Yale University
Department of Computer Science

A Modular Semantics for Compiler Generation [1]

Sheng Liang

YALEU/DCS/TR-1067
February 1995

[1]This work was supported by the Advanced Research Project Agency and the Office of Naval Research under Arpa Order 8888, Contract N00014-92-C-0153.

Abstract

We propose a modular specification of programming language semantics to facilitate semantics-directed compiler generation. *Modular monadic semantics* allows us to define a language with a rich set of features from reusable building blocks, each implementing a single feature. We achieve modularity by using *monads* and *monad transformers* to structure denotational semantics, and still retain the power of equational reasoning. The added benefits of reasoning in monadic style include useful laws for programming language constructs, better structured proofs and more general results. To demonstrate, we present an axiomatization of environments and use it to prove the correctness of a well-known compilation technique. The monadic approach also helps to generate efficient code for various target languages with different sets of built-in features. In addition to the standard semantics, a variety of program analysis tools, such as type checkers and abstract interpreters, can also be incorporated in our framework.

1 Introduction

We propose a modular semantics which allows language designers to:

- automatically generate compilers from semantic descriptions,
- add (or remove) programming language features without causing global changes to the existing specification,
- prove the correctness of compilation strategies,
- generate code for a variety of target languages with different sets of built-in features, and
- express various aspects of programming language semantics (in addition to core semantics, for example, type systems and optimization techniques).

Our goal is similar to that of Action Semantics [32] and related approaches (by, for example, Lee [26]). It has long been recognized that traditional denotational semantics [39] is not suitable for compiler generation for a number of crucial reasons [26], among which is the lack of modularity and extensibility.

We take advantage of a new development in programming language theory, called *monads* [29], to structure denotational semantics so that it achieves a high level of modularity and extensibility. Figure 1 shows how our modular monadic semantics is organized. Language designers specify semantic modules by using a set of basic operations. The expression "$e_1 := e_2$", for example, is interpreted by "Assign(v_1, v_2)", where v_1 and v_2 are the results of e_1 and e_2. Each function in the intermediate layer (such as "Assign") is in turn implemented using "kernel-level" primitive operations (such as "update").

While it is a well-known practice to base programming language semantics on a kernel-language, the novelty of our approach lies in how the kernel-level primitive operations are organized. In our framework, depending on how much support the upper layers need, any set of primitive operations can be put together in a modular way using an abstraction mechanism called *monad transformers* [29] [27]. Monad transformers provide the power needed to represent the abstract notion of programming language features, but still allow us to access low-level semantic details. In fact, since monad transformers are defined as higher-order functions, our monadic semantics is no more than a structured version of denotational semantics, and all conventional reasoning methods (such as β substitution) apply.

The source language we consider in this paper has a variety of features, including both call-by-name and call-by-value versions of functions and variable bindings:

$$
\begin{array}{rcll}
e & ::= & n \mid e_1 + e_2 & \text{(arithmetic operations)} \\
& \mid & v \mid \lambda_n v.e \mid \lambda_v v.e \mid e_1\ e_2 & \text{(cbn and cbv functions)} \\
& \mid & \mathrm{let}_n\ v = e_1\ \mathrm{in}\ e_2 \mid \mathrm{let}_v\ v = e_1\ \mathrm{in}\ e_2 & \text{(let bindings)} \\
& \mid & \mathrm{callcc} & \text{(first-class continuations)} \\
& \mid & e_1 := e_2 \mid \mathrm{ref}\ e \mid \mathrm{deref}\ e & \text{(imperative features)} \\
& \mid & e_1 \rightarrow e_2, e_3 & \text{(conditionals)}
\end{array}
$$

Our monadic semantics is developed with automatic compiler generation in mind. We will investigate how an interpreter based on the modular monadic semantics can be turned into a compiler. Major overheads of interpreters include run-time dispatch on the abstract syntax, explicit variable lookups, dynamic type checking, and the inability to perform most optimizations. We address each of these problems in this paper. In Section 2, we will define a compositional high-level semantics for our source language which guarantees that we can unfold all recursive calls to the evaluator, and thus avoid the overhead of dispatching on the abstract syntax tree. In section 3, we show how monad laws and axioms can be used to optimize intermediate code. To demonstrate

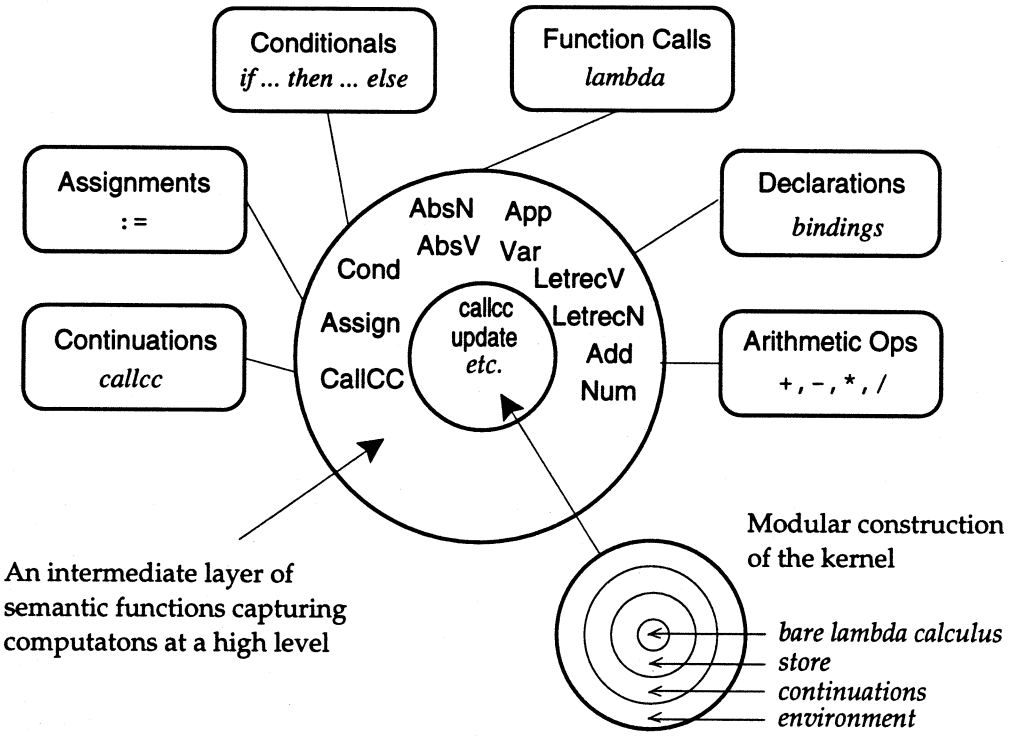

Figure 1: The organization of modular monadic semantics

the reasoning powers of monad transformers, in Section 4 we generalize Wand's [43] proof of the correctness of a well-known technique to overcome the overhead of dynamic variable lookups by transforming variables in the source language into variables in the meta-language. In section 5 we discuss how to utilize the modularity provided by monad transformers to generate efficient code for a variety of target languages. In section 6 we address how to specify non-standard semantics (for type checking and abstract interpretation) for the source language.

Although we have not actually generated a compiler from a monadic semantics, we believe that the issues addressed in this paper are important steps towards that goal.

The major contribution of this paper is to propose a modular and easy to reason about approach to semantics-directed compiler generation. Along the way we demonstrate that reasoning in monadic style enables us to better structure proofs and obtain more general results than in denotational semantics. Such insights are potentially useful in verifying large and evolving language implementations [14].

We present our results in the traditional denotational semantics style [39], augmented with a Haskell-like [19] type declaration syntax to express monads as type constructors. No prior knowledge of monads is assumed.

2 A Modular Monadic Semantics

This section uses some of the results from our earlier work on modular interpreters [27] to define a modular semantics for our source language.

2.1 A High-level Monadic Semantics

Traditional denotational semantics maps, say, a term, an environment and a continuation to an answer. In contrast, monadic semantics maps terms to computations, where the details of the environment, store, etc. are "hidden". Specifically, our semantic evaluation function E has type:

$$E \;:\; \mathit{Term} \to \mathit{Compute\ Value}$$

where *Value* denotes the result of the computation. The type constructor *Compute* is called a *monad*. It abstracts away the details of a computation, exposing only the result type. We will define monads more formally later, but for now we note that *Compute* comes equipped with two basic operations:

$$\begin{array}{ll} \mathsf{then} & : \; \mathit{Compute}\ a \to (a \to \mathit{Compute}\ b) \to \mathit{Compute}\ b \\ \mathsf{return} & : \; a \to \mathit{Compute}\ a \end{array}$$

We usually write "then" in an infix form. Intuitively, "c_1 then $\lambda v.c_2$" is a computation that first computes c_1, binds the result to v, and then computes c_2. "Return v" is a trivial computation that simply returns v as result.

Monadic combinators are semantic functions returning a computation, such as:

$$\begin{array}{ll} \mathsf{Num} & : \; \mathit{Int} \to \mathit{Compute\ Value} \\ \mathsf{Add} & : \; (\mathit{Value}, \mathit{Value}) \to \mathit{Compute\ Value} \end{array}$$

where "Num(n)" just returns the number n (injected into the *Value* domain) as the result of a trivial computation, and "Add(v_1, v_2)" is a computation that adds v_1 and v_2, and returns the result. Using these monadic combinators, we can define a (very tiny) arithmetic semantics as follows:

$$\begin{array}{lcl} E[\![n]\!] & = & \mathsf{Num}(n) \\ E[\![e_1 + e_2]\!] & = & E[\![e_1]\!]\ \mathsf{then}\ \lambda v_1.E[\![e_2]\!]\ \mathsf{then}\ \lambda v_2.\mathsf{Add}(v_1, v_2) \end{array}$$

To evaluate "$e_1 + e_2$", we evaluate e_1 and e_2 in turn, and then pass the results to Add. In denotational semantics, the interpretations for arithmetic expressions are slightly different depending on whether we are passing an environment around, or whether we write in direct or continuation-passing styles. In contrast, our monadic semantics for arithmetic expressions stays the same no matter what details of computation (e.g., continuations, environments, states) are captured in the underlying monad.

The following are high-level specifications of a few other building blocks, those for functions, declarations, first-class continuations, assignments, and conditionals:

$$\begin{array}{ll} \mathsf{Var} & : \; \mathit{Name} \to \mathit{Compute\ Value} \\ \mathsf{AbsN} & : \; (\mathit{Name}, \mathit{Compute\ Value}) \to \mathit{Compute\ Value} \\ \mathsf{AbsV} & : \; (\mathit{Name}, \mathit{Compute\ Value}) \to \mathit{Compute\ Value} \\ \mathsf{App} & : \; (\mathit{Value}, \mathit{Compute\ Value}) \to \mathit{Compute\ Value} \end{array}$$

$$\begin{array}{lcll} E[\![v]\!] & = & \mathsf{Var}(v) & \\ E[\![\lambda_n v.e]\!] & = & \mathsf{AbsN}(v, E[\![e]\!]) & \text{(call-by-name)} \\ E[\![\lambda_v v.e]\!] & = & \mathsf{AbsV}(v, E[\![e]\!]) & \text{(call-by-value)} \\ E[\![e_1\ e_2]\!] & = & E[\![e_1]\!]\ \mathsf{then}\ \lambda f.\mathsf{App}(f, E[\![e_2]\!]) & \end{array}$$

$$\begin{array}{ll} \mathsf{LetN} & : \; (\mathit{Name}, \mathit{Compute\ Value}, \mathit{Compute\ Value}) \to \mathit{Compute\ Value} \\ \mathsf{LetV} & : \; (\mathit{Name}, \mathit{Compute\ Value}, \mathit{Compute\ Value}) \to \mathit{Compute\ Value} \end{array}$$

$$\begin{array}{lcll} E[\![\mathsf{let}_n\ x = e_1\ \mathsf{in}\ e_2]\!] & = & \mathsf{LetN}(x, E[\![e_1]\!], E[\![e_2]\!]) & \text{(name substitution)} \\ E[\![\mathsf{let}_v\ x = e_1\ \mathsf{in}\ e_2]\!] & = & \mathsf{LetV}(x, E[\![e_1]\!], E[\![e_2]\!]) & \text{(value substitution)} \end{array}$$

$$\mathsf{CallCC} \;:\; \mathit{Compute\ Value}$$

$$E[\![\text{callcc}]\!] = \mathsf{CallCC}$$

$$\begin{array}{lcl} \mathsf{Assign} & : & (\mathit{Value}, \mathit{Value}) \to \mathit{Compute\ Value} \\ \mathsf{Ref} & : & \mathit{Value} \to \mathit{Compute\ Value} \\ \mathsf{Deref} & : & \mathit{Value} \to \mathit{Compute\ Value} \end{array}$$

$$\begin{array}{lcl} E[\![e_1 := e_2]\!] & = & E[\![e_1]\!]\ \mathsf{then}\ \lambda l.E[\![e_2]\!]\ \mathsf{then}\ \lambda r.\mathsf{Assign}(l, r) \\ E[\![\text{ref } e]\!] & = & E[\![e]\!]\ \mathsf{then}\ \lambda v.\mathsf{Ref}\ v \\ E[\![\text{deref } e]\!] & = & E[\![e]\!]\ \mathsf{then}\ \lambda a.\mathsf{Deref}\ a \end{array}$$

$$\mathsf{Cond} : (\mathit{Value}, \mathit{Compute\ Value}, \mathit{Compute\ Value}) \to \mathit{Compute\ Value}$$

$$E[\![e_1 \to e_2, e_3]\!] = E[\![e_1]\!]\ \mathsf{then}\ \lambda b.\mathsf{Cond}(b, E[\![e_2]\!], E[\![e_3]\!])$$

Our high-level monadic semantics bears a strong resemblance to action semantics, except that it uses only "then" and "return" to thread computations. Together with monadic combinators, these two operations are powerful enough to model various kinds of control flows (e.g., error handling, function calls and callcc) in sequential languages. Like in action semantics, we make a deliberate effort to give a very high-level view of the source language semantics.

We require that a semantics specified in terms of monadic combinators to be *compositional*: the arguments in recursive calls to E are substructures of the argument received on the left-hand side. From a theoretical point of view, it makes induction proofs on program structures possible. In practice, this guarantees that given any abstract syntax tree, we can recursively unfold all calls to the interpreter, effectively removing the run-time dispatch on abstract syntax tree.

Since most of the above semantics clauses are trivial, the intermediate layer of monadic combinators is not strictly necessary. We put them in mainly for the purpose of presentation — from now on we will focus on computations represented as functions, and stay away from syntax. In addition, realistic languages may have many syntactic constructs, but require relatively few monadic combinators. A few other benefits of using an intermediate layer of monad combinators will be mentioned during the course of the paper.

From now on we assume that the source program has been expanded into a core language composed out of "then", "return" and monadic combinators such as "Add" and "App". Semantics of the source language is specified by defining the monadic combinators accordingly.

2.2 The Standard Semantics

In the standard semantics, *Value* is the domain sum of basic values and functions. Functions map computations to computations:

$$\begin{array}{lcl} \textbf{type } \mathit{Fun} & = & \mathit{Compute\ Value} \to \mathit{Compute\ Value} \\ \textbf{type } \mathit{Value} & = & \mathit{Int} + \mathit{Bool} + \mathit{Addr} + \mathit{Fun} + \ldots \end{array}$$

The standard semantics for arithmetic expressions is as follows:

$$\begin{array}{lcl} \mathsf{Num}_s(n) & = & \mathsf{return}(n\ \textbf{in}\ \mathit{Value}) \\ \mathsf{Add}_s(v_1, v_2) & = & \textbf{if}\ \mathsf{checkType}(v_1, v_2)\ \textbf{then}\ \mathsf{err}\ \text{"}\mathit{type\ error}\text{"} \\ & & \qquad \textbf{else}\ \mathsf{return}((v_1 \mid \mathit{Int}) + (v_2 \mid \mathit{Int})\ \textbf{in}\ \mathit{Value}) \end{array}$$

We use a *primitive monadic combinator* (a semantic function directly supported by the underlying monad *Compute*):

$$\mathsf{err} : \mathit{String} \to \mathit{Compute}\ a$$

to report type errors. For clarity, from now on we will omit domain injection/projection and type checking.

Function abstractions and applications need access to an environment *Env* which maps variable names to computations, and two more primitive monadic combinators which retrieve the current environment and perform a computation in a given environment, respectively:

$$\begin{array}{lcl} \textbf{type } \mathit{Env} & = & \mathit{Name} \to \mathit{Compute\ Value} \\ \mathsf{rdEnv} & : & \mathit{Compute\ Env} \\ \mathsf{inEnv} & : & \mathit{Env} \to \mathit{Compute\ Value} \to \mathit{Compute\ Value} \end{array}$$

The standard semantics for functions is as follows:

$$\begin{array}{lcl} \mathsf{Var}_s\, s & = & \mathsf{rdEnv\ then}\ \lambda\rho.\rho\ s \\ \mathsf{AbsN}_s(s,c) & = & \mathsf{rdEnv\ then}\ \lambda\rho.\ \mathsf{return}(\lambda c_1.\mathsf{inEnv}\ (\rho[c_1/s])\ c) \\ \mathsf{AbsV}_s(s,c) & = & \mathsf{rdEnv\ then}\ \lambda\rho.\ \mathsf{return}(\lambda c_1.c_1\ \mathsf{then}\ \lambda v.\mathsf{inEnv}\ (\rho[\mathsf{return}\ v/s])\ c) \\ \mathsf{App}_s(f,c) & = & \mathsf{rdEnv\ then}\ \lambda\rho.f(\mathsf{inEnv}\ \rho\ c) \end{array}$$

The difference between call-by-value and call-by-name is clear: the former reduces the argument before evaluating the function body, whereas the latter does not. In a function application, the argument is packaged up with the current environment to form a closure.

In this paper, we only consider non-recursive let bindings, which can be translated into applications of lambda abstractions:

$$\begin{array}{lcl} \mathsf{LetN}_s(s,c_1,c_2) & = & \mathsf{App}_s(\mathsf{AbsN}_s(s,c_2),c_1) \\ \mathsf{LetV}_s(s,c_1,c_2) & = & \mathsf{App}_s(\mathsf{AbsV}_s(s,c_2),c_1) \end{array}$$

Here we see another benefit of using an intermediate layer of monadic combinators such as App: Monadic combinators can be defined in terms of each other, while the high-level semantics remains compositional.

To simplify the presentation somewhat, we assume that Assign_s, Deref_s, and Ref_s can be defined using the primitive monad combinator:

$$\mathsf{update} \ : \ (\mathit{Store} \to \mathit{Store}) \to \mathit{Compute\ Store}$$

for some suitably chosen *Store*. We can read the store by passing update the identity function, and change the store by passing it a state transformer. Although update returns the entire state, properly defined Assign_s, Deref_s and Ref_s can guarantee that the store is never duplicated (see, for example, [36]).

CallCC simply returns a function expecting another function as an argument, to which the current continuation will be passed.

$$\mathsf{callcc} \ : \ ((\mathit{Value} \to \mathit{Compute\ Value}) \to \mathit{Compute\ Value}) \to \mathit{Compute\ Value}$$

$$\mathsf{CallCC}_s \ = \ \mathsf{return}(\lambda f.f\ \mathsf{then}\ \lambda f'.\mathsf{callcc}(\lambda k.f'(\lambda a.a\ \mathsf{then}\ k)))$$

Conditionals are supported by the meta-language:

$$\mathsf{Cond}_s(v,c_1,c_2) \ = \ \textbf{if } v \textbf{ then } c_1 \textbf{ else } c_2$$

2.3 Constructing the *Compute* Monad

It is clear that monad *Compute* needs to support the following primitive monad combinators:

$$\begin{array}{lcl} \mathsf{err} & : & \mathit{String} \to \mathit{Compute}\ a \\ \mathsf{rdEnv} & : & \mathit{Compute\ Env} \\ \mathsf{inEnv} & : & \mathit{Env} \to \mathit{Compute\ Value} \to \mathit{Compute\ Value} \\ \mathsf{update} & : & (\mathit{Store} \to \mathit{Store}) \to \mathit{Compute\ Store} \\ \mathsf{callcc} & : & ((\mathit{Value} \to \mathit{Compute\ Value}) \to \mathit{Compute\ Value}) \to \mathit{Compute\ Value} \end{array}$$

If we follow the traditional denotational semantics approach, now is the time to set up domains and implement the above functions. The major drawback of such monolithic approach is that if we add some source language feature later on, all the functions may have to be redefined.

For the sake of modularity, we start from a simple monad and add more and more features. The simplest monad of all is the identity monad. All it captures is function application:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Id}\ a & = & a \\ \\ \mathsf{return}_{\mathit{Id}}\ x & = & x \\ c\ \mathsf{then}_{\mathit{Id}}\ f & = & f\ c \end{array}$$

A *monad transformer* takes a monad, and returns a new monad with added features. For example, "*StateT s*" adds a state s to any monad m:

$$\begin{array}{lcl} \textbf{type}\ \mathit{StateT}\ s\ m\ a & = & s \rightarrow m\ (s, a) \\ \\ \mathsf{return}_{(\mathit{StateT}\ s\ m)}\ x & = & \lambda s.\ \mathsf{return}_m(s, x) \\ c\ \mathsf{then}_{(\mathit{StateT}\ s\ m)}\ k & = & \lambda s_0.c\ s_0\ \mathsf{then}_m\ \lambda(s_1, a).k\ a\ s_1 \end{array}$$

To see how monad transformers work, let us apply *StateT* to the identity monad *Id*:

$$\begin{array}{lcl} \textbf{type}\ \mathit{StateT}\ s\ \mathit{Id}\ a & = & s \rightarrow \mathit{Id}\ (s, a) \\ & = & s \rightarrow (s, a) \\ \\ \mathsf{return}_{(\mathit{StateT}\ s\ \mathit{Id})}\ x & = & \lambda s.\ \mathsf{return}_{\mathit{Id}}(s, x) \\ & = & \lambda s.(s, x) \\ c\ \mathsf{then}_{(\mathit{StateT}\ s\ \mathit{Id})}\ k & = & \lambda s_0.c\ s_0\ \mathsf{then}_{\mathit{Id}}\ \lambda(s_1, a).k\ a\ s_1 \\ & = & \lambda s_0.\mathsf{let}\ (s_1, a) = m\ s_0\ \mathsf{in}\ k\ a\ s_1 \end{array}$$

Note that "*StateT s Id*" is the standard state monad found, for example, in Wadler's work [40].

To make the newly introduced state accessible, "*StateT s*" introduces update on s which applies f to the state, and returns the old state:

$$\begin{array}{lcl} \mathsf{update} & : & (s \rightarrow s) \rightarrow \mathit{StateT}\ s\ m\ a \\ \mathsf{update}\ f & = & \lambda s.\ \mathsf{return}(f\ s, s) \end{array}$$

Figure 2 gives the definitions of several other monad transformers, including those for errors (*ErrorT*), continuations (*ContT*) and environments (*EnvT*). Now we can construct *Compute* by applying a series of monad transformers to the base monad *Id*:

$$\textbf{type}\ \mathit{Compute}\ a = \mathit{EnvT}\ \mathit{Env}\ (\mathit{ContT}\ \mathit{Answer}\ (\mathit{StateT}\ \mathit{Store}\ (\mathit{ErrorT}\ \mathit{Id})))\ a$$

Env, *Store* and *Answer* are the type of the environment, store and answer, respectively.

One issue remains to be addressed. The update function introduced by *StateT* does not work on *Compute*, which contains features added later by other monad transformers. In general, this is the problem of *lifting* operations through monad transformers. Figure 3 gives a brief summary of useful liftings (See [27] for a detailed description.) For example, in the *Compute* monad above, "update f" is:

$$\lambda s.Ok\ (f\ s, s)$$

when first introduced by *StateT*. After *Compute* is finally constructed, "update f" becomes:

$$\lambda\rho.\lambda k.\lambda s.k\ s\ (f\ s)$$

In summary, monad transformers allow us to easily construct monads with a certain set of primitive monadic combinators, defined as higher-order functions.

Continuation:

$$\textbf{type } \mathit{ContT}\ ans\ m\ a = (a \to m\ ans) \to m\ ans$$

$$\mathsf{return}_{(\mathit{ContT}\ ans\ m)}\ a = \lambda k.ka$$

$$c\ \mathsf{then}_{(\mathit{ContT}\ ans\ m)}\ f = \lambda k.c(\lambda a.fak)$$

$$\mathsf{callcc}\ f = \lambda k.f(\lambda a.\lambda k'.ka)k$$

Environment:

$$\textbf{type } \mathit{EnvT}\ e\ m\ a = e \to m\ a$$

$$\mathsf{return}_{(\mathit{EnvT}\ e\ m)}\ a = \lambda \rho.\ \mathsf{return}_m\ a$$

$$c\ \mathsf{then}_{(\mathit{EnvT}\ e\ m)}\ k = \lambda \rho.c\rho\ \mathsf{then}_m\ \lambda a.ka\rho$$

$$\mathsf{rdEnv} = \lambda \rho.\ \mathsf{return}_m\ \rho$$

$$\mathsf{inEnv}\ \rho\ c = \lambda \rho'.c\ \rho$$

Errors:

$$\textbf{type } \mathit{Error}\ a = \mathit{Ok}\ a \mid \mathit{Error}\ \mathit{String}$$

$$\textbf{type } \mathit{ErrorT}\ m\ a = m\ (\mathit{Error}\ a)$$

$$\mathsf{return}_{(\mathit{ErrorT}\ m)}\ a = \mathsf{return}_m \cdot \mathit{Ok}$$

$$c\ \mathsf{then}_{(\mathit{ErrorT}\ m)}\ k = c\ \mathsf{then}_m\ \lambda a.\ \textbf{case } a \textbf{ of}$$

$$(\mathit{Ok}\ x) \to kx$$

$$(\mathit{Error}\ msg) \to \mathsf{return}_m(\mathit{Error}\ msg)$$

$$\mathsf{err} = \mathsf{return}_m \cdot \mathit{Error}$$

Figure 2: Monad transformers

Every monad transformer t has a function:

$$\mathsf{lift}_{(t\ m)} : m\ a \to t\ m\ a$$

which embeds a computation in monad m into "$t\ m$". Functions err, update and rdEnv are easily lifted using lift:

$$\mathsf{err}_{(t\ m)} = \mathsf{lift}_{(t\ m)} \cdot \mathsf{err}_m$$

$$\mathsf{update}_{(t\ m)} = \mathsf{lift}_{(t\ m)} \cdot \mathsf{update}_m$$

$$\mathsf{rdEnv}_{(t\ m)} = \mathsf{lift}_{(t\ m)}\mathsf{rdEnv}_m$$

Some liftings of callcc, inEnv and the definitions of lift for each monad transformer are listed in the following table:

$t\ m$	$\mathsf{callcc}_{(t\ m)}\ f$	$\mathsf{inEnv}_{(t\ m)}\ \rho\ c$	$\mathsf{lift}_{(t\ m)}\ c$
$\mathit{EnvT}\ e\ m$	$\lambda \rho.\mathsf{callcc}_m(\lambda k.f(\lambda a.\lambda \rho'.ka)\rho)$	$\lambda \rho'.\mathsf{inEnv}_m\ \rho\ (c\rho')$	$\lambda \rho.c$
$\mathit{ContT}\ ans\ m$			$\lambda k.c\ \mathsf{then}_m\ k$
$\mathit{StateT}\ s\ m$	$\lambda s_0.\mathsf{callcc}_m(\lambda k.$ $f(\lambda a.\lambda s_1.k(s_1,a))s_0)$	$\lambda s.\mathsf{inEnv}_m\ \rho\ (cs)$	$\lambda s.c\ \mathsf{then}_m\ \lambda x.$ $\mathsf{return}_m(s,x)$
$\mathit{ErrorT}\ m$	$\mathsf{callcc}_m\ (\lambda k.f(\lambda a.k(\mathit{Ok}\ a)))$	$\mathsf{inEnv}_m\ \rho\ c$	$\mathsf{map}_m\ \mathit{Ok}$

Figure 3: Liftings

3 Using Monad Laws to Transform Programs

Following the monadic semantics presented in the previous section, by unfolding all calls to semantic function E,we can transform source-level programs into monadic-style code. For example, $((\lambda x.x + 1)\ 2)_v$ is transformed to:

```
rdEnv then λρ.
return(λc.inEnv (ρ[c/"x"])
                (rdEnv then λρ.
                 ρ"x" then λv1.
                 return 1 then λv2.
                 return(v1 + v2))) then λf.
return 2 then λv.
f(return v)
```

In this section we formally introduce monads and their laws, and show how to use the laws to simplify the above program.

3.1 Monads and Monad Laws

Definition 3.1 A **monad** M is a type constructor, together with two operations:

$$\begin{array}{lcl} \mathsf{then} & : & M\ a \to (a \to M\ b) \to M\ b \\ \mathsf{return} & : & a \to M\ a \end{array}$$

satisfying the following laws [40]:

$$\begin{array}{rcll} (\mathsf{return}\ a)\ \mathsf{then}\ k & = & k\ a & \text{(left unit)} \\ c\ \mathsf{then}\ \mathsf{return} & = & c & \text{(right unit)} \\ c_1\ \mathsf{then}\ \lambda v_1.(c_2\ \mathsf{then}\ \lambda v_2.c_3) & = & (c_1\ \mathsf{then}\ \lambda v_1.c_2)\ \mathsf{then}\ \lambda v_2.c_3 & \text{(associativity)} \end{array}$$

Intuitively, the (left and right) unit laws say that trivial computations can be skipped in certain contexts; and the associativity law captures the very basic property of sequencing, one that we usually take for granted in programming languages.

We can verify, by equational reasoning, for example, that return_{Id} and then_{Id} satisfy the above laws, and *EnvT*, *ContT* etc. indeed transform monads to monads.

Monad laws are important for reasoning about monadic style programs. For example, our compiler translates the expression "2 + 3" to:

return 2 then λv_1. return 3 then λv_2. return$(v_1 + v_2)$

We can apply the left unit law twice, and reduce the above to:

return$(2 + 3)$

which can of course be further optimized to "return 5".

Every application of a monad law usually corresponds to a number of β reductions. Monad laws allow us to perform β reductions at the right places, and avoid those corresponding to actual computations in the source program, which in turn may lead to non-termination.

Without knowledge about the environment-handling operations inEnv and rdEnv, however, monad laws alone can only simplify the example in the beginning of the section to:

```
rdEnv then λρ.
(λc.inEnv (ρ[c/"x"])
          (rdEnv then λρ.
           ρ"x" then λv.
           return(v + 1)))
(return 2))
```

To further simplify the above program, we need to look at the laws environment-related operations should satisfy.

3.2 Environment Axioms

We axiomatize the environment-manipulating functions as follows:

Definition 3.2 Monad M is an **environment monad** if it has two operations: rdEnv and inEnv, which satisfy the following axioms:

$$
\begin{array}{rcll}
(\mathsf{inEnv}\ \rho) \cdot \mathsf{return} & = & \mathsf{return} & \text{(unit)} \\
\mathsf{inEnv}\ \rho\ (c_1\ \mathsf{then}\ \lambda v.c_2) & = & \mathsf{inEnv}\ \rho\ c_1\ \mathsf{then}\ \lambda v.\mathsf{inEnv}\ \rho\ c_2 & \text{(distribution)} \\
\mathsf{inEnv}\ \rho\ \mathsf{rdEnv} & = & \mathsf{return}\ \rho & \text{(cancellation)} \\
\mathsf{inEnv}\ \rho\ (\mathsf{inEnv}\ \rho'\ e) & = & \mathsf{inEnv}\ \rho'\ e & \text{(overriding)}
\end{array}
$$

Intuitively, a trivial computation cannot depend on the environment (the unit law); the environment stays the same across a sequence of computations (the distribution law); the environment does not change between a set and a read if there are no intervening computations (the cancellation law); and an inner environment supercedes an outer one (the overriding law).

As with the monad laws, the environment axioms can be verified by equational reasoning.

Proposition 3.1 *The monads supporting* rdEnv *and* inEnv *constructed using monad transformers* ErrorT, EnvT, StateT *and* ContT *are environment monads.*

Equipped with the environment axioms, we can further transform the example monadic code to:

```
rdEnv then λρ.
(λc.c then λv. return(v + 1))(return 2)
```

Note that explicit environment accesses have disappeared. Instead, the meta-language environment is directly used to support function calls. This is exactly what many partial evaluators achieve when they transform interpreters to compilers.

4 Using Monad Laws to Reason about Computations

We successfully transformed away the explicit environment in the above example, but can we do the same for arbitrary source programs? It turns out that we can indeed prove such a general result by using monad laws and environment axioms. We follow Wand [43], define a "natural semantics" which translates source language variables to lexical variables in the meta-language, and prove that it is equivalent to the standard semantics.

4.1 A Natural Semantics

We adopt Wand's definition of a *natural semantics* (different from Kahn's [5] notion) to our functional sub-language. For any source language variable name s, we assume there is a corresponding variable name v_s in the meta-language.

Definition 4.1 A **natural semantics** uses the environment of the meta-language for variables in the source language, and is given as follows:

$$\begin{array}{lcl} \mathsf{Var}_n(s) & = & v_s \\ \mathsf{AbsN}_n(s,c) & = & \mathsf{return}(\lambda v_s.c) \\ \mathsf{AbsV}_n(s,c) & = & \mathsf{return}(\lambda v'_s.v'_s \text{ then } \lambda v''_s.\, \mathsf{return}(\mathsf{return}\, v''_s) \text{ then } \lambda v_s.c) \\ \mathsf{App}_n(f,c) & = & fc \end{array}$$

Other monadic combinators, such as CallCC, Assign etc., do not explicitly deal with the environment, and have the same natural semantics as standard semantics.

4.2 Correspondence between Natural and Standard Semantics

The next theorem, a variation of Wand's [43], guarantees that it is safe to implement function calls in the source language using the meta-language environment.

Theorem 4.1 *Let c be a program composed out of* then, return, *and monad combinators. Let c_s be its standard semantics in an environment monad, c_n be its natural semantics in the same monad,*[1] *and ρ be the mapping from the source language variable names s to v_s, we have:*

$$\mathsf{inEnv}\ \rho\ c_s = c_n$$

To emphasize the modularity provided in our framework, we first prove the theorem for the functional sub-language, and then extend the result to the complete language.

4.2.1 Proof for the Functional Sub-language

We can establish the theorem for the functional sub-language by induction on the structure of *computations* composed out of return, then and the monad combinators for variables, lambda abstractions, and function applications. The full proof is given in the Appendix. The basic technique is the same as Wand's, except that in addition to the basic rules of lambda calculus (e.g., β reduction), we also use monad laws and environment axioms.

The proof is possible because both the source language and meta language are lexically scoped. If the source language contained dynamically scoped functions:

$$\mathsf{AbsD}_s(s,c) \quad = \quad \mathsf{return}(\lambda c_1.\mathsf{rdEnv} \text{ then } \lambda\rho.\mathsf{inEnv}\ (\rho[c_1/s])\ c)$$

where the caller-site environment is used within the function body, the theorem would fail to hold.

[1] This means that in natural semantics, we are still implicitly passing around an environment, even though it is never used. Thus the theorem as stated does not strictly correspond to Wand's result [43]. Fortunately, the *naturality of liftings* (see our earlier work [27] for details) guarantees that adding and removing a feature does not affect computations which do not use that particular feature. Therefore the theorem still holds if we remove the explicit environment support from the underlying monad in natural semantics. (The next section addresses this in more detail.)

4.2.2 Extension to the Complete Language

Consider another monad combinator CallCC. Since in proving the theorem we only used the axioms of environment monads, none of the cases already analyzed need to be proved again. We only have to verify that:

- the monad supporting continuations is still an environment monad, and
- the induction hypothesis holds for CallCC.

The former is stated in Proposition 3.1, and can be proved once and for all as we come up with monad transformers. The latter can be easily proved: CallCC does not explicitly deal with the environment, and has exactly the same natural semantics as the standard semantics. In addition, it is a trivial computation (see the definition in the last section). Thus the induction hypothesis holds following the unit axiom of environment monads.

Similarly we can extend the theorem to cover other features such as Assign.

In denotational semantics, adding a feature may change the structure of the entire semantics, forcing us to redo the induction for every case of abstract syntax. Indeed, Wand [43] pointed out that he could change the semantics into continuation-based, and prove the theorem, but only by modifying the proofs accordingly.

In denotational semantics, a computation is captured as a piece of syntax tree coupled with an environment, a store etc. On the other hand, we view computations as abstract entities with a set of equations. Therefore, like *Semantic Algebras* [31] in action semantics, monads provide an *axiomatic view* to denotational semantics.

5 Targeting Monadic Code

In general, it is more efficient to use target language built-in features instead of monadic combinators defined as higher-order functions. We have seen how the explicit environment can be "absorbed" into the meta-language. This section addresses the question of whether we can do the same for other features, such as the store and continuation.

We can view a target language as having a built-in monad supporting a set of primitive monadic combinators. For example, the following table lists the correspondence between certain ML constructs and primitive monadic combinators:

primitive monadic operators	ML construct
return x	x
c_1 then $\lambda x.c_2$	**let val** $x = c_1$ **in** c_2 **end**
update*	**ref**, **!**, **:=**
callcc	callcc
err	**raise Err**
...	

* ML reference cells support single-threaded states only.

It is easy to verify that the monad laws are satisfied in the above context. For example, the ML **let** construct is associative (assuming no unwanted name capturings occur):

```
let val v2 = let val v1 = c1          let val v1 = c1
             in c2                    in let val v2 = c2
             end                         in c3
in c3                                    end
end                                  end
```

Recall (in Section 2) that the *Compute* monad is constructed as:

type *Compute a* = *EnvT Env* (*ContT Answer* (*StateT Store* (*ErrorT Id*))) *a*

Now we substitute the base monad *Id* with the built-in ML monad (call it M_{ML}):

type *Compute'* *a* = *EnvT Env* (*ContT Answer* (*StateT Store* (*ErrorT* M_{ML}))) *a*

Note that *Compute'* supports two sets of continuation, state and error handling functions. The monadic code can choose to use the ML built-in ones instead of those implemented as higher-order functions. In addition, all liftings we construct satisfy an important property (called the Naturality of Liftings [29] [27]): adding or deleting a monad transformer does not change the result of programs which do not use its operations. Since none of the monad transformers in *Compute'* is used anymore, it suffices to run the target program on *Compute''*:

type *Compute''* *a* = M_{ML} *a*

which directly utilizes the more efficient ML built-in features.

The above transformation is possible because ML has a strictly richer set of features than our source language. If the source language requires a non-updatable version of state (for example, for the purpose of debugging), the corresponding state monad transformer will remain, and ensure the state is threaded correctly through all computations. If we instead target our source language to C, both the environment and continuation transformers have to be kept.

Therefore by using a monad with a set of primitive monadic combinators, we can expose the features embedded in the target language. It then becomes clear what is directly supported in the target language, and what needs to be compiled explicitly.

The above process seems trivial, but would have been impossible had we been working with traditional denotational semantics. Various features clutter up and make it hard to determine whether it is safe to remove certain interpretation overhead, and how to achieve that.

Earlier work [25] [11] [27] has shown that the order of monad transformers (in particular, some cases involving *ContT*) has an impact on the resulting semantics. In practice, we need to make sure when we discard the monad transformers, that the resulting change of ordering does not have unwanted effects on semantics.

6 Non-standard Monadic Semantics

In this section we show that by using a different set of monads, and redefining the monadic combinators, we can derive non-standard semantics corresponding to a wide range of language processing tools, in particular, a Hindley-Milner type checker [8], and a higher-order operation count analyzer based on abstract interpretation [7].

Specifying language processing tools as non-standard semantics of monadic combinators has two advantages:

1. We can use equational reasoning to verify the correctness of such tools, and investigate their relationships with the standard semantics.

2. We can share the high-level evaluator between the standard and non-standard semantics.

6.1 A Type Checker

The algorithm is based on a monadic type checker written by Hammond [16]. The difference is that our monad includes the type environment, and is constructed using monad transformers.

The type-checking monad and the value type are:

$$
\begin{array}{lcl}
\textbf{type } \mathit{TyVar} & = & \mathit{Int} \\
\textbf{type } \mathit{Compute}\ a & = & \mathit{EnvT}\ \mathit{TEnv}\ (\mathit{StateT}\ \mathit{Subst}\ (\mathit{StateT}\ \mathit{TyVar}\ (\mathit{ErrorT}\ \mathit{Id})))\ a \\
\textbf{type } \mathit{Value} & = & \mathit{TyVar} + (\mathit{Value} \to \mathit{Value})
\end{array}
$$

where *TEnv*, *Subst*, and *TyVar* denote the type environment, substitutions, and the counter for generating fresh type variables, respectively. Values here denote type signatures, which, for simplicity, are either a type variable (represented using an integer) or a function type.

We also need the following helper functions:

$$
\begin{array}{lcl}
\textbf{type } \mathit{Maybe}\ a & = & \mathit{Just}\ a + \mathit{Nothing} \\
\mathsf{lookup} & : & \mathit{String} \to \mathit{TEnv} \to \mathit{Maybe}\ (\mathit{Value}, \mathsf{N} + \mathsf{G}) \\
\mathsf{newVar} & : & \mathit{Compute}\ \mathit{Value} \\
\mathsf{subst} & : & \mathit{Value} \to \mathit{Compute}\ \mathit{Value} \\
\mathsf{instType} & : & \mathit{Value} \to \mathit{Compute}\ \mathit{Value} \\
\mathsf{unify} & : & \mathit{Value} \to \mathit{Value} \to \mathit{Compute}\ \mathit{Value}
\end{array}
$$

The environment maps variables to a pair consisting of a type and a flag indicating whether the variable is let-bound or lambda-bound. A type-checking semantics for monadic combinators can be given as (only the interesting cases are listed):

$$
\begin{array}{lcl}
\mathsf{Var}_t(s) & = & \mathsf{rdEnv}\ \mathsf{then}\ \lambda\rho. \\
 & & \textbf{case}\ \mathsf{lookup}\ s\ \rho\ \textbf{of} \\
 & & \quad \mathit{Just}\ (t, \mathsf{G}) \to \mathsf{instType}\ t\ \mathsf{then}\ \lambda t'.\mathsf{subst}\ t' \\
 & & \quad \mathit{Just}\ (t, \mathsf{N}) \to \mathsf{subst}\ t \\
 & & \quad \mathit{Nothing} \to \mathsf{err}(\text{"}\mathit{undefined\ var:}\ \text{"} \mathbin{++} s) \\
\mathsf{AbsN}_t(s, c) & = & \mathsf{rdEnv}\ \mathsf{then}\ \lambda\rho. \\
 & & \mathsf{newVar}\ \mathsf{then}\ \lambda t. \\
 & & \mathsf{inEnv}\ (\rho[(t, \mathsf{N})/s]\ c\ \mathsf{then}\ \lambda t_c. \\
 & & \mathsf{subst}\ t\ \mathsf{then}\ \lambda t', \\
 & & \mathsf{return}(t' \to t_c) \\
\mathsf{App}_t(t_1, c) & = & c\ \mathsf{then}\ \lambda t_2. \\
 & & \mathsf{newVar}\ \mathsf{then}\ \lambda t. \\
 & & \mathsf{unify}\ t_1\ (t_2 \to t)\ \mathsf{then}\ \lambda x. \\
 & & \mathsf{subst}\ t \\
\mathsf{LetN}_t(s, c_1, c_2) & = & c_1\ \mathsf{then}\ \lambda t_1. \\
 & & \mathsf{subst}\ t_1\ \mathsf{then}\ \lambda t_1'. \\
 & & \mathsf{inEnv}\ (\rho[(t_1', \mathsf{G})/s])c_2
\end{array}
$$

Note that lambda-bound type variables are marked with "N", whereas let-bound ones are marked with "G", meaning that they can be instantiated later. Unify calls err if it fails.

6.2 An Abstract Interpreter

Young [46] has built a parameterized abstract interpreter which does a variety of analyses by redefining primitive functions. It is much harder, however, to reuse part of the *standard* interpreter for abstract interpretation, due to the dramatic differences in evaluation strategies. Hall [15] has parameterized a standard evaluator for abstract interpretation, but she dealt with a very simple language, which, for example, did not need an environment.

In our framework, various abstract interpretation algorithms can be specified by giving a non-standard semantics to monadic combinators. As an example, we derive a simple higher-order operation-count analysis — which is useful for a variety of compiler optimizations — by specifying the monad and value types as follows:

$$\begin{array}{lcl}
\textbf{type } \mathit{Compute}\ a & = & \mathit{EnvT}\ \mathit{OpEnv}\ \mathit{Id}\ a \\
\textbf{type } \mathit{Value} & = & (\mathit{Int}, \mathit{Value} \to \mathit{Value})
\end{array}$$

Following Hudak and Young [20], we use a pair to capture the first-order and higher-order properties of a given expression. For the simple analysis we describe below, which does not deal with recursion, it is enough to use a simple environment monad mapping variables to their operation counts. If recursions exist, however, it will be necessary to introduce another environment to cache pending function calls [45], in order to guarantee termination.

$$\begin{array}{lcl}
\mathsf{Add}_c(v_1, v_2) & = & \mathsf{return}(1, \bot) \\
\mathsf{Var}_c(s) & = & \mathsf{rdEnv\ then}\ \lambda\rho.\rho\ s \\
\mathsf{AbsN}_c(s, c) & = & \mathsf{rdEnv\ then}\ \lambda\rho.\ \mathsf{return}(1, \lambda c_1.\mathsf{inEnv}\ (\rho[c_1/s])\ c) \\
\mathsf{AbsV}_c(s, c) & = & \mathsf{rdEnv\ then}\ \lambda\rho. \\
 & & \mathsf{return}(1,\ \lambda c_1.c_1\ \mathsf{then}\ \lambda v. \\
 & & \qquad \mathsf{inEnv}\ (\rho[\mathsf{return}(0, \mathsf{snd}\, v)/s])\ c\ \mathsf{then}\ \lambda a. \\
 & & \qquad \mathsf{return}(\mathsf{fst}\, v + \mathsf{fst}\, a, \mathsf{snd}\, a)) \\
\mathsf{App}_c(f, c) & = & \mathsf{rdEnv\ then}\ \lambda\rho.(\mathsf{snd}\, f)(\mathsf{inEnv}\ \rho\ c)\ \mathsf{then}\ \lambda a. \\
 & & \mathsf{return}(\mathsf{fst}\, f + \mathsf{fst}\, a + 1, \mathsf{snd}\, a) \\
\mathsf{Cond}_c(v, c_1, c_2) & = & c_1\ \mathsf{then}\ \lambda i_1.c_2\ \mathsf{then}\ \lambda i_2. \\
 & & \mathsf{return}(\mathsf{fst}\, v + \mathsf{imax}(\mathsf{fst}\, i_1, \mathsf{fst}\, i_2) + 1, \mathsf{fmax}(\mathsf{snd}\, i_1, \mathsf{snd}\, i_2))
\end{array}$$

Operation counts for all basic operations (such as addition and the creation of a closure) are arbitrarily set to 1. Note that the operations in an argument are counted every time it is evaluated in the body of a call-by-name function, but only once upon entering a call-by-value function. Imax simply returns the larger of two integers, whereas fmax compares two integer-returning functions, and has to wait for more arguments before it decides.

7 Related work

Due to the problems with traditional denotational semantics, earlier efforts [30] [35] [42] in semantics-directed compiler generation did not produce efficient compilers.

Mosses's *Action Semantics* [32] allows modular specification of programming language semantics, from which efficient compilers can be generated. In action semantics, source language constructs are translated into *actions*, built from a set of primitives and combinators. While action semantics is easy to construct, extend, understand and implement, we note the following comments made by Mosses ([32], page 5):

> Although the foundations of action semantics are firm enough, the *theory* for reasoning about actions (and hence about programs) is still rather weak, and needs further development. This situation is in marked contrast to that of denotational semantics, where the theory is strong, but severe pragmatic difficulties hinder its application to realistic programming languages.

Action semantics provided much of the inspiration for our work, which essentially attempts to formulate actions in a denotational semantics framework. Monad transformers roughly correspond to *facets* in action semantics, although issues such as concurrency are beyond the power of our denotational semantics-based approach.

Action semantics [4] [34] [33] and a related approach by Lee [26] have been successfully used to generate efficient compilers. Lee has given a denotational semantics to actions, and Even and Schmidt have presented a categorical model [12] and a type inferencer [13] for action semantics.

Doh and Schmidt [10] have presented a way to assign each action in action semantics specific "analysis functions," such as a type checker and a binding-time analyzer. Such analysis tools can

thus be generated automatically together with the compiler. The differences between our approach and Doh and Schmidt's are:

1. They redefine the operators for threading actions. We do not have to, since "return" and "then" come with any monad.

2. They showed how the type-checking rules for the source language can be automatically derived from the type-checking rules given on actions. Instead, we directly specify the type-checking algorithm. Their approach is clever, and handles subtyping, but does not deal with let-bound polymorphism.

Moggi [29] first used monads and monad transformers to structure denotational semantics. Wadler [41] [40] popularized Moggi's ideas in the functional programming community by using monads to structure functional programs, in particular, interpreters. Our work on monad-based modular interpreters [27] follows a series of earlier attempts by Steele [38], Jones and Duponcheel [21], and Espinosa [11].

Moggi [29] also raised (although did not address in detail) the issue of reasoning in a monadic framework. Wadler [41] listed the laws a state monad should satisfy. Hudak [18] suggested a more general framework — mutable abstract data types (MADTs) — to reason about states.

One application of partial evaluation [22] is to automatically generate compilers from interpreters [44] [2] [6] [28]. A partial evaluator has been successfully applied to an action interpreter [3], and similar results can be achieved with monadic interpreters [9].

Staging transformations, first proposed by Jørring and Scherlis [23], are a class of general program transformation techniques for separating a given computation into stages. Monadic transformers make computational stages somewhat more explicit by separating compile-time features, such as the environment, from run-time features.

Several researchers, including Kelsey and Hudak [24], Appel and Jim [1], and others, have built efficient compilers for higher-order languages by transforming the source language into continuation-passing style (CPS). The suitability of a monadic form as an intermediate form has been observed by many researchers (including, for example, Sabry and Felleisen [37] and Hatcliff and Danvy [17]). We will continue to explore along this direction in order to generate machine-level code from a monadic intermediate form.

8 Conclusions

We have shown that the monadic framework provides good support for high-level extensible specifications, program transformations, and reasoning about computations. Monadic-style proofs are better structured and easier to extend. The modular monadic semantics allows us to have an axiomatized formulation of well-known programming languages features such as environments. Our approach also facilitates generating code for various target languages and constructing program analysis tools.

Overall, we believe that modular monadic semantics is particularly suitable for compiler generation.

Acknowledgement

I would like to thank Paul Hudak for support and advice, Rajiv Mirani, Zhong Shao, Satish Pai, Dan Rabin and the PEPM'95 anonymous referees for helpful comments, and Ross Paterson, Tim Sheard and John Launchbury for ideas and discussions.