terms can be embedded in a call-by-value language with "composable continuations". As part of the development, we extend Meyer and Wand's characterization of the relationship between continuation-passing and direct style to one for continuation-passing vs. general "monadic" style. We further show that the composable-continuations construct can itself be represented using ordinary, non-composable first-class continuations and a single piece of state. Thus, in the presence of two specific computational effects – storage and escapes –

any expressible monadic structure (e.g., nondeterminism as represented by the list monad) can be added as a purely definitional extension, without requiring a reinterpretation of the whole language. The paper includes an implementation of the construction (in Standard ML with some New Jersey extensions) and several examples.

1 Introduction

1.1 Background and overview

Over the last few years, monads have gained considerable acceptance in the lazy functional programming world. Originally proposed by Moggi as a convenient framework for structuring the *semantics* of languages

*This research was sponsored in part by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168. NSF also sponsored this research under grant no. CCR-8922109.
The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1994.

[14, 16], they were quickly popularized by Wadler and others as a technique for structuring functional *programs* [32, 18]. It is not hard to see the reason for this popularity: monads promise access to state, control operators, I/O, etc., while retaining the strong reasoning principles valid for pure functional languages. Briefly, restricting programs to so-called "monadic style" (very similar in spirit and appearance to continuation-passing style) sets up a uniform infrastructure for representing and manipulating computations with effects as first-class objects.

It is somewhat remarkable that monads have had no comparable impact on "impure" functional programming. Perhaps the main reason is that – as clearly observed by Moggi, but perhaps not as widely appreciated in the "purely functional" community – the monadic framework is *already* built into the semantic core of eager functional languages with effects, and need not be expressed explicitly. "Impure" constructs, both lin-

guistic (e.g., updatable state, exceptions, or first-class continuations) and external to the language (I/O, OS interface, etc.), all obey a monadic discipline. The only aspect that would seem missing is the ability for programmers to use their own, application-specific monadic abstractions – such as nondeterminism or parsers [31] – with the same ease and naturality as built-in effects.

Actually, many of the useful monadic effects that are not already included can be defined in terms of existing concepts in typical eager functional languages. For example, backtracking can be expressed with call/cc and an updatable stack of backtracking points [5]. Still, such implementations appear ad-hoc, require a thorough understanding of the imperative features used, and have no clear connection to the "pure" monadic abstractions they implement. And although all of the usual monads seem to yield to this approach, it is far from obvious that they must all do so.

In the following, we will show that in fact *any* monadic effect whose definition is itself expressible in a functional language can be synthesized from just two "impure" constructs: first-class continuations and a

storage cell. In other words, a language like Scheme [2], or ML with first-class continuations [5], is already "monadically complete" in the sense that any program expressible in the somewhat contorted monadic style can also can be written in direct style. Moreover, all uses of computational effects in the definition can be encapsulated into an abstraction customarily called *composable*, *functional*, or *partial* continuations, and the remaining program contains no explicit references to either escapes or state.

The rest of this section contains a very brief introduction to monads (a reader unfamiliar with the concept would be well advised to read one of the papers by Moggi or Wadler for a more complete presentation) and Moggi's convenient notation for monadic effects. The following sections then derive the representation result as a succession of three steps, each of which is potentially useful in its own right and directly extends or supplements earlier work. First, we develop a formal correspondence between "monadic style" and continuation-passing style. Using this, we show that all non-standard manipulations of the continuation in "monadic CPS" can be expressed in terms of two opera-

tors for composable continuations. Finally, we show how to define these two operators using ordinary first-class continuations and a piece of state. To supplement the abstract development, section 5 presents the complete embedding as executable ML code and illustrates how some common monadic effects can be uniformly represented as instances of the construction. A comparison with related work and some conclusions complete the paper.

1.2 Monads and monadic reflection

For the purposes of "monadic functional programming", a *monad* consists of a type constructor T and operations (polymorphic functions)

$$\eta : \alpha \to T\alpha \qquad \text{and} \qquad -^* : (\alpha \to T\beta) \to T\alpha \to T\beta$$

called *unit* and *extension*, respectively. (Wadler uses a binary infix operator for the latter, writing m `` `bind` `` f or $m \star f$ for our $f^* m$. His notation is probably superior for writing actual programs in monadic style, but the variant above seems preferable for the formal manipulations we will be performing.) The operations are required to satisfy three *monad laws*:

$$\begin{aligned} \eta^* &= id_{T\alpha} \\ f^* \circ \eta &= f \\ (f^* \circ g)^* &= f^* \circ g^* \end{aligned}$$

Monads can be used to give a semantics of various "computational effects" (such as state, exceptions, or I/O) in applicative programming languages [14, 16]. In particular, our development is set in a simple call-by-value (CBV) functional language based on "Moggi's principle":

> *Computations of type α correspond to values of type $T\alpha$.*

Informally, ηa represents a "pure" (i.e., effect-free) computation yielding a, while f^*t represents the computation consisting of t's effects followed by the result of applying f to the value computed by t.

As also noted by Moggi, the correspondence principle can be embodied in an "introspective" language extension which could be called *monadic reflection* (by analogy to computational reflection [28, 34]), given by two operators:

$$\frac{\Gamma \vdash E : T\alpha}{\Gamma \vdash \mu(E) : \alpha} \quad \text{and} \quad \frac{\Gamma \vdash E : \alpha}{\Gamma \vdash [E] : T\alpha}$$

For any $E : T\alpha$, $\mu(E)$ *reflects* the value of E as an "effectful" computation of type α. Conversely, given a general computation $E : \alpha$, $[E]$ *reifies* it as the corresponding "effect-free" value of type $T\alpha$.

For example, let T be the exception monad, defined as

$$\begin{aligned} T\alpha &= \alpha + \mathsf{exn} \\ \eta &= \lambda a.\, \mathsf{inl}\ a \\ f^* &= \lambda t.\, \mathbf{case}\ t\ \mathbf{of}\ \mathsf{inl}\ a \to f\ a \parallel \mathsf{inr}\ e \to \mathsf{inr}\ e \end{aligned}$$

(where exn is a type of exception names). Then $\mu(E)$ expresses the value of $E : \alpha + \mathsf{exn}$ as a computation: we get an exception-raising construct by

$$\mathbf{raise}\ E \stackrel{\text{def}}{=} \mu(\mathsf{inr}\ E)$$

(where E is an expression – typically just a value – of type exn). Conversely, $[E]$ turns a possibly exception-raising α-expression E into a value of type $\alpha + \mathsf{exn}$, so we can define an exception-handling construct like this:

E_1 **handle** $e \Rightarrow E_2$

$$\stackrel{\text{def}}{=} \textbf{case } [E_1] \textbf{ of } \mathsf{inl}\ a \rightarrow a \parallel \mathsf{inr}\ e \rightarrow E_2$$

(i.e., if E_1 raises an exception, the handler E_2 is invoked with e bound to the exception name; a general pattern-matching **handle** construct like SML's can easily be expressed in terms of this one).

Justifying the designation as a "correspondence principle", monadic reification and reflection are inverses on their respective domains. That is, for any *expression* $E : \alpha$ (possibly with computational effects) and any *value* $V : T\alpha$,

$$\mu([E]) = E \qquad \text{and} \qquad [\mu(V)] = V$$

The more general notation $\mu(E)$ can be seen as simply shorthand for **let** $v = E$ **in** $\mu(v)$, so in practice $\mu(\cdot)$

would be provided as a function $\mathsf{reflect} : T\alpha \rightarrow \alpha$. It is not necessary to have $[\cdot]$ as a special form either: we can exploit the usual bijection between computations of type α and values of type $1 \rightarrow \alpha$ to get a function $\mathsf{reify} : (1 \rightarrow \alpha) \rightarrow T\alpha$, extracting the monadic representation from a *suspended* computation. For the theoretical

development in sections 2–4, however, we will keep the more compact $\mu(\cdot)/[\cdot]$-notation.

2 Monads and CPS

As the first step of our development, let us investigate the formal connections between "monadic style" and continuation-passing style (CPS). As noted by Wadler and others, the two appear closely related, but the actual correspondence is quite involved and benefits from a more detailed analysis.

In this section, we consider two translations (monadic and CPS) from a simply-typed CBV functional language with monadic reflection and reification operators (our *object language*) into a "purely functional" *meta-language*: a typed $\lambda_{\beta\eta}$-calculus with monadic unit and extension functions.

We then relate the two translations, generalizing the results of Meyer and Wand [13] about the typed CPS transform: their method can be seen as covering the particular case where T is the *identity* monad (i.e., $T\alpha = \alpha$, $\eta = id$, and $f^* = f$).

2.1 The monadic translation

The monadic translation transforms an object-language term E with free variables $x_1, \ldots, x_n$,

$$x_1 \colon \alpha_1, \ldots, x_n \colon \alpha_n \vdash E : \beta$$

into the meta-language term

$$x_1 \colon [\![\alpha_1]\!]_T, \ldots, x_n \colon [\![\alpha_n]\!]_T \vdash [\![E]\!]_T : T[\![\beta]\!]_T$$

The translation on types is given by:

$$\begin{aligned} [\![\iota]\!]_T &= \iota \\ [\![\alpha \rightharpoonup \beta]\!]_T &= [\![\alpha]\!]_T \to T[\![\beta]\!]_T \\ [\![T\alpha]\!]_T &= T[\![\alpha]\!]_T \end{aligned}$$

Here ι ranges over base types, and we use $\alpha \rightharpoonup \beta$ for the CBV function space to distinguish it from the underlying "pure" function space $\alpha \to \beta$. The extension to structured types (products, sums, etc.) is straightforward but omitted here for brevity. The term translation is given by

$$\begin{aligned} [\![x]\!]_T &= \eta\, x \\ [\![\lambda x.\, E]\!]_T &= \eta(\lambda x.\, [\![E]\!]_T) \\ [\![E_1\, E_2]\!]_T &= (\lambda f.\, f^*\, [\![E_2]\!]_T)^*\, [\![E_1]\!]_T \\ [\![\mu(E)]\!]_T &= id^*\, [\![E]\!]_T \end{aligned}$$

$$[\![\lceil E \rceil]\!]_T \;=\; \eta\,[\![E]\!]_T$$

(where the last two, perhaps less familiar-looking, equations are taken from Moggi [14].) As an example of monadic reasoning, let us quickly check that the monad laws verify the correspondence principle for $\lceil\cdot\rceil$ and $\mu(\cdot)$:

$$\begin{aligned}[\![\mu(\lceil E \rceil)]\!]_T &= id^*\,[\![\lceil E \rceil]\!]_T = id^*\,(\eta\,[\![E]\!]_T) \\ &= (id^* \circ \eta)\,[\![E]\!]_T = id\,[\![E]\!]_T = [\![E]\!]_T\end{aligned}$$

Conversely, taking x as a representative value (the other cases are analogous):

$$\begin{aligned}[\![\lceil \mu(x) \rceil]\!]_T &= \eta\,[\![\mu(x)]\!]_T = \eta(id^*\,[\![x]\!]_T) \\ &= \eta(id^*(\eta\,x)) = \eta x = [\![x]\!]_T\end{aligned}$$

2.2 The CPS translation

Let us now consider the CBV CPS translation for the same pair of languages, and in particular still with reflection and reification operators for the monad T. The translation on types looks similar:

$$\begin{aligned}[\![\iota]\!]_K &= \iota \\ [\![\alpha \multimap \beta]\!]_K &= [\![\alpha]\!]_K \to K\,[\![\beta]\!]_K\end{aligned}$$

$$[\![T\alpha]\!]_K \;=\; T[\![\alpha]\!]_K$$

where $K\gamma = (\gamma \to To) \to To$ for a type o of final answers. (The key idea of making To the "new" answer type is due to Wadler [32]). To get a simple relationship between the two translations, we assume o to contain all denotable values [22] (note that such an o does not have to be a type expressible in the source language).[1] Further, to avoid clutter in the term equations, we omit injections into and projections from o.

Now our source term E is translated into

$$x_1 : [\![\alpha_1]\!]_K, \ldots, x_n : [\![\alpha_n]\!]_K \vdash [\![E]\!]_K : K[\![\beta]\!]_K$$

where the term translation is given by

$$\begin{aligned}
[\![x]\!]_K &= \lambda k.\, k\, x \\
[\![\lambda x.\, E]\!]_K &= \lambda k.\, k(\lambda x.\, [\![E]\!]_K) \\
[\![E_1\, E_2]\!]_K &= \lambda k.\, [\![E_1]\!]_K\, (\lambda f.\, [\![E_2]\!]_K\, (\lambda a.\, f\, a\, k)) \\
[\![\mu(E)]\!]_K &= \lambda k.\, [\![E]\!]_K\, k^* \\
[\![\lfloor E \rfloor]\!]_K &= \lambda k.\, k([\![E]\!]_K\, \eta)
\end{aligned}$$

The first three equations are the usual ones [19]. We will verify that the last two really are the correct CPS analogs of their T-translation counterparts next.

2.3 A relation between between monadic style and CPS

Let us first note that we can define a type-indexed family of functions mediating between monadic and CPS

[1] Alternatively, with a little more care, we can take $K\gamma = \forall o.\,(\gamma \to To) \to To$; it is straightforward to check that both the term translation and the operations defined in the following can in fact be typed according to this schema.

types, $\phi_\alpha : [\![\alpha]\!]_T \to [\![\alpha]\!]_K$ and $\psi_\alpha : [\![\alpha]\!]_K \to [\![\alpha]\!]_T$:

$$\begin{aligned}
\phi_\iota &= \lambda i.\, i \\
\phi_{\alpha\to\beta} &= \lambda f.\, \lambda x.\, \lambda k.\, (k \circ \phi_\beta)^* \,(f\,(\psi_\alpha\, x)) \\
\phi_{T\,\alpha} &= T(\phi_\alpha) \;=\; (\eta \circ \phi_\alpha)^*
\end{aligned}$$

$$\begin{aligned}
\psi_\iota &= \lambda i.\, i \\
\psi_{\alpha\to\beta} &= \lambda g.\, \lambda y.\, g\,(\phi_\alpha\, y)\,(\eta \circ \psi_\beta) \\
\psi_{T\,\alpha} &= T(\psi_\alpha) \;=\; (\eta \circ \psi_\alpha)^*
\end{aligned}$$

(Meyer and Wand use the names i and j for functions analogous to ϕ and ψ, but the definition of ψ given above

is slightly more convenient when T is not necessarily the identity.)

It is straightforward to verify that $\langle \phi_\gamma, \psi_\gamma \rangle$ form a retraction pair, i.e., that for any source-type γ,

$$\psi_\gamma \circ \phi_\gamma = id_{[\![\gamma]\!]_T}$$

in the metalanguage. For $\gamma = \iota$, the result is immediate; for $\gamma = \alpha \rightharpoonup \beta$:

$$\begin{aligned}
&\psi_{\alpha\rightharpoonup\beta} \circ \phi_{\alpha\rightharpoonup\beta} \\
&= \lambda f.\, \psi_{\alpha\rightharpoonup\beta}\,(\phi_{\alpha\rightharpoonup\beta}\, f) \\
&= \lambda f.\, \psi_{\alpha\rightharpoonup\beta}\,(\lambda x.\, \lambda k.\, (k \circ \phi_\beta)^*\,(f\,(\psi_\alpha\, x))) \\
&= \lambda f.\, \lambda y.\, [\lambda x.\, \lambda k.\, (k \circ \phi_\beta)^*\,(f\,(\psi_\alpha\, x))]\,(\phi_\alpha\, y)\,(\eta \circ \psi_\beta) \\
&= \lambda f.\, \lambda y.\, (\eta \circ \psi_\beta \circ \phi_\beta)^*\,(f\,(\psi_\alpha\,(\phi_\alpha\, y))) \\
&\overset{\text{ih}}{=} \lambda f.\, \lambda y.\, \eta^*\,(f\, y) \\
&= id
\end{aligned}$$

and for $\gamma = T\alpha$:

$$\begin{aligned}
\psi_{T\alpha} \circ \phi_{T\alpha} &= (\eta \circ \psi_\alpha)^* \circ (\eta \circ \phi_\alpha)^* \\
&= ((\eta \circ \psi_\alpha)^* \circ \eta \circ \phi_\alpha)^* = (\eta \circ \psi_\alpha \circ \phi_\alpha)^* \overset{\text{ih}}{=} \eta^* = id
\end{aligned}$$

Thus the type translations faithfully embed the T-

translation of a source-language *type* in the corresponding K-translated type. But to properly relate the two translations, we want the stronger property that the T-meaning of any source *term* can always be recovered from its K-meaning, i.e., that the CPS-translation really captures all the subtleties of the monadic one.

The proof of this property is more complicated than might be expected: in particular, an attempt to prove it by induction on the term structure alone will not work. To get a feeling for what goes wrong, consider the *untyped* variants of the translations with T as simply the identity monad (so in particular, the $\mu(\cdot)$ and $[\cdot]$ operations have no effect). Now let U be any atomic value, and consider the term

$$E = (\lambda d.\, U)((\lambda x.\, x\, x)(\lambda x.\, x\, x))$$

Then $[\![E]\!]_T = E = U$ (remember that we have full β in our metalanguage, even though $[\![\cdot]\!]_T$ is nominally a CBV translation), but since $[\![\cdot]\!]_K$ still specifies a CBV CPS transformation, $[\![E]\!]_K\,(\lambda x.\, x) \neq U$, and in fact there is no functional way to "extract" U from $[\![E]\!]_K$.

More abstractly, the problem is that we have actually introduced an effect (nontermination) in the source language without a corresponding modification of the monadic structure to encompass partiality. To rule out

such surprises, we need to make explicit use of the type structure.

Specifically, for any source type α, we isolate a set of "T-compatible" CPS values $\mathcal{V}_\alpha \subseteq [\![\alpha]\!]_K$ and computations $\mathcal{C}_\alpha \subseteq K[\![\alpha]\!]_K$, defined as follows:

$$\begin{aligned}
\mathcal{V}_\iota &= \{m \in \iota \mid true\} \\
\mathcal{V}_{\alpha\rightharpoonup\beta} &= \{m \in [\![\alpha]\!]_K \to K[\![\beta]\!]_K \mid \forall n \in \mathcal{V}_\alpha . mn \in \mathcal{C}_\beta \\
&\qquad\qquad \wedge (\psi_{\alpha\rightharpoonup\beta}\, m)(\psi_\alpha\, n) = mn\,(\eta \circ \psi_\beta)\} \\
\mathcal{V}_{T\alpha} &= \{m \in T[\![\alpha]\!]_K \mid \lambda c.\, c^*\, m \in \mathcal{C}_\alpha\} \\
\\
\mathcal{C}_\alpha &= \{t \in K[\![\alpha]\!]_K \mid \lambda c.\, c^*\,(t\,\eta_{|\mathcal{V}_\alpha}) = t\}
\end{aligned}$$

(where $\eta_{|\mathcal{V}_\alpha}$ is the restriction of η to $\mathcal{V}_\alpha$; the equation in $\mathcal{C}_\alpha$ means that the left-hand side is defined and equal to the right-hand side). Part of the result we are aiming for states that the K-meanings of all object-language values and expressions are in fact T-compatible in this sense.

The motivation for all the specific conditions is fairly technical, but we can try to give some intuition. Most importantly, if $t \in \mathcal{C}_\alpha$ then

$$\begin{aligned}
k^*\,(t\,f) = k^*\,([\lambda c.\, c^*\,(t\,\eta_{|\mathcal{V}_\alpha})]\,f) &= k^*\,(f^*\,(t\,\eta_{|\mathcal{V}_\alpha})) \\
&= (k^* \circ f)^*\,(t\,\eta_{|\mathcal{V}_\alpha}) = t\,(k^* \circ f)
\end{aligned}$$

and hence in particular

$$\begin{aligned} t\,k &= k^*\,(t\,\eta_{|\mathcal{V}_\alpha}) = k^*\,(t\,(\eta \circ id_{\mathcal{V}_\alpha})) \\ &= t\,(k^* \circ \eta \circ id_{\mathcal{V}_\alpha}) = t\,(k \circ id_{\mathcal{V}_\alpha}) = t\,k_{|\mathcal{V}_\alpha} \end{aligned}$$

(i.e., a $\mathcal{C}_\alpha$-term t only invokes its continuation with a $\mathcal{V}_\alpha$-term, so that if k and k' agree on $\mathcal{V}_\alpha$ then $t\,k = t\,k'$). The first condition for functions and the one for $T\alpha$ ensure that "latent" computations involving only $\mathcal{V}$-terms (in particular, arguments of continuations) are well-behaved when activated; note that the translation of $\mu(E)$ can be written as

$$[\![\mu(E)]\!]_K = \lambda k.\,[\![E]\!]_K\,(\lambda m.\,[\lambda c.\,c^*\,m]\,k)$$

Finally, the second condition for functions expands to

$$m\,(\phi_\alpha\,(\psi_\alpha\,n))\,(\eta \circ \psi_\beta) = m\,n\,(\eta \circ \psi_\beta)$$

which states that a "well-behaved" function m cannot itself depend on more information about its argument n than what is preserved by conversion back to T-translated types.

449

It is easy to check by induction on types that

$$\phi_\alpha\, v \in \mathcal{V}_\alpha \text{ for any } v \in [\![\alpha]\!]_T$$

(so in particular if $w = \phi_\alpha\,(\psi_\alpha\, w)$ then $w \in \mathcal{V}_\alpha$). But not every element of $\mathcal{V}_\alpha$ is of this form. For example, let T again be the identity monad and consider the source values

$$\lambda f.\,\lambda x.\,x \qquad \text{and} \qquad \lambda f.\,\lambda x.\,(\lambda d.\,x)\,(f\,x)$$

Their T-meanings are equal, but their K-meanings are not, so only one of the latter can be in the image of ϕ.

For a substitution σ, let us write $M\{\sigma\}$ (to avoid yet another overloading of brackets) for the capture-avoiding application of σ to a meta-language term M, and $M\{\sigma \circ \theta\}$ for $(M\{\sigma\})\{\theta\}$. We can then state a key result relating CPS and monadic style.

Theorem *Let $x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n \vdash E : \beta$, and let σ be a substitution assigning a $\mathcal{V}_{\alpha_i}$-term to each x_i. Then*

$$([\![E]\!]_K)\{\sigma\} \in \mathcal{C}_\beta$$

and

$$[\![E]\!]_T\{\psi_{\bar{\alpha}} \circ \sigma\} \;=\; [\![E]\!]_K\{\sigma\}(\eta \circ \psi_\beta)$$

(where $\psi_{\bar{\alpha}}$ is the substitution mapping x_i to $\psi_{\alpha_i} x_i$ for $1 \le i \le n$).

The proof is by somewhat tedious structural induction

on E (Meyer and Wand's shortcut of only analyzing SK-combinators does not appear as useful in the general case). As a direct consequence, we get:

$$[\![E]\!]_T = [\![E]\!]_T\{\psi_{\bar{\alpha}} \circ \phi_{\bar{\alpha}}\} = [\![E]\!]_K\{\phi_{\bar{\alpha}}\}(\eta \circ \psi_\beta)$$

In particular, if E is closed of base type (so ψ_β is the identity), we have the simple equality $[\![E]\!]_T = [\![E]\!]_K\,\eta$. More generally, using the above and the first half of the theorem, we get a *monadic congruence* result:

$$k^*\,[\![E]\!]_T \;=\; [\![E]\!]_K\{\phi_{\bar{\alpha}}\}(k \circ \psi_\beta)$$

For example, in the case of the *partiality monad* [16], $T\alpha = \alpha \uplus \{-\}$, with k^* as the *strict extension* of k (i.e., $k^*\,a = k\,a$ for $a \in \alpha$; $k^*\,- = -$), we recover the usual restriction [25] that the continuation be strict to get a congruence; the monadic characterization generalizes this requirement to other computational effects.

Finally, we can check explicitly that the reflection principle is satisfied when all free variables denote $\mathcal{V}$-terms:

$$[\![\mu(\texttt{[}E\texttt{]})]\!]_K = \lambda k.\,[\![\texttt{[}E\texttt{]}]\!]_K\,k^* = \lambda k.\,k^*([\![E]\!]_K\,\eta) = [\![E]\!]_K$$

and

$$[\![\texttt{[}\mu(x)\texttt{]}]\!]_K = \lambda k.\,k([\![\mu(x)]\!]_K\,\eta) = \lambda k.\,k([\![x]\!]_K\,\eta^*)$$

$$= \lambda k.\, k\,(\eta^*\, x) = \lambda k.\, k\, x = [\![x]\!]_K$$

Armed with a proof that the continuation-passing characterization of monadic reflection and reification faithfully represents the original definitions, we can now return to the embedding result.

3 Monadic Reflection from Composable Continuations

The analysis in the previous section applies to an arbitrary monad T. But let us now make the natural assumption that the *meta-language* monad functions η and $-^*$ can actually be defined in the "pure" (i.e., effect-free) functional sublanguage of our *object language*. In other words, the definition of the monad must be sufficiently "algorithmic" that we can write a source program in monadic style in the first place! If this is the case, we say that the monad T is *expressible* in our language.

As we have seen, we can express all monadic effects in CPS instead of in monadic style. A priori, this does not leave us not much better off, however: to reach the "non-standard" CPS terms used to interpret $\mu(\cdot)$ and $[\![\cdot]\!]$ for any particular monad T, we still seem to need

a T-specific translation phase (performed either manually, or by compilation, interpretation, partial evaluation, or some other automated technique). But given object-language terms for η and $-^*$, it turns out that we can represent all the required CPS terms in direct style extended with two *fixed* operators for manipulating the continuation as a "composable" function. Thus every expressible monad can be simulated by a single, "universal" effect which could be added to our object language once and for all.

Specifically, we extend the source language of the CPS translation with operators shift and reset, defined as follows:

$$\begin{aligned}
[\![\mathcal{S}\,E]\!]_K &= \lambda\kappa.\,[\![E]\!]_K\,(\lambda f.\,f\,[\lambda v.\,\lambda\kappa'.\,\kappa'\,(\kappa\,v)]\,(\lambda x.\,x)) \\
[\![\#E]\!]_K &= \lambda\kappa.\,\kappa\,([\![E]\!]_K\,(\lambda x.\,x))
\end{aligned}$$

shift captures (and erases) the evaluation context up to the nearest dynamically enclosing reset (every program is run with an implicit all-enclosing reset), and passes this context to its argument as an ordinary function. For example:

$$\begin{aligned}
&1 + \#(2 \times \mathcal{S}(\lambda k.\,k\,(k\,10))) \\
&\qquad = 1 + \mathbf{let}\ k = \lambda v.\,2 \times v\ \mathbf{in}\ k\,(k\,10) \;=\; 41
\end{aligned}$$

(For our purposes, reset coincides with Felleisen's prompt [6], whose #-notation we have adopted here; but shift differs from prompt's original companion control (or $\mathcal{F}$) in that the continuation κ is *not* given control over κ' in the definition of $\mathcal{S}$.) For more details on shift/reset and their relation to other notions of composable continuations, see [3, 4, 17, 33]. As with the monadic $[\![\cdot]\!]$, the operation $\#\cdot$ would typically be provided as a function on thunks rather than as a special form.

By our assumption that the meta-language η can be included in the object language as a "pure" function of type $\alpha \rightharpoonup T\alpha$, we have

$$[\![\eta]\!]_K \;=\; \lambda k.\, k\,(\lambda a.\, \lambda\kappa.\, \kappa\,(\eta\, a))$$

$$\begin{array}{rcl}
\mathcal{E}[\![x]\!]\rho & = & \lambda\kappa.\, \lambda\underline{\gamma}.\, \kappa\,(\rho\, x)\,\underline{\gamma} \\
\mathcal{E}[\![\lambda x.\, E]\!]\rho & = & \lambda\kappa.\, \lambda\underline{\gamma}.\, \kappa\,(\lambda v.\, \lambda\kappa'.\, \lambda\underline{\gamma'}.\, \mathcal{E}[\![E]\!](\rho[x \mapsto v])\,\kappa'\,\underline{\gamma'})\,\underline{\gamma} \\
\mathcal{E}[\![E_1\, E_2]\!]\rho & = & \lambda\kappa.\, \lambda\underline{\gamma}.\, \mathcal{E}[\![E_1]\!]\rho\,(\lambda f.\, \lambda\underline{\gamma'}.\, \mathcal{E}[\![E_2]\!]\rho\,(\lambda a.\, \lambda\underline{\gamma''}.\, f\,a\,\kappa\,\underline{\gamma''})\,\underline{\gamma'})\,\underline{\gamma} \\
\mathcal{E}[\![\mathcal{K}\, E]\!]\rho & = & \lambda\kappa.\, \lambda\underline{\gamma}.\, \mathcal{E}[\![E]\!]\rho\,(\lambda f.\, \lambda\underline{\gamma'}.\, f[\lambda v.\, \lambda\kappa'.\, \lambda\underline{\gamma''}.\, \kappa\, v\,\underline{\gamma''}]\,\kappa\,\underline{\gamma'})\,\underline{\gamma} \\
\mathcal{E}[\![\mathcal{S}\, E]\!]\rho & = & \lambda\kappa.\, \lambda\underline{\gamma}.\, \mathcal{E}[\![E]\!]\rho\,(\lambda f.\, \lambda\underline{\gamma'}.\, f[\lambda v.\, \lambda\kappa'.\, \lambda\gamma''.\, \kappa\, v\,(\lambda w.\, \kappa'\, w\,\gamma'')]\,(\lambda x.\, \lambda\gamma''.\, \gamma''\, x)\,\underline{\gamma'})\,\underline{\gamma} \\
\mathcal{E}[\![\#\, E]\!]\rho & = & \lambda\kappa.\, \lambda\gamma.\, \mathcal{E}[\![E]\!]\rho\,(\lambda x.\, \lambda\gamma'.\, \gamma'\, x)\,(\lambda r.\, \kappa\, r\,\gamma)
\end{array}$$

Figure 1: Meta-continuation semantics

Similarly, extension is expressible as a function of type

$(\alpha \rightharpoonup T\beta) \rightharpoonup T\alpha \rightharpoonup T\beta$ that preserves "purity":

$$\begin{aligned} [\![f^*]\!]_K &= \lambda k.\, k\,(\lambda a.\, \lambda\kappa.\, \kappa\,(\varphi^*\, a)) \\ \text{if } [\![f]\!]_K &= \lambda k.\, k\,(\lambda a.\, \lambda\kappa.\, \kappa\,(\varphi\, a)) \end{aligned}$$

Now, in the CPS definition of $[\![E]\!]$ we want to evaluate E with a continuation η, and only then propagate the result to the surrounding evaluation context. This is in fact almost what $\#\cdot$ does – we only need to add the η:

$$\begin{aligned} &[\![\#(\eta\, E)]\!]_K \\ &\quad = \lambda\kappa.\, \kappa([\![\eta\, E]\!]_K\,(\lambda x.\, x)) \\ &\quad = \lambda\kappa.\, \kappa([\![\eta]\!]_K\,(\lambda f.\, [\![E]\!]_K\,(\lambda a.\, f\, a\,(\lambda x.\, x)))) \\ &\quad = \lambda\kappa.\, \kappa([\![E]\!]_K\,(\lambda v.\, [\lambda a.\, \lambda\kappa.\, \kappa(\eta\, a)]\, v\,(\lambda x.\, x))) \\ &\quad = \lambda\kappa.\, \kappa([\![E]\!]_K\,(\lambda v.\, (\lambda x.\, x)(\eta\, v))) \\ &\quad = \lambda\kappa.\, \kappa([\![E]\!]_K\,\eta) \\ &\quad = [\![[E]]\!]_K \end{aligned}$$

Conversely, for $\mu(E)$ we need to replace the current continuation with its extended version, which again can be directly expressed using $\mathcal{S}$:

$$\begin{aligned} &[\![\mathcal{S}\,(\lambda k.\, k^*\, E)]\!]_K \\ &\quad = \lambda\kappa.\, [\![\lambda k.\, k^*\, E]\!]_K\,(\lambda f.\, f\,(\lambda v.\, \lambda\kappa'.\, \kappa'(\kappa\, v))(\lambda x.\, x)) \\ &\quad = \lambda\kappa.\, [\lambda k.\, [\![k^*\, E]\!]_K](\lambda v.\, \lambda\kappa'.\, \kappa'(\kappa\, v))(\lambda x.\, x) \end{aligned}$$

$$
\begin{aligned}
&= \lambda\kappa.\,[\lambda k.\,[\![k^*]\!]_K\,(\lambda g.\,[\![E]\!]_K\,(\lambda a.\,g\,a\,(\lambda x.\,x)))]\\
&\qquad\quad (\lambda v.\,\lambda\kappa'.\,\kappa'\,(\kappa\,v))\\
&= \lambda\kappa.\,[\![E]\!]_K\,(\lambda a.\,[\lambda v.\,\lambda\kappa'.\,\kappa'\,(\kappa^*\,v)]\,a\,(\lambda x.\,x))\\
&= \lambda\kappa.\,[\![E]\!]_K\,(\lambda a.\,(\lambda x.\,x)\,(\kappa^*\,a))\\
&= \lambda\kappa.\,[\![E]\!]_K\,\kappa^*\\
&= [\![\mu(E)]\!]_K
\end{aligned}
$$

(where $k \notin FV(E)$). This means that for any expressible η and $-^*$, we can define $\texttt{[}E\texttt{]}$ and $\mu(E)$ in terms of the composable-continuation primitives:

$$
\begin{aligned}
\texttt{[}E\texttt{]} &\stackrel{\text{def}}{=} \#(\eta\,E)\\
\mu(E) &\stackrel{\text{def}}{=} \mathcal{S}(\lambda k.\,k^*\,E)
\end{aligned}
$$

So if we only include shift and reset in our programming language, we can write all "monadic" programs in direct style.

4 Composable Continuations from Storable Continuations

In the final step of our construction, we will see that shift and reset can themselves be defined in terms of non-composable continuations and a single storage cell. The trick is to view the CPS translation with continuation-

composing definitions of shift and reset as a *direct-style* specification of a language (with κ as just another higher-order function), and obtain from it a proper continuation semantics using a new "meta-continuation" γ, as detailed in [3].

The result is displayed in Figure 1. (Here $\mathcal{K}$ is the usual call/cc-operator which invokes its argument on the current continuation represented as an *escaping* function, as seen by the discarded κ'.) Note in particular how the nested application $\kappa'(\kappa\, v)$ in the definition of $\mathcal{S}$ is sequentialized into the usual $\lambda\gamma''.\, \kappa\, v\, (\lambda w.\, \kappa'\, w\, \gamma'')$; likewise, the outer κ in $[\![\#E]\!]_K$ is put onto the meta-continuation. On the other hand, all the underlined γ's can actually be η-reduced away, so the metacontinuation only really comes into play in shift and reset. The meaning of a complete program is now

$$\mathcal{E}[\![E]\!]\rho_{init}\, (\lambda x.\, \lambda\gamma.\, \gamma\, x)\, \gamma_{init}$$

where γ_{init} is the usual top-level continuation, typically simply the identity function.

First, let us note that given $\mathcal{K}$, #, and the simpler operator $\mathcal{A}$ ("abort") with denotation

$$\mathcal{E}[\![\mathcal{A}\, E]\!]\rho \;=\; \lambda\kappa.\, \lambda\underline{\gamma}.\, \mathcal{E}[\![E]\!]\rho(\lambda v.\, \lambda\gamma'.\, \gamma'\, v)\,\underline{\gamma}$$

[so $\mathcal{A}\,E$ is equivalent to $\mathcal{S}(\lambda d.\,E)$, where $d \notin FV(E)$], we can express $\mathcal{S}$ as

$$\mathcal{S}\,E \stackrel{\text{def}}{=} \mathcal{K}(\lambda k.\,\mathcal{A}(E(\lambda x.\,\#(k\,x))))$$

(informally, the $\mathcal{A}$ erases the context once it is captured as k; and by wrapping a `reset` around $k\,x$, we ensure that only the identity continuation gets discarded when k is invoked). Thus, we only need to define $\mathcal{A}$ and #. (That $\mathcal{K}$, $\mathcal{A}$, and # together suffice for defining all "pure" CPS

terms in a domain-theoretic setting was already noted by Sitaram and Felleisen [27]).

The second, key observation is that – except for the definitions of $\mathcal{A}$ and # – the meta-continuation is threaded through the meta-continuation semantics exactly like the global *store* in a Scheme-like language! This means that we we can simply designate a single, updatable location to hold γ (represented as a procedure that ignores both the continuation and the metacontinuation passed to it), and only access it in $\mathcal{A}$ and #.

Specifically, in a language with continuations and

state, we have operations

$$\begin{aligned}
&\mathcal{E}[\![\mathsf{mk} := E]\!]\rho \\
&\qquad = \lambda\kappa.\,\lambda\sigma.\,\mathcal{E}[\![E]\!]\rho\,(\lambda v.\,\lambda\sigma'.\,\kappa\,()\,(\sigma'[\ell_{\mathsf{mk}} \mapsto v]))\,\sigma \\
&\mathcal{E}[\![!\mathsf{mk}]\!]\rho \;=\; \lambda\kappa.\,\lambda\sigma.\,\kappa\,(\sigma\,\ell_{\mathsf{mk}})\,\sigma
\end{aligned}$$

(where σ maps locations to values; ℓ_{mk} is the location assigned to mk; $\mathsf{mk} := E$ evaluates E, stores the value, and returns (); and !mk returns the current contents of the cell without changing it). It is then easy to check that the following definitions give terms with the right denotations:

$$\begin{aligned}
\mathcal{A}\,E &\stackrel{\text{def}}{=} \mathbf{let}\ v = E\ \mathbf{in}\ !\mathsf{mk}\ v\ \ [= (\lambda v.\,!\mathsf{mk}\ v)\,E] \\
\#E &\stackrel{\text{def}}{=} \mathcal{K}(\lambda k.\,\mathbf{let}\ m = !\mathsf{mk} \\
&\qquad\qquad \mathbf{in}\ (\mathsf{mk} := (\lambda r.\,(\mathsf{mk} := m; k\,r)); \mathcal{A}\,E))
\end{aligned}$$

(where **let** and ; are the usual abbreviations). Note in particular that since k is an escaping function, the λ-abstraction stored into mk in $\#E$ does denote a procedure that when invoked uses neither its continuation nor the current contents of mk. We also need to initialize mk to the initial continuation; the easiest way to do this is to simply wrap a reset around any top-level

expression to be evaluated.

This completes our embedding of any expressible monadic structure into a language with escapes and state, a somewhat surprising result given the deceptively general-appearing monad laws. It should be stressed again, though, that the construction only applies to monads whose definitions can be captured as functional programs in the first place: more esoteric effects like probabilistic computations defy such a simple decomposition.

Incidentally, the above definitions of shift and reset in terms of call/cc and and state could well have practical applications unrelated to monads. For example, Lawall and Danvy are investigating applications of composable continuations for continuation-based partial evaluation [12]; preliminary results indicate that using the embedded shift/reset instead of an explicit CPS transformation step can give significant improvements in time and in space, when run under an efficient implementation of call/cc [9].

5 Implementation and Examples

In this section we transcribe the abstract construction presented so far into runnable code. To emphasize the

typing issues involved, we use the New Jersey dialect of Standard ML [1] as our concrete language, but the operational content should translate straightforwardly into Scheme as well (though instantiation to different monads may be less convenient without a module facility). We also give several examples; the reader may want to compare these with Wadler's presentation [32].

5.1 Composable continuations

In SML/NJ, first-class continuations have a type distinct from the type of general procedures. Let us therefore first set up a representation of such continuations as Scheme-style non-returning functions (this is not essential but makes for a more direct correspondence with the semantics in section 4):

```
signature ESCAPE =
  sig
    type void
    val coerce : void -> 'a
    val escape : (('1a -> void) -> '1a) -> '1a
  end;

structure Escape : ESCAPE =
  struct
    datatype void = VOID of void
    fun coerce (VOID v) = coerce v
    fun escape f = callcc (fn k=>f (fn x=>throw k x))
```

```
  end;
```

For example, we can write

```
let open Escape
in 3 + escape (fn k=>6 + coerce (k 1)) end;
(* val it = 4 : int *)
```

(The use of void and coerce instead of an unconstrained type variable in escape permits storage of continuations in ref-cells while staying within the ML type system [5].)

Now we can define a composable-continuations facility, parameterized by the type of final answers:

```
signature CONTROL =
  sig
    type ans
    val reset : (unit -> ans) -> ans
    val shift : (('1a -> ans) -> ans) -> '1a
  end;

functor Control (type ans) : CONTROL =
  struct
    open Escape
    exception MissingReset
    val mk : (ans -> void) ref =
      ref (fn _=>raise MissingReset)
    fun abort x = coerce (!mk x)

    type ans = ans
    fun reset t = escape (fn k=>let val m = !mk in
                    mk := (fn r=>(mk := m; k r));
                    abort (t ()) end)
```

```
    fun shift h = escape (fn k=>abort (h (fn v=>
                    reset (fn ()=>coerce (k v)))))
  end;
```


For example,

```
structure IntCtrl = Control (type ans = int);

let open IntCtrl
in 1 + reset (fn ()=>2 * shift (fn k=>k (k 10))) end;
(* val it = 41 : int*)
```

5.2 Monadic reflection

Building on the composable-continuations package, we implement the construction of Section 3. The signature of a monad is simple:

```
signature MONAD =
  sig
    type 'a t
    val unit : 'a -> 'a t
    val ext : ('a -> 'b t) -> 'a t -> 'b t
  end;
```

(the monad laws have to be verified manually, though). Our goal is to define reflection and reification operations

for an arbitrary monad `M` to get

```
signature RMONAD =
  sig
    structure M : MONAD
    val reflect : 'a1 M.t -> 'a1
    val reify : (unit -> 'a) -> 'a M.t
  end;
```

Before we can proceed, however, there is one twist: our construction needs a *universal type* (the o of section 2.2):

```
signature UNIVERSAL =
  sig
    type u
    val to_u : 'a -> u
    val from_u : u -> 'a
  end;
```

such that `from_u` $\circ$ `to_u` is the identity for any `'a`. (Note that ensuring that the instances of `'a` do in fact match up *dynamically* now becomes our responsibility; the ML system is free to dump core on attempts to execute code like `1 + from_u (to_u "foo")`). This signature can be implemented in SML/NJ as

```
structure Universal : UNIVERSAL =
  struct
    type u = System.Unsafe.object
```

```
    val to_u = System.Unsafe.cast
    val from_u = System.Unsafe.cast
  end;
```

where `cast` behaves as an identity function, but has the general type `'a -> 'b`.[2] We can now complete the construction:

[2] Even without a universal type, we still get a usable definition if we pick a suitable concrete type of answers. Then reification becomes restricted to computations of that type, but reflection remains polymorphic; in many cases, e.g., in an interpreter where all evaluations happen at a single type of denotable values, this is sufficient.

```
functor Represent (M : MONAD) : RMONAD =
  struct
    structure C = Control (type ans = Universal.u M.t)

    structure M = M
    fun reflect m = C.shift (fn k=>M.ext k m)
    fun reify t = M.ext (M.unit o Universal.from_u)
                    (C.reset (fn ()=>M.unit
                                (Universal.to_u (t ()))))
  end;
```

(Recall that operationally `to_u` and `from_u` are identities, and so is `M.ext M.unit`. Also, it is worth stressing that only the *implementation* of `Represent` needs a typing loophole; its *interface* remains ML-typable and safe.)

5.3 Example: exceptions

The example from the introduction becomes, in concrete syntax:

```
structure ErrorMonad =
  struct
    datatype 'a t = SUC of 'a | ERR of string
    val unit = SUC
    fun ext f (SUC a) = f a
      | ext f (ERR s) = (ERR s)
  end;

structure ErrorRep = Represent (ErrorMonad);

local open ErrorMonad ErrorRep in
  fun myraise e = reflect (ERR e)
  fun myhandle t h = case reify t of SUC a => a
                                  | ERR s => h s
end;
(* val myraise = fn : string -> 'a1a
   val myhandle = fn : (unit -> 'a) -> (string -> 'a)
                          -> 'a *)

fun show t =
  myhandle (fn ()=>"OK: " ^ makestring (t ():int))
           (fn s=>"Error: " ^ s);
(* val show = fn : (unit -> int) -> string *)

show (fn ()=>1 + 2);
(* val it = "OK: 3" : string *)

show (fn ()=>1 + myraise "oops");
```

```
(* val it = "Error: oops" : string *)
```

5.4 Example: state

The state monad with Wadler's counting operations:

```
functor StateMonad (type state) : MONAD =
  struct
    type 'a t = state -> 'a * state
    fun unit a = fn s0=>(a,s0)
    fun ext f m = fn s0=>let val (a,s1) = m s0
                         in f a s1 end
  end;

structure IntStateRep =
  Represent (StateMonad (type state = int));

fun tick () = IntStateRep.reflect (fn s=>((),s+1))
fun fetch () = IntStateRep.reflect (fn s=>(s,s))
fun store n = IntStateRep.reflect (fn s=>((),n));
(* val tick = fn : unit -> unit
   val fetch = fn : unit -> ?.<Parameter>.state(*= int*)
   val store = fn : int -> unit *)
```



```
#1 (IntStateRep.reify (fn ()=>(store 5; tick ();
                                2 * fetch ())) 0);
(* val it = 12 : int *)
```

5.5 Example: nondeterminism

A nondeterministic computation can be represented as a *list* of answers:

```
structure ListMonad : MONAD =
  struct
    type 'a t = 'a list
    fun unit x = [x]
    fun ext f [] = []
      | ext f (h::t) = f h @ ext f t
  end;

structure ListRep = Represent (ListMonad);

local open ListRep in
  fun amb (x,y) = reflect (reify (fn ()=>x) @
                           reify (fn ()=>y))
  fun fail () = reflect []
end;
(* val amb = fn : '1a * '1a -> '1a
   val fail = fn : unit -> '1a *)

ListRep.reify (fn ()=>let val x = amb (3,4) * amb (5,7)
                      in if x >= 20 then x
                         else fail () end);
(* val it = [21,20,28] : int ListMonad.t *)
```

More generally, we get Haskell-style list comprehensions "for free", in that the schema

$$[E \mid x_1 \leftarrow E_1;\ \ldots;\ x_n \leftarrow E_n]$$

(where each x_i may be used in $E_{i+1}, \ldots, E_n$ and in E) can be expressed directly as

$$\textbf{[let } x_1 = \mu(E_1) \textbf{ in } \ldots \textbf{ let } x_n = \mu(E_n) \textbf{ in } E\textbf{]}$$

For example, we can compute the "cartesian product" of two lists as

```
let open ListRep in
  reify (fn ()=>let val x = reflect [3, 4, 5];
                    val y = reflect ["foo", "bar"]
                in (x,y) end)
end;
(* val it = [(3,"foo"),(3,"bar"),(4,"foo"),(4,"bar"),
             (5,"foo"),(5,"bar")]: (int * string) list *)
```

Of course, this is probably not the most efficient way of implementing list comprehensions in ML. As observed by Wadler [31], however, list comprehensions can be generalized to arbitrary monads; similarly we get general monad comprehensions in ML simply by supplying the appropriate `[·]` and $\mu(\cdot)$ operations.

5.6 Example: continuations

Finally, let us consider the continuation monad (for an

arbitrary but fixed answer type):

```
functor ContMonad (type answer) : MONAD =
  struct
    type 'a t = ('a -> answer) -> answer
    fun unit x = fn k=>k x
    fun ext f t = fn k=>t (fn v=>f v k)
  end;

structure ContRep =
  Represent (ContMonad (type answer = string));

local open ContRep in
  fun myescape h =
    reflect (fn c=>let fun k a = reflect (fn c'=>c a)
                   in reify (fn ()=>h k) c end)
  fun myshift h =
    reflect (fn c=>let fun k a = reflect (fn c'=>c' (c a))
                   in reify (fn ()=>h k) (fn x=>x) end)
  fun myreset t = reflect (fn c=>c (reify t (fn x=>x)))
end;
(* val myescape = fn : (('1a -> '1b) -> '1a) -> '1a
   val myshift = fn : (('1a -> string) -> string) -> '1a
   val myreset = fn : (unit -> string) -> string *)

ContRep.reify (fn ()=>3 + myescape (fn k=>6 + k 1))
              makestring;
(* val it = "4" : string *)

ContRep.reify (fn ()=>"a" ^ myreset (fn ()=>
                                "b" ^ myshift (fn k=>
                                        k (k "c"))))
              (fn x=>x)
(* val it = "abbc" : string *)
```

6 Related Work

The study of relationships between direct and continuation semantics has a long history. Early investigations [22, 25, 30] were set in a domain-theoretic framework where the main difficulties concerned reflexive domains; as a result, these methods and results seem closely tied to specific semantic models. On the other hand, Meyer and Wand's more abstract approach applies to all models of (typed) λ-calculi, but does not encompass computational effects – not even nontermination. The present extension of Meyer and Wand's retraction theorem to monadic effects should partially bridge this gap, and add another facet to our understanding of CPS. It seems natural to expect other results about continuation-passing vs. direct style to scale up to monadic style as well; in particular, it should be possible to extend the results presented here to languages with reflexive types, perhaps by adapting one of the semantics-based proofs mentioned above.

A possible equivalence between monads and CPS was conjectured by Danvy and Filinski [3] and partially fleshed out by Wadler [32], but even the latter was

quite informal – since the result generalizes Meyer and Wand's, one would expect the proof to be at least as complicated. Another glimmer of the correspondence can be seen in Sabry and Felleisen's result [24] that $\beta\eta$-equivalence of CPS terms coincides with direct-style equivalence in Moggi's computational λ-calculus [14], which captures exactly the equivalences that hold in the presence of arbitrary monadic effects. Peyton Jones and

Wadler [18] probe the relationship between monads and CPS further, and Wadler [33] analyzes composable continuations from a monadic perspective, but in both cases the restriction to Hindley-Milner typability obscures the deeper connections.

"Composable continuations" have also been studied by a number of researchers [10, 8, 3]. Many of these constructs depend on explicit support from the compiler or runtime system, such as the ability to mark or splice together delimited stack segments. However, an encoding in standard Scheme of one variant was devised by Sitaram and Felleisen [26]. The embedding is fairly complex, relying on dynamically-allocated, mutable data structures, `eq?`-tests, and with no direct con-

nection to a formal semantics of the constructs. Yet another Scheme-implementable notion of partial continuations was proposed by Queinnec and Serpette [20]; the code required is perhaps even more intricate. (To be fair, both of these constructs are apparently more general than shift/reset, though the practical utility of this additional power remains to be seen.) The much simpler construction presented in this paper uses only a single cell holding a continuation, and is directly derived from the denotational definition of shift and reset.

Finally, recent work by Riecke [23] on effect delimiters may be somehow related to the present paper, as they share several concepts and techniques (specifically, monads, prompts, and retractions). On closer inspection, though, the similarities become much less apparent (for example, Riecke only considers a few specific monads and attaches no special significance to CPS); certainly the specific goals of the two papers are quite different, and the results obtained seem incomparable. Still, there might be some deeper connections to uncover, and the subject is probably worth exploring further.

7 Conclusions

By exploiting the correspondence between monadic and continuation-passing styles, we can embed any definable monad into a language with a "composable continuations" construct. Further, such a construct can itself be decomposed into ordinary first-class continuations and a storage cell. Thus, it is possible in principle to express any definable monadic effect as a combination of control and state. In practice, of course, many such effects – including, obviously, call/cc and ref-cells themselves – can be more naturally expressed directly, without the detour over composable continuations.

However, the construction presented here should still be of some practical use in experiments with, and rapid prototyping of, more complicated monadic structures. The embedding approach does not incur the interpretive overhead of a "monadic interpreter" or the complexity of an explicit source-to-source "monadic translation" step. And perhaps even more importantly, it allows us to retain with no extra effort all the conveniences of the original language, including pattern-matching, static typechecking, and module system. The efficiency of the general embedding may not be quite as good as hand-coded monadic style specialized to a particular monad, especially since many compilers do not attempt to track

continuations across storage cells. On the other hand, if effects are rare, programs run at full speed without the overhead of explicitly performing the administrative manipulations specified by η and $-^*$, such as tagging and checking return values in the exception monad.

The embedding result is also a strong argument for inclusion of first-class continuations in practical eager languages, especially ones like ML that already have mutable cells: providing call/cc does not simply add "yet another monadic effect" – it completes the language to *all* such effects! Moreover, a sophisticated module system like SML's lets us expose as little or as much of this underlying raw power as we need: by picking the appropriate monadic structure, we can introduce effects ranging from simple exceptions to full composable continuations.

But surely there is more to "functional programming with escapes and state" than monadic effects. After all, monads provide only the lowest-level framework for sequencing computations; in practical programs, we often need tools for expressing higher-level, application-oriented abstractions. A strict monad-based partitioning of effects may be adequate in many cases, but monads cannot and should not take place of a proper module

facility. In fact, it might be that the syntactic "noise" due to writing everything in monadic (or any other) style makes it *harder* to recognize and exploit organizational units that do not conveniently fit into the monadic mold (for example, concurrency packages like Reppy's CML [21], or "imperative unification" using mutable data structures).

The present work also sheds some light on the problem of integrating individual monads to express composite effects. Briefly, the complication is that a monad by itself is a closed package that contains too little information: we need instead to express the monadic data as an *increment* to be layered on top of other possible effects. How to do this uniformly is still not quite clear; Moggi's *monad constructors* [15] and Steele's *pseudomonads* [29] are two possible techniques. In the composable-continuations characterization of monads, monad combination seems to correspond to also letting the *target* language of the defining translations contain monadic effects, leading to the hierarchy of control operators and the associated meta^n-continuation-passing style introduced in [3] and further investigated by Murthy [17].

However, such approaches all lead to an inherently

"vertical" or "hierarchical" notion of monad composition, because in general we must answer such questions as "should backtracking undo I/O" or "should exceptions undo state mutation" (and perhaps also, "is this really the right way to think about supposedly functional programs") Yet many monadic effects can in fact be naturally combined in a "horizontal" or "independent" way, such as different pieces of state, or storage and I/O; both the monadic and the (generalized) CPS formulation seem awkward in such cases, but individually mutable cells capture this situation directly.

Much recent work on monads in "purely functional" languages vs. control and state in an "imperative functional" setting seems largely disjoint. Perhaps the connections outlined in this paper can lead to some cross-fertilization and help avoid duplication of effort. For example, "pure" functional programmers might benefit from work on organizing and reasoning about first-class continuations and storage cells in the "imperative" setting (e.g., [7]); noting that these are monadic effects is clearly not sufficient to actually reason about them. Conversely, results about algebraic properties of partic-

ular monads (e.g., [11]) could be useful for recognizing and exploiting patterns of continuation and state usage in eager languages.

Acknowledgments

I want to thank John Reynolds for support, and Olivier Danvy, Matthias Felleisen, Julia Lawall, Greg Morrisett, Amr Sabry, Phil Wadler, and the reviewers for their helpful comments on various drafts of this paper.

References

[1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991.

[2] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 4(3):1–55, July 1991.

[3] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990.

[4] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[5] Bruce F. Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, Orlando, Florida, January 1991.

[6] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988.

[7] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, September 1992.

[8] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988.

[9] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Languages Design and Implementation*, pages 66–77, White Plains, New York, June 1990.

[10] Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 158–168, San Diego, California, January 1988.

[11] David J. King and Philip Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow 1992*, pages 134–143, Ayr, Scotland, 1993. Springer-

Verlag.

[12] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. Indiana University and Aarhus University. Personal communication, October 1993.

[13] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, June 1985.

[14] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE.

[15] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, April 1990.

[16] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[17] Chetan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In *Proceedings of the ACM SIGPLAN Workshop on Continuations*, pages 49–71, San Francisco, California, June 1992. (Technical Report No. STAN-CS-92-1426, Department of Computer Science, Stanford University).

[18] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 71–84, Charleston, South Carolina, January 1993.

[19] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[20] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 174–184, Orlando, Florida, January 1991.

[21] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, Toronto, Canada, June 1991.

[22] John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *2nd Colloquium on Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 141–156, Saarbrücken, West Germany, July 1974.

[23] Jon G. Riecke. Delimiting the scope of effects. In *Functional Programming Languages and Computer Architecture 1993*, pages 146–155, Copenhagen, Denmark, June 1993. ACM Press.

[24] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 288–298, San Francisco, California, June 1992. Revised version to appear in *Lisp and Symbolic Computation*.

[25] Ravi Sethi and Adrian Tang. Constructing call-by-value continuation semantics. *Journal of the ACM*, 27(3):580–597, July 1980.

[26] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.

[27] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 161–175, Nice, France, June 1990.

[28] Brian C. Smith. *Reflection and Semantics in a Procedural Language.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1982. MIT-LCS-TR-272.

[29] Guy L. Steele, Jr. Building interpreters by composing monads. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. (To appear).

[30] Joseph E. Stoy. The congruence of two program-

ming language definitions. *Theoretical Computer Science*, 13(2):151–174, February 1981.

[31] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, December 1992. (An earlier version appeared in *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*).

[32] Philip Wadler. The essence of functional programming (invited talk). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992.

[33] Philip Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, 1994. (To appear).

[34] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1), May 1988.