Compiling Monads *

Olivier Danvy, Jürgen Koslowski, and Karoline Malmkjær
Department of Computing and Information Sciences
Kansas State University †
(danvy, koslowj, karoline)@cis.ksu.edu

December 1991

Abstract

Computational monads offer a powerful way to parameterize functional specifications, but they give rise to exceedingly tedious simplifications to instantiate this "monadic" interpreter. We report on the use of partial evaluation to achieve the following instantiations automatically.

- We derive equivalent formulations of the monadic λ-interpreter, based on equivalent specifications of monads from algebra and category theory (join, clone, exp, and bind). This increases flexibility in formulating the parameterized interpreter.
- We derive several well-known styled interpreters by instantiating the monadic interpreter with a particular monad (store, continuation, *etc.*). This illustrates the generality of the monadic interpreter.
- We specialize the monadic interpreter with respect to a particular program, thereby compiling this program into a parameterized representation of its meaning (a.k.a. a monadic form). This makes it possible to reason about particular programs independently of their interpretation and also to measure the generality of the monadic interpreter.

From a computational viewpoint, partial evaluation makes it possible for each of these instantiations to run significantly faster.

In the spirit of the Scheme programming language, our Kleisli interpreter handles a fixed set of special forms. Therefore, the natural way to extend it is to add predefined functions in the initial environment, using Church encoding (**call/cc** instead of **escape**, *etc.*) if necessary. We study the consequences of this design. Equipping the exception monad with the usual operators forces us to adopt a new representation of arrows and therefore to modify the interpreter in a way reminiscent of call-by-name, even though the defined language is still call-by-value. Instantiating this new interpreter with the continuation monad yields the lesser-known continuation-passing style advocated by Reynolds.

Finally, we relate the parameterization offered by monads with other ways of structuring a formal semantics, either for the purpose of flow analysis by abstract interpretation or for the purpose of semantics-directed compiling and compiler generation.

Keywords

Computational monads, Kleisli λ-interpreter, Reynolds λ-interpreter, partial evaluation, Scheme

*Technical report CIS-92-03, Department of Computing and Information Sciences, Kansas State University.

†Manhattan, Kansas 66506, USA. (913) 532-6350. Part of this work was supported by NSF under grant CCR-9102625.

1 Introduction

Formal semantics of programming languages seem to have a side-effect: their expressive power spawns programming languages mimicking their metalanguage. Denotational semantics gave birth to Scheme, SML, and Haskell; algebraic semantics, to OBJ; logic semantics, of course, to Prolog; operational semantics (a.k.a. natural semantics), to Typol. Action semantics should give rise to some new programming language soon. These languages arise due to the need to execute semantic specifications.

Categorical semantics seems to have the same goal in formal semantics as category theory does in mathematics: categorical semantics aims at structuring the expressive power of our programming languages in a coherent and unified way. Moggi's work on computational monads offers a striking illustration. With a simple algebraic structure generalizing monoids, *monads*, Moggi is able to model amazingly many programming language features: side-effects, exceptions, continuations, interactive input and output, and non-determinism, to name a few [18].

Monads can also be used for a modular approach to denotational semantics [17]. A monadic denotational semantics definition can be transliterated directly into a functional language. Wadler set out to do this with Haskell and came up with a new notation for "comprehending" monads and with an exemplary definitional λ-interpreter structured with monads [27, 28].

λ-interpreters have a long-standing tradition in the Lisp world. This tradition originates with McCarthy's meta-circular definition of Lisp and was actively pursued with the blossoming of artificial intelligence languages (Planner, Conniver, Scheme, Intrigue), a direction that progressively departed from artificial intelligence to stand on its own with Steele and Sussman's "Art of the Interpreter" [25]. But interpreters have a runtime price and this triggered a direction of research activity on how to transform interpreters to compilers [7], paralleling research on denotational semantics-directed compiler generation [14]. These two research areas today appear to coincide, both in theory and in practice, with partial evaluation [10].

1.1 This paper

We structure and parameterize a Scheme λ-interpreter using monads and we are able to remove several interpretive overheads using partial evaluation.

In the spirit of Scheme, the interpreter is designed to work on a fixed BNF, *i.e.*, with a fixed set of special forms. The defined language is a subset of Scheme. It may be extended by introducing new procedures in the initial environment. This makes it possible to keep the same monadic interpreter for all the possible monads.

On the other hand, the denotation of procedures can be represented in two distinct ways, leading to a "Kleisli interpreter" and a "Reynolds interpreter", as addressed in Sections 2 and 3.

Figure 1 displays an interactive session with our parameterized monadic interpreters. We successively boot the Kleisli and the Reynolds interpreters with the state monad, the continuation monad, the continuation & state monad, and the exception monad (see below for their description). Then we evaluate a few illuminative Scheme expressions, using the extensions corresponding to the different monads.

Given a particular monad, the interpreter can be simplified into a familiar "styled" interpreter. A continuation monad yields a continuation-passing style interpreter; a state monad yields a state-passing style interpreter; and so on. These instantiations are documented in Moggi's and Wadler's published work [17, 28]. For lack of space, they are not reproduced here.

Given a particuliar program, the interpreter can be simplified into the monadic form of this program.

```
> (boot-kleisli state-monad)                      ;;; pure Scheme + get & set
Kleisli interpreter -- join mode
state> (add1 (get))
(1 . 0)
state> (set 3)
(void . 3)
state> (begin (set 3) (let ([x (get)]) (begin (set 4) (* x x))))
(9 . 4)
state> ^D
> (boot-kleisli continuation-monad)              ;;; pure Scheme + call/cc
Kleisli interpreter -- join mode
continuation> (+ 10 (call/cc (lambda (k) (add1 (k 1)))))
11
continuation> ^D
> (boot-kleisli continuation+state-monad)        ;;; pure Scheme + call/cc + get & set
Kleisli interpreter -- join mode
continuation+state> (set (+ 10 (call/cc (lambda (k) (add1 (k 1))))))
(void . 11)
continuation+state> (begin (set 3) (call/cc (lambda (k) (begin (set 4) (k 9)))))
(9 . 4)
continuation+state> ^D
> (boot-reynolds exception-monad)                ;;; pure Scheme + raise & catch
Reynolds interpreter -- join mode
exception> (+ 10 20)
(expected . 30)
exception> (+ 10 (raise))
(excepted . void)
exception> (+ 10 (begin (catch (/ (raise) 0)) 20))
(expected . 30)
exception> ^D
>
```

Figure 1: An interactive Scheme session with the Kleisli and Reynolds interpreters

As a simple example, the expression `(lambda (x) x)` is compiled into the following monadic form

```
(unit (lambda (actuals) (unit (car actuals))))
```

as a specialized instance of the monadic interpreter.[1]

In turn, for each monad, monadic forms can be simplified. For example, given the continuation monad (see below), the monadic form above can be simplified as follows.

```
(lambda (k) (k (lambda (actuals) (lambda (k) (k (car actuals))))))
```

which is the familiar CPS counterpart of the identity procedure in an arbitrary context.

Finally, a monad can be specified in many equivalent ways. This gives rise to equivalent formulations of the parameterized interpreter. Given a particular monad, these formulations get instantiated into the same styled λ-interpreter.

Partial evaluation (a.k.a. program specialization) makes it possible to carry out all these instantiations, specializations, and reformulations automatically.

[1]The interpreter represents a n-ary procedure as a unary function taking a list of n arguments. Here `actuals` denotes a list of one element.

1.2 Background and related work

1.2.1 Monads

Monads (also known as "triples") were invented by category theorists in the late 50's (under the name "standard constructions") and became popular in various areas of category theory in the 60's. In particular, much of universal algebra can be formulated in terms of monads [16]. It soon became clear that monads are in fact a 2-categorical concept, not restricted to the 2-category of categories, functors, and natural transformations [26].

Here we are interested in monads in an enriched context, so-called *strong monads* over a symmetric monoidal closed category $\mathcal{X}$. Such an $\mathcal{X}$ is equipped with

- a functor $\mathcal{X} \times \mathcal{X} \xrightarrow{\otimes} \mathcal{X}$ (called *a tensor product*) that is associative, *i.e.*, $(X \otimes Y) \otimes Z$ and $X \otimes (Y \otimes Z)$ are naturally isomorphic, and symmetric, *i.e.*, $X \otimes Y$ and $Y \otimes X$ are naturally isomorphic;
- a distinguished object I such that $I \otimes X$ and $X \otimes I$ are naturally isomorphic to X;

subject to certain *coherence axioms* [13]. In addition, we have

- a functor $\mathcal{X}^{\mathrm{op}} \times \mathcal{X} \xrightarrow{[\_ \to \_]} \mathcal{X}$ such that for every object Y the functor $\mathcal{X} \xrightarrow{Y \otimes -} \mathcal{X}$ is left adjoint to $\mathcal{X} \xrightarrow{[Y \to \_]} \mathcal{X}$, *i.e.*, there is a natural isomorphism between the sets $\mathcal{X}\langle X \otimes Y, Z\rangle$ and $\mathcal{X}\langle Y, [X \to Z]\rangle$. The counit of the adjunction yields an internal version of *application* $[X \to Y] \otimes X \xrightarrow{@_Y^X} Y$.

Notice that the hom-sets $\mathcal{X}\langle X, Z\rangle$ and $\mathcal{X}\langle I, [X \to Z]\rangle$ are isomorphic. The object $[X \to Z]$ should be viewed as an internal version of the external hom-set $\mathcal{X}\langle X, Z\rangle$. Hence the external composition functions $\mathcal{X}\langle Y, Z\rangle \times \mathcal{X}\langle X, Y\rangle \xrightarrow{\circ} \mathcal{X}\langle X, Z\rangle$ and the external identity arrows $1 \xrightarrow{id_X} \mathcal{X}\langle X, X\rangle$ can be internalized in terms of $\mathcal{X}$-arrows $[Y \to Z] \otimes [X \to Y] \xrightarrow{\odot} [X \to Z]$ and $I \xrightarrow{1_X} [X \to X]$, respectively.

Just like the cartesian closedness of ***Set*** enables one to define ordinary categories, functors, and natural transformations, the monoidal closedness of $\mathcal{X}$ allows us to define the corresponding notions relative to $\mathcal{X}$ [13]. In this paper we only need a fragment of this theory. An endofunctor $\mathcal{X} \xrightarrow{T} \mathcal{X}$ on $\mathcal{X}$ is called *closed* if its action on arrows admits internalization, *i.e.*, if there exists a family of $\mathcal{X}$-arrows $[X \to Y] \xrightarrow{map^T_{X \to Y}} [TX \to TY]$ compatible with internal composition and internal identity arrows via

$$\begin{aligned} map^T_{X \to Y} \circ \odot &= \odot \circ (map^T_{Y \to Z} \otimes map^T_{X \to Y}) \\ map^T_{X \to X} \circ 1_X &= 1_{TX} \end{aligned}$$

Notice that the action of T on external arrows can be recovered from the object-function of T (often called a *type constructor*) and the family map^T that specifies the action of T on internal arrows. It is possible to replace this family by a *tensorial strength*, *i.e.*, a family of $\mathcal{X}$-arrows $X \otimes TY \xrightarrow{\vartheta_{X,Y}} T(X \otimes Y)$ subject to suitable conditions. However, here we prefer the formulation in terms of map^T.

If S is another closed endofunctor on $\mathcal{X}$, then a *closed natural transformation* r from S to T consists of a family of $\mathcal{X}$-arrows $I \xrightarrow{r_X} [[X \to Y] \to [SX \to TX]]$ such that

$$[SX \to r_Y] \circ map^S_{X \to Y} = [r_X \to TY] \circ map^T_{X \to Y}$$

where $[SX \to r_Y]$ and $[r_X \to TY]$ denote the internal composition with r_Y from the left, and with r_X from the right, respectively.

A *strong monad* $\boldsymbol{T} = \langle T, map^T, unit^T, join^T \rangle$ on $\mathcal{X}$ now consists of

- a type constructor $\boldsymbol{Ob}(\mathcal{X}) \xrightarrow{T} \boldsymbol{Ob}(\mathcal{X})$;
- a family of $\mathcal{X}$-arrows map^T that makes T into a strong functor;
- two strong natural transformations $id_{\mathcal{X}} \xrightarrow{unit^T} T$ and $TT \xrightarrow{join^T} T$;

subject to the following axioms:

- associativity: $join^T_X \circ join^T_{TX} = join^T_X \circ map^T_{TTX \to TX}(join^T_X)$
- two-sided unit: $join^T_X \circ unit^T_{TX} = 1_{TX} = join^T_X \circ map^T_{X \to TX}(unit^T_X)$

There exist various equivalent formulations of this concept, which replace the strong natural transformation $join^T$ by different data. Some of these are discussed in Section 5.

Each strong monad $\boldsymbol{T}$ on $\mathcal{X}$ gives rise to a category $\mathcal{X}_{\boldsymbol{T}}$ enriched in $\mathcal{X}$, called the *Kleisli-category* of $\boldsymbol{T}$, with the same objects as $\mathcal{X}$, and the internal hom-object $[X \to Y]_{\boldsymbol{T}}$ given by $[X \to TY]$. The internal composition $\diamond$ in $\mathcal{X}_{\boldsymbol{T}}$ is defined by the diagram

$$\begin{array}{rcl} [Y \to TZ] \otimes [X \to TY] & \xrightarrow[\circ]{} & [X \to TZ] \\ {\scriptstyle map^T_{Y \to TZ} \otimes id} \downarrow & & \uparrow {\scriptstyle [X \to join^T_Z]} \\ [TY \to TTZ] \otimes [X \to TY] & \xrightarrow[\odot]{} & [X \to TTZ] \end{array}$$

1.2.2 Computational monads

Moggi showed how the formal semantics of a programming language could be structured using monads [18]. Wadler proposed a notation reminiscent of set and list comprehensions to tame the use of monads in the realm of lazy functional programming [27].

We found the following instantiations particularly interesting. For example, writing a λ-expression in monadic form and instantiating it with the continuation monad yields the continuation-passing counterpart of this λ-expression. For another example, instantiating a monadic λ-interpreter with the continuation monad yields a continuation-passing λ-interpreter.

However, we were disheartened by the tedious simplification steps such instantiations of monads require. The present paper reports our contribution to automate this simplification with partial evaluation.

1.2.3 Monadic λ-interpreters

In his investigation of duality in programming language semantics, Filinski noted the connection between his work and the Kleisli category of the continuation monad [8]. It is still an open problem, though, to specify Filinski's symmetric λ-calculus using a monad.

In March 1990 at MIT (personal communication to the first author), Rees had developed a monadic Lisp-like λ-evaluator similar in structure to ours, in that (1) it is written in Scheme; (2) it has a fixed set of special forms; and (3) it is extensible through functions in the initial environment. Our λ-interpreter subsumes Rees's Kleisli interpreter in that it handles a few more special forms, including letrec, and it is

interactive. Unlike Rees, though, we reformulated our Kleisli interpreter into a Reynolds interpreter to provide the usual operators handling exceptions (*cf.* Section 3).

Wadler has also investigated the use of monads to structure functional programs and their relation to continuation-passing style [28]. Wadler's interpreter is written in Haskell and deliberately uses the lazy and statically typed features of this language. His emphasis is to keep the interpreter small. Extending the language is achieved by introducing new special forms, *i.e.*, new lines in the BNF of the language, which therefore forces one to alter the core of the interpreter. In contrast, our core is fixed and one extends the language with predefined functions in the initial environment. Also, we use Scheme, an eager and dynamically typed λ-language [3]. We chose Scheme for two reasons: Scheme is an extraordinarily neat programming language; and more practically, several partial evaluators are available for Scheme.

1.2.4 Partial evaluation

Partial evaluation is a program transformation technique aimed at specializing a program with respect to part of its input and producing specialized programs that are usually faster than the source ones. As such, partial evaluation is a natural choice for instantiating an interpreter with a monad, for specializing an interpreter with respect to a program, and for translating an interpreter from one monadic form to another.

The first and the third of these instantiations are new. Specializing an interpreter with respect to a program is known as the first Futamura projection [9]. It amounts to compiling the program from the defined language to the defining language of the interpreter. This compilation can be optimized in various ways, essentially by self-applying the partial evaluator (second and third Futamura projections) [12, 1, 6]. Correspondingly, our instantiations can also be optimized.

We are using Consel's self-applicable partial evaluator Schism [4, 5].

1.3 Overview

Section 2 presents our Kleisli interpreter and a few monads. Failure to extend this λ-interpreter with a usual exception operator leads us to an alternate representation of arrows and our Reynolds interpreter, described in Section 3. Section 4 presents the two applications of partial evaluation offered by the parameterized interpreter. Section 5 reviews equivalent formulations of a monad and the corresponding translations using partial evaluation. Section 6 presents conclusions and new issues.

2 Kleisli Interpreter

We want to evaluate the Scheme-like expressions specified by the following BNF.

$$
\begin{aligned}
&e ::= c \mid i \mid l \mid e_0\,(e_1,\,...,\,e_n) \mid e_1 \;\rightarrow\; e_2,\; e_3 \\
&\qquad \mid \text{let}\,(i_1,\,...,\,i_n) = (e_1,\,...,\,e_n)\,\text{in}\,e_0 \mid \text{letrec}\,(i_1,\,...,\,i_n) = (l_1,\,...,\,l_n)\,\text{in}\,e_0 \mid e_0;\,...\,;e_m \\
&l ::= \lambda\,(i_1,\,...,\,i_n)\,.\,e
\end{aligned}
$$

Moggi interprets programs as arrows of the Kleisli category $\mathcal{X}_T$, for an arbitrary monad T. Our interpreter mimics this approach by representing λ-abstractions of type $A \rightarrow B$ as elements of the exponent $[A \rightarrow TB]$, where T is the type constructor of the monad, as defined in Section 1.2.1.

The meaning of each syntactic construct is determined using a combination of monad operations.

$$
\begin{array}{rcl}
\mathcal{M} & : & \mathrm{Exp} \otimes \mathit{Env} \to T(\mathit{Val}) \\
\mathcal{W} & : & \mathrm{List}(\mathrm{Exp}) \otimes \mathit{Env} \to T(\mathrm{List}(\mathit{Val})) \\
\mathit{Env} & = & \mathrm{Ide} \to \mathit{Val} \\
\mathit{Val} & = & \mathit{Bool} + \mathit{Num} + \mathit{String} + (\mathit{Val} \times \mathit{Val}) + \mathit{Proc} \\
\mathit{Proc} & = & [\mathrm{List}(\mathit{Val}) \to T(\mathit{Val})]
\end{array}
$$

The initial environment binds the usual Scheme predefined procedures. Here is a section of our Kleisli interpreter.

$$
\begin{array}{rcl}
\mathcal{M}\,[\![i]\!]\,\rho & = & \mathit{unit}(\rho\, i) \\
\mathcal{M}\,[\![\lambda\,(i_1,\,...,\,i_n)\,.\,e]\!]\,\rho & = & \mathit{unit}(\lambda\,(v_1,\,...,\,v_n)\,.\,\mathcal{M}\,[\![e]\!]\,\rho[i_1 \mapsto v_1,\,...,\,i_n \mapsto v_n]) \\
\mathcal{M}\,[\![e_0\,(e_1,\,...,\,e_n)]\!]\,\rho & = & \mathit{join}(\mathit{map}\,(\lambda\, f\,.\,\mathit{join}(\mathit{map}\, f\,(\mathcal{W}\,[\![(e_1,\,...,\,e_n)]\!]\,\rho))) \\
 & & \qquad\qquad (\mathcal{M}\,[\![e_0]\!]\,\rho)) \\
\mathcal{M}\,[\![e_1 \to e_2,\, e_3]\!]\,\rho & = & \mathit{join}(\mathit{map}(\lambda\, b\,.\, b \to \mathcal{M}\,[\![e_2]\!]\,\rho,\, \mathcal{M}\,[\![e_3]\!]\,\rho)(\mathcal{M}\,[\![e_1]\!]\,\rho)) \\
 & ... &
\end{array}
$$

Here, *map*, *unit*, and *join* are functions corresponding to the families of arrows given by the monad, as defined in Section 1.2.1.

Now we are ready to specify some monads.

For any domain *State*, Figure 2 displays the corresponding state monad. Using this monad equips the interpreter with a state. Now we can extend the interpreted language with operations over the state. As an example, we consider a state that holds one value. We can read and update it using two operators *get* and *set*. These operators are bound in the initial environment.

For any domain of answers, Figure 3 displays the continuation monad. Using this monad equips the interpreter with an explicit continuation. Now we can extend the interpreted language with control operators, for example with Scheme's `call/cc`. This operator is bound in the initial environment.

Let us equip the continuation monad with a state. This is simple: we just refine the domain of answers, given some other domain of answers Answer'.

$$\mathit{Answer} \;=\; [\mathit{State} \to \mathit{Answer}']$$

The actual definition of `call/cc` does not change since a control operation is independent of the state (σ gets η-reduced). Figure 4 displays the new state operators and their functionalities.

For any domain of exceptions, Figure 5 displays the exception monad. Using this monad equips the interpreter with an explicit exception mechanism. Now we can extend the interpreted language with operations to abort the current computation.

However, we cannot introduce an operator to catch an exception. Such an operator would need to receive the result $A + \mathit{Exception}$ in order to determine whether it is an exception result to be handled or a proper result in A to be passed along. Since we, however, require all extensions to be defined as functions in the environment, they must be of type $[A \to EB]$ for some A and B. They will then be applied using the specification of application in the interpreter. Therefore, if an exception is raised, the function will never be applied — instead $\mathit{map}^E_{A \to EB}$ will simply pass the exception along.

We could circumvent this problem with thunks (*i.e.*, parameterless procedures) in the defined language. However, we can also adopt another representation of λ-abstractions, as investigated in the following section.

$$
\begin{array}{rcl}
SA & = & [State \to A \times State] \\
map^S_{A\to B} & = & \lambda f \,.\, \lambda \widehat{a} \,.\, \lambda \sigma \,.\, \textbf{let}\ (a, \sigma') = \widehat{a}\ \sigma\ \textbf{in}\ (f\ a, \sigma') \\
unit^S_A & = & \lambda a \,.\, \lambda \sigma \,.\, (a, \sigma) \\
join^S_A & = & \lambda \widehat{\widehat{a}} \,.\, \lambda \sigma \,.\, \textbf{let}\ (\widehat{a}, \sigma') = \widehat{\widehat{a}}\ \sigma\ \textbf{in}\ \widehat{a}\ \sigma'
\end{array}
$$

$$
\begin{array}{rcll}
\texttt{get} & \mapsto & \lambda () \,.\, \lambda \sigma \,.\, (\sigma, \sigma) & : \ \mathrm{List}(A) \to [State \to A \times State] \\
\texttt{set} & \mapsto & \lambda (a) \,.\, \lambda \sigma \,.\, (\text{`}void\text{'}, a) & : \ \mathrm{List}(A) \to [State \to A \times State]
\end{array}
$$

Figure 2: State monad and two state operators for the Kleisli interpreter

$$
\begin{array}{rcl}
CA & = & [[A \to Answer] \to Answer] \\
map^C_{A\to B} & = & \lambda f \,.\, \lambda \widehat{a} \,.\, \lambda \kappa \,.\, \widehat{a}\ (\lambda a \,.\, \kappa\ (f\ a)) \\
unit^C_A & = & \lambda a \,.\, \lambda \kappa \,.\, \kappa\ a \\
join^C_A & = & \lambda \widehat{\widehat{a}} \,.\, \lambda \kappa \,.\, \widehat{\widehat{a}}\ (\lambda \widehat{a} \,.\, \widehat{a}\ \kappa)
\end{array}
$$

$$
\texttt{call/cc} \ \mapsto \ \lambda f \,.\, \lambda \kappa \,.\, f\ (\lambda (a) \,.\, \lambda \kappa' \,.\, \kappa\ a)\ \kappa \ : \ \mathrm{List}(A) \to [[A \to Answer] \to Answer]
$$

Figure 3: Continuation monad and a control operator for the Kleisli interpreter

$$
\begin{array}{rcll}
\texttt{get} & \mapsto & \lambda () \,.\, \lambda \kappa \,.\, \lambda \sigma \,.\, \kappa\ \sigma\ \sigma & : \ \mathrm{List}(A) \to [[A \to [State \to Answer']] \to [State \to Answer']] \\
\texttt{set} & \mapsto & \lambda (a) \,.\, \lambda \kappa \,.\, \lambda \sigma \,.\, \kappa\ \text{`}void\text{'}\, a & : \ \mathrm{List}(A) \to [[A \to [State \to Answer']] \to [State \to Answer']]
\end{array}
$$

Figure 4: Operators for the continuation-state monad for the Kleisli interpreter

$$
\begin{array}{rcl}
EA & = & A + Exception \\
map^S_{A\to B} & = & \lambda f \,.\, \lambda \widehat{a} \,.\, \textbf{case}\ \widehat{a}\ \textbf{of} \\
 & & \quad isExpected(a) \ \to \ inExpected(f\ a) \\
 & & \quad \mid \ isExcepted() \ \to \ \widehat{a} \\
 & & \quad \textbf{end} \\
unit^S_A & = & \lambda a \,.\, inExpected(a) \\
join^S_A & = & \lambda \widehat{\widehat{a}} \,.\, \textbf{case}\ \widehat{a}\ \textbf{of} \\
 & & \quad isExpected(a) \ \to \ a \\
 & & \quad \mid \ isExcepted() \ \to \ inExcepted() \\
 & & \quad \textbf{end}
\end{array}
$$

$$
\texttt{raise} \ \mapsto \ \lambda () \,.\, inExcepted() \ : \ \mathrm{List}(A) \to A + Exception
$$

Figure 5: Exception monad and an exception operator for the Kleisli interpreter

3 Reynolds Interpreter

Our Reynolds interpreter represents λ-abstractions of type $A \to B$ as an element of the exponent $[TA \to TB]$. This representation of functions seems to correspond to the ideas motivating Brian Smith's reconstruction of Lisp [24]. During a computation, one never obtains actual values but only representations of values. Thus the denotation of a procedure (computing a function $A \to B$) expects a representation of value TA rather than a value A and returns a representation of value TB.

As before, the meaning of each syntactic construct is determined as a combination of monad operations.

$$\begin{array}{rcl}
\mathcal{M} & : & \mathrm{Exp} \otimes \mathit{Env} \to T(\mathit{Val}) \\
\mathcal{W} & : & \mathrm{List}(\mathrm{Exp}) \otimes \mathit{Env} \to T(\mathrm{List}(\mathit{Val})) \\
\mathit{Env} & = & \mathrm{Ide} \to \mathit{Val} \\
\mathit{Val} & = & \mathit{Bool} + \mathit{Num} + \mathit{String} + (\mathit{Val} \times \mathit{Val}) + \mathit{Proc} \\
\mathit{Proc} & = & [T(\mathrm{List}(\mathit{Val})) \to T(\mathit{Val})]
\end{array}$$

The initial environment binds the usual Scheme predefined procedures.

In Wadler's work, this representation of λ-abstractions entails call-by-name [28], but this does not have to be so. We can still get call-by-value as follows.

$$\begin{array}{rcl}
\mathcal{M}\,[\![i]\!]\,\rho & = & \mathit{unit}(\rho\, i) \\
\mathcal{M}\,[\![\lambda\,(i_1, ..., i_n)\,.\,e]\!]\,\rho & = & \mathit{unit}(\lambda\,(t_1, ..., t_n)\,.\,\mathit{join}(\mathit{map}\,(\lambda\,(v_1, ..., v_n)\,.\,\mathcal{M}\,[\![e]\!]\,\rho[i_1 \mapsto v_1, ..., i_n \mapsto v_n]) \\
 & & \qquad\qquad (t_1, ..., t_n))) \\
\mathcal{M}\,[\![e_0\,(e_1, ..., e_n)]\!]\,\rho & = & \mathit{join}(\mathit{map}\,(\lambda\, f\,.\, f\,(\mathcal{W}\,[\![(e_1, ..., e_n)]\!]\,\rho)) \\
 & & \qquad\qquad (\mathcal{M}\,[\![e_0]\!]\,\rho)) \\
 & ... &
\end{array}$$

Now we can equip the monad for exceptions with an operator to catch exceptions (*cf.* Figure 6). The Reynolds interpreter makes it possible to do it without resorting to thunks in the defined language.

$$\begin{array}{lcll}
\texttt{raise} & \mapsto & \lambda\,()\,.\,\mathit{inException}(\text{`void'}) & : \; E(\mathrm{List}(A)) \to A + \mathit{Exception} \\
\texttt{catch} & \mapsto & \lambda\,\widehat{a}\,.\,\textbf{case}\ \widehat{a}\ \textbf{of} & : \; E(\mathrm{List}(A)) \to A + \mathit{Exception} \\
 & & \quad \mathit{isExpected}(a) \to \widehat{a} & \\
 & & \quad \mid \mathit{isExcepted}(\text{`void'}) \to \mathit{inExpected}(\text{`void'}) & \\
 & & \quad \textbf{end} &
\end{array}$$

Figure 6: Two exception operators for the Reynolds interpreter

The next section describes the instantiation of the Kleisli and of the Reynolds interpreters. In particular, instantiating the Reynolds interpreter with the continuation monad motivates the name of this interpreter.

4 Instantiating the λ-Interpreters

4.1 With a monad

Using partial evaluation, we have instantiated the parameterized interpreters for each monad, automatically. For the continuation and the state monads, this instantiation yielded the usual continuation-passing and state-passing style interpreters, respectively.

Specializing the Reynolds interpreter with respect to the continuation monad yielded an interpreter that coincided with Reynolds's CPS interpreter in an early paper [22] — hence the name "Reynolds interpreter."

4.2 With a program

We can also specialize the parameterized interpreters with respect to a program, compiling this program into a monadic form. This allows us to produce the denotation of a program, parameterized with the monad.

This relies on the fact that specializing a monadic interpreter M with respect to a program p maps this program into a monadic form p_m:

$$\text{run } PE\langle M, p\rangle \quad = \quad p_m$$

4.3 With a λ-interpreter

Our monadic λ-interpreters clearly generalize the usual Scheme interpreters. To determine whether this generalization is conservative, we propose the following experiment.

Consider a Scheme interpreter I written in Scheme. Specializing a monadic interpreter M with respect to I maps I into a monadic form I_m, as above:

$$\text{run } PE\langle M, I\rangle \quad = \quad I_m$$

Now, I_m is a Scheme interpreter in monadic form.

If M truly generalizes a Scheme interpreter, then I_m and M should look pretty much alike. We have not yet had the time to perform this experiment. We are optimistic about the result, however, since specializing M with respect to the trivial (identity) monad yields I.

5 Reformulating the monads

Ordinary monads (involving plain categories, functors, and natural transformations) capture essential parts of universal algebra; hence Manes also calls them *algebraic theories in monoid form* [16]. He shows two other types of algebraic theories to be equivalent to monads, and this equivalence does in fact carry through to the enriched context. Moreover, a fourth formulation becomes available (Wadler's `bind`) that is particularly appealing from a computer science point of view because it closely matches continuation-passing style [28].

All these formulations feature (i) a type constructor $\boldsymbol{Ob}(\mathcal{X}) \xrightarrow{T} \boldsymbol{Ob}(\mathcal{X})$ together with a family map^T satisfying the axioms of Section 1.2.1 (*i.e.*, a strong endofunctor $\mathcal{X} \xrightarrow{T} \mathcal{X}$) and ($ii$) a strong natural transformation $1_{\mathcal{X}} \xrightarrow{unit^T} T$.

The strong natural transformation $TT \xrightarrow{join^T} T$ may alternatively be replaced by

clone form: a family of $\mathcal{X}$-arrows $I \xrightarrow{clone^T_{X,Y,Z}} [[Y \to TZ] \to [[X \to TY] \to [X \to TZ]]]$ that satisfy

$$\begin{array}{rcll} clone^T_{X,Y,Y}\,(unit^T_Y)\,h & = & h & \\ clone^T_{X,Y,Z}\,k\,(unit^T_Y \odot f) & = & k \odot f & \text{for each } I \xrightarrow{f} [X \to Y] \\ clone^T_{X,Y,W}\,(clone^T_{Y,Z,W}\,n\,k)\,h & = & clone^T_{X,Z,W}\,n\,(clone^T_{X,Y,Z}\,k\,h) & \text{for each } I \xrightarrow{n} [Z \to TW] \end{array}$$

ext form: a family of $\mathcal{X}$-arrows $I \xrightarrow{ext^T_{X,Y}} [[X \to YT] \to [XT \to YT]]$ that satisfy

$$\begin{array}{rcl} (ext^T_{X,Y}\,h) \odot unit^T_X & = & h \\ ext^T_{X,X}(unit^T_X) & = & 1_{TX} \\ ext^T_{X,Z}\left((ext^T_{Y,Z}\,k) \odot h\right) & = & (ext^T_{Y,Z}\,k) \odot (ext^T_{X,Y}\,h) \end{array}$$

bind form: a family of $\mathcal{X}$-arrows $I \xrightarrow{bind^T_{X,Y}} [XT \to [[X \to TY] \to TY]]$ that satisfy

$$\begin{array}{rcll} bind^T_{X,Y}(unit^T_X\,x)\,h & = & h@x & \text{for each } I \xrightarrow{x} X \\ bind^T_{X,X}\,\widehat{x}\,(unit^T_X) & = & \widehat{x} & \text{for each } I \xrightarrow{\widehat{x}} TX \\ bind^T_{Y,Z}\,(bind^T_{X,Y}\,\widehat{x}\,h)\,k & = & bind^T_{X,Z}\,\widehat{x}\,m & \text{where } m@y = bind^T_{Y,Z}(k@y)h \end{array}$$

whenever $I \xrightarrow{h} [X \to TY]$ and and $I \xrightarrow{k} [Y \to TZ]$.

Since these formulations are interdefinable, it is possible to derive a monadic λ-interpreter in any form a by partially evaluating a given monadic λ-interpreter expressed in the form b with respect to the definition of a in terms of b.

6 Conclusion and Issues

Monads appear to be an elegant and general structuring device in programming languages. They can encapsulate entire aspects of computation, in a way that not only hides the aspect from the rest of the computation but also ensures that effects of the encapsulated part on the visible parts are performed in the natural way. However, practical applications have been complicated by the need for tedious instantiations.

We have illustrated that partial evaluation is an ideal tool for automating this process. Using partial evaluation, we can automatically instantiate a λ-interpreter with a particular monad. We can reduce a program to its monadic components by specializing the λ-interpreter with respect to the program. And we can shift between the different interdefinable representations of monads by specializing a λ-interpreter using one representation with the definition of another representation. This eliminates most of the need for special tools when introducing monads into a programming language workbench.

Scheme, SML, and Haskell

All the work reported in this paper has been carried out with Scheme. We are currently transposing our monadic λ-interpreter to SML, to benefit from its static type system, and to Haskell, to get closer to Wadler's investigation — even though at present, we do not know of any partial evaluator powerful enough to repeat the present experiment in these programming languages.

Compiler construction

Monads in a λ-interpreter naturally ensure the single-threadedness of resources, which is crucial in efficient semantics-directed program manipulation [23, 2]. Wadler reports a critical use of monads in the construction of the Haskell compiler at Glasgow [28].

Binding time analysis

Binding time analysis is a general-purpose flow analysis which computes a safe approximation of binding times in a program [21, 4, 11]. This analysis can determine the compile-time and run-time aspects of a programming language specification, enabling semantics-based compiling and compiler generation [20, 6].

As illustrated in this paper, a monadic specification already commits the binding time division in a λ-interpreter, in that the core λ-interpreter defines compile-time computations. We are currently investigating this issue.

Modularization

In the area of abstract interpretation, it has proven very useful to modularize the semantics and to abstract only the modules, instead of the entire semantics. This technique is known as "factorization" and Nielson has proven that if only the standard domain constructors (direct sum, product, and function domain) and the corresponding λ-terms are used, then a relation will hold between the standard and the abstract semantics if it holds between the two modules [19].

The idea of parameterizing a λ-interpreter with a monad appears to generalize Nielson's framework since the monad also involves a type constructor (domain constructor) as part of the module. However the monad has a fixed structure, giving us less room to organize the modules at will.

References

[1] Anders Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1990. DIKU Report 90-17.

[2] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

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

[4] Charles Consel. Binding time analysis for higher order untyped functional languages. In LFP'90 [15], pages 264–272.

[5] Charles Consel. *The Schism Manual.* Yale University, New Haven, Connecticut, December 1990. Version 1.0.

[6] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[7] Pär Emanuelson and Anders Haraldsson. On compiling embedded languages in Lisp. In *Conference Record of the 1980 LISP Conference*, pages 208–215, Stanford, California, Aug 1980.

[8] Andrzej Filinski. Declarative continuations: An investigation of duality in programming language semantics. In D.H. Pitt et al., editors, *Category Theory and Computer Science*, number 389 in Lecture Notes in Computer Science, pages 224–249, Manchester, UK, September 1989.

[9] Yoshihito Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls 2, 5*, pages 45–50, 1971.

[10] Paul Hudak and Neil D. Jones, editors. *First ACM SIGPLAN and IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, New Haven, Connecticut, June 1991. ACM, ACM Press.

[11] Neil D. Jones. Tutorial on binding time analysis. In Hudak and Jones [10].

[12] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[13] G. M. Kelly. *Basic Concepts of Enriched Category Theory*. London Mathematical Society Lecture Note Series. Cambridge University Press, Cambridge, 1982.

[14] Peter Lee and Uwe Pleban. On the use of LISP in implementing denotational semantics. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 233–248, Cambridge, Massachusetts, August 1986.

[15] *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[16] Ernest G. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics*. Springer-Verlag, New York – Berlin, 1976.

[17] Eugenio Moggi. An abstract view of programming languages. Course notes, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, May 1989.

[18] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[19] Flemming Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69(2):117–242, 1989.

[20] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988.

[21] Hanne Riis Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 98–106, January 1988.

[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] David A. Schmidt. Detecting global variables in denotational definitions. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

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

[25] Guy L. Steele Jr. and Gerald J. Sussman. The art of the interpreter or, the modularity complex (parts zero, one, and two). AI Memo 453, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[26] Ross Street. The formal theory of monads. *J. Pure Appl. Algebra*, 2:149–168, 1972.

[27] Philip Wadler. Comprehending monads. In LFP'90 [15], pages 61–78.

[28] Philip Wadler. The essence of functional programming. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992. ACM Press.