Modular Monadic Semantics

Sheng Liang   Paul Hudak

Department of Computer Science
Yale University
P.O. Box 208285
New Haven, CT 06520-8285, USA
{liang,hudak}@cs.yale.edu

Abstract

Modular monadic semantics is a high-level and modular form of denotational semantics. It is capable of capturing individual programming language features as small building blocks which can be combined to form a programming language of arbitrary complexity. Interactions between features are isolated in such a way that the building blocks are invariant. This paper explores the theory and application of modular monadic semantics, including the building blocks for individual programming language features, equational reasoning with laws and axioms, modular proofs, program transformation, modular interpreters, and semantics-directed compilation. We demonstrate that modular monadic semantics makes programming languages easier to specify, reason about, and implement than the alternative of using conventional denotational semantics.

Our contributions include: (a) the design of a fully modular interpreter based on monad transformers, including important features missing from several earlier efforts, (b) a method to lift monad operations through monad transformers, including difficult cases not achieved in earlier work, (c) a study of the semantic implications of the order of monad transformer composition, (d) a formal theory of modular monadic semantics that justifies our choice of liftings based on a notion of naturality, and (e) an implementation of our interpreter in Gofer, whose constructor classes provide just the added power over Haskell type classes to allow precise and convenient expression of our ideas.

A note to reviewers: this paper is rather long. Short of resorting to “Part I / Part II”, the one way we see to shorten it would be to remove Section 4 and its Appendix B, which would amount to eliminating contribution (e) above. This would shorten the paper by about 12 pages.

1 Introduction

1.1 Overview

Denotational semantics (Stoy, 1977) is among the most important developments in programming language theory. It gives a precise mathematical description of programming languages, useful in designing and implementing languages as well as reasoning about programs. For example, advances in denotational semantics have led to clarifications of features, to more consistent programming language design, and to new programming constructs.

It has long been recognized, however, that traditional denotational semantics lacks modularity and extensibility (Mosses, 1984) (Lee, 1989). This is regarded as a major obstacle in applying denotational semantics to realistic programming languages.

In this paper, we take advantage of a new development in programming language theory—a monadic approach (Moggi, 1990) to structured denotational semantics. The resulting *modular monadic semantics* achieves a high level of modularity and extensibility. It is able to capture individual programming language features in reusable building blocks, and to specify programming languages by composing the necessary features.

Because modular monadic semantics is no more than a structured denotational semantics, standard equational reasoning methods still apply. In addition, we show that modular monadic semantics further facilitates reasoning by allowing us to specify axioms of programming language features and to construct reusable modular proofs.

Modular monadic semantics can be implemented using modern programming languages such as Haskell (Hudak *et al.*, 1992), ML (Milner *et al.*, 1990), or Scheme (Clinger & Rees, 1991). The result is a modular interpreter. We have discovered, however, that the relatively new idea of *constructor classes* in Gofer (and Haskell 1.3) are particularly suitable for representing some rather complex typing relationships in modular interpreters, and thus we choose Gofer for the interpreter described in Section 4. Our work is also directly applicable to semantics-directed compiler construction, and we present a compilation method based on monadic semantics and monadic program transformations.

Before introducing modular monadic semantics, in the next section we give an example to demonstrate the lack of modularity in traditional denotational semantics. The presentation follows the traditional denotational semantics style, augmented with a types declaration syntax similar to that of Haskell or ML. We assume the reader has basic knowledge of denotational semantics and functional programming.

1.2 The Lack of Modularity in Denotational Semantics

Let us first look at the denotational semantics of a simple arithmetic language:

$$\begin{array}{lcl} E & : & \mathit{Term} \rightarrow \mathit{Value} \\ E[\![n]\!] & = & n \\ E[\![e_1 + e_2]\!] & = & E[\![e_1]\!] + E[\![e_2]\!] \end{array}$$

Denotational semantics maps *terms* in the source language into *values* in the meta language. The source language terms are enclosed in “$[\![\ ]\!]$”. The n and $+$ symbols on the right hand side correspond to the meta language concepts of a *number* and the *add* arithmetic operation.

An important measure of modularity is how a semantic description can be extended to incorporate new programming language features. For example, if we extend the source language with variables and functions, we need to introduce an environment—a mapping from variable names to values:

$$\begin{array}{lcl} E & : & \mathit{Term} \rightarrow \mathit{Env} \rightarrow \mathit{Value} \\ E[\![n]\!] & = & \lambda\rho.n \\ E[\![e_1 + e_2]\!] & = & \lambda\rho.E[\![e_1]\!]\rho + E[\![e_2]\!]\rho \\ E[\![v]\!] & = & \lambda\rho.\rho[\![v]\!] \end{array}$$

Note that even though numbers are independent of the environment, we must change the semantics of numbers to accommodate the newly introduced environment argument. Similarly, the environment argument must be passed recursively to the subexpressions of $e_1 + e_2$, even though the arithmetic operation itself is independent of the environment.

If we further add continuations to our semantics (for supporting, for example, the sequencing operator “;”), we must change the semantics of numbers once again:

$$\begin{array}{lcl}
E & : & Term \to Env \to (Value \to Ans) \to Ans \\
E[\![n]\!] & = & \lambda\rho.\lambda k.kn \\
E[\![e_1 + e_2]\!] & = & \lambda\rho.\lambda k.E[\![e_1]\!]\rho(\lambda i.E[\![e_2]\!]\rho(\lambda j.k(i+j))) \\
E[\![e_1; e_2]\!] & = & \lambda\rho.\lambda k.E[\![e_1]\!]\rho(\lambda x.E[\![e_2]\!]\rho k)
\end{array}$$

In summary, we must make global changes to the traditional denotational semantics in order to add new features into the source language. This lack of modularity of denotational semantics has long been recognized (Mosses, 1984) (Lee, 1989), and is regarded by many as the most significant obstacle in applying denotational semantics to realistic programming languages.

1.3 Monads to the Rescue

Consider now a type constructor M and two functions:

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

The intuitive meanings of these operations are as follows:

- $M\ a$ is a *computation* returning a value of type a.
- *bind* c_1 $(\lambda v.c_2)$ 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.

With these operations we can rewrite the semantics for arithmetic expressions as follows:

$$\begin{array}{lcl}
E & : & Term \to M\ Value \\
E[\![n]\!] & = & return\ n \\
E[\![e_1 + e_2]\!] & = & bind\ (E[\![e_1]\!]) \\
 & & \quad (\lambda i.\ bind\ (E[\![e_2]\!]) \\
 & & \qquad\quad (\lambda j.\ return\ (i+j)))
\end{array}$$

Note now that the semantic function E maps terms to *computations* (of type $M\ Value$). The above equations can be read: "the semantics of $E[\![n]\!]$ is a trivial computation that returns n as result; the semantics of $E[\![e_1 + e_2]\!]$ is a computation that computes $E[\![e_1]\!]$, binds the result to i, computes $E[\![e_2]\!]$, binds the result to j, and finally returns $i + j$."

We call this a *parameterized semantics* because, depending on how we instantiate M, *return* and *bind*, we get different concrete semantics. For example, Figure 1 shows how the arithmetic semantics can be instantiated to the trivial and environment-based semantics described earlier. To give meaning to variables in the context of the environment-based semantics, we simply add the equation:

$$\begin{array}{lcl}
E[\![v]\!] & = & bind\ (rdEnv) \\
 & & \quad (\lambda\rho.\ return(\rho[\![v]\!]))
\end{array}$$

where *rdEnv* (defined in a later section) is a computation that reifies the environment. The key point here is that the previous equations did not need to be altered. In a similar way, we show later that appropriate definitions of M, *return* and *bind* can yield the continuation-based semantics discussed earlier, as well as several other important semantics to support other programming language features.

The type constructor M, together with the two functions *return* and *bind*, are called a *monad*, and a parameterized semantics using monads is called a *monadic semantics.* A monadic semantics can be instantiated using different *underlying monads.* In general, to add a new feature to a monadic semantics, we only need to add a semantic description of the new feature and change the underlying monad, but no changes are required of the semantic descriptions of the existing features.

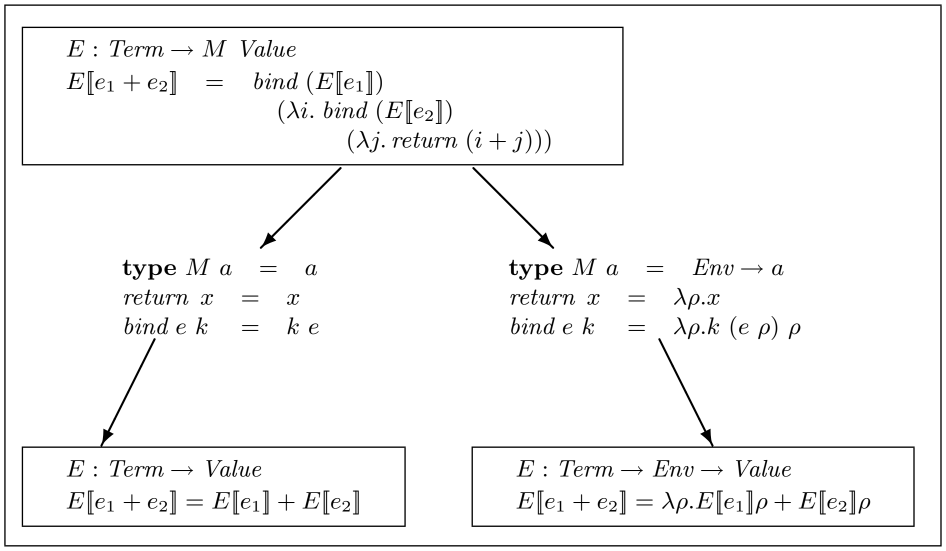
$E : Term \rightarrow M \; Value$
$E[\![e_1 + e_2]\!] = bind\ (E[\![e_1]\!])\ (\lambda i.\ bind\ (E[\![e_2]\!])\ (\lambda j.\ return\ (i + j)))$
$\textbf{type}\ M\ a = a$
$return\ x = x$
$bind\ e\ k = k\ e$
$\textbf{type}\ M\ a = Env \rightarrow a$
$return\ x = \lambda\rho.x$
$bind\ e\ k = \lambda\rho.k\ (e\ \rho)\ \rho$
$E : Term \rightarrow Value$
$E[\![e_1 + e_2]\!] = E[\![e_1]\!] + E[\![e_2]\!]$
$E : Term \rightarrow Env \rightarrow Value$
$E[\![e_1 + e_2]\!] = \lambda\rho.E[\![e_1]\!]\rho + E[\![e_2]\!]\rho$

Fig. 1. A parameterized arithmetic semantics

1.4 Background and Organization of the Paper

This paper explores the theory and practical applications of monads and monadic semantics, building on previous work in this area. The concept of monads originates in category theory (Mac Lane, 1971). The formulation of monads using a triple (*bind*, *return*, and the type constructor) is due to Kleisli. Moggi (Moggi, 1990) first proposed that monads provided a useful tool for structuring denotational semantics. Early work by Wadler (Wadler, 1990) showed the relationships between monads and functional programming. Recently, there has been a great deal of interest in using monads to construct modular semantics and modular interpreters (Wadler, 1992) (Jones & Duponcheel, 1993) (Espinosa, 1993) (Steele Jr., 1994).

In Section 2, we present the modular monadic semantics for a wide range of programming language features. We demonstrate how *monad transformers* capture individual features, and how *liftings* capture the interactions between different features.

In Section 3, we investigate the theory of monads and monad transformers. This includes, for example, the formal properties of monad transformers and liftings. We use monad laws and axioms to perform equational reasoning at a higher level than in traditional denotational semantics.

In Section 4, we demonstrate how the formal concepts of monads and monad transformers fit nicely into the Gofer (Jones, 1991) type system. By implementing modular monadic semantics in Gofer, we obtain a modular interpreter.

Finally, in Section 5, we apply monadic semantics to semantics-directed compilation. We show how an effective and provably correct complication scheme can be derived, taking advantage of the modularity and reasoning power of the monadic framework. We put some of our ideas to test by building a retargeted Haskell compiler.

Throughout the paper, we use a common source language to address various issues in monadic semantics, modular interpreters, and compilation. This source language is introduced in the next section.

1.5 The Source Language

The source language we consider has a variety of features, including different function call mechanisms, imperative features, first-class continuations, tracing (for debugging), and nondeterminism.

$$
\begin{array}{lcll}
e & ::= & n \mid e_1 + e_2 & \text{(arithmetic operations)} \\
 & \mid & v \mid \lambda v.e & \text{(variables and functions)} \\
 & \mid & (e_1\ e_2)_n & \text{(call-by-name)} \\
 & \mid & (e_1\ e_2)_v & \text{(call-by-value)} \\
 & \mid & (e_1\ e_2)_l & \text{(lazy evaluation)} \\
 & \mid & \text{callcc} & \text{(first-class continuations)} \\
 & \mid & e_1 := e_2 \mid \text{ref } e \mid \text{deref } e & \text{(imperative features)} \\
 & \mid & \mathit{label} \mathbin{@} e & \text{(trace labels)} \\
 & \mid & \{e_0, e_1, \ldots\} & \text{(nondeterminism)}
\end{array}
$$

To simplify the presentation, we use one form of function abstraction that can be applied using any of the three function application mechanisms: call-by-name, call-by-value, and lazy evaluation. We can observe the differences with the help of trace messages. For example, evaluating:

$$((\lambda x.x + x)(l \mathbin{@} 1))_n$$

results in 2 after printing the trace message "l" twice, whereas

$$((\lambda x.x + x)(l \mathbin{@} 1))_v$$

prints "l" only once. Nondeterminism is captured by returning all possible results. For example:

$$\{1, 3\} + \{2, 5\}$$

results in $\{3, 6, 5, 8\}$.

Although there is no single programming language that has all of the features of our source language—indeed, one could argue that this would not be a very good language design—it is nevertheless an excellent test of our methodology.

1.6 A Notation

For clarity, we adopt the following short-hand for monadic sequencing:

$$
\boxed{
\begin{array}{lcl}
\multicolumn{3}{l}{E : \mathit{Term} \to M\ \mathit{Value}} \\
E[\![e_1 + e_2]\!] & = & \mathit{bind}\ (E[\![e_1]\!]) \\
 & & \quad (\lambda i.\ \mathit{bind}\ (E[\![e_2]\!]) \\
 & & \qquad (\lambda j.\ \mathit{return}\ (i + j)))
\end{array}
}
$$

$$\Downarrow$$

$$
\boxed{
\begin{array}{lcl}
\multicolumn{3}{l}{E : \mathit{Term} \to M\ \mathit{Value}} \\
E[\![e_1 + e_2]\!] & = & \{\ i \leftarrow E[\![e_1]\!]; \\
 & & \ \ j \leftarrow E[\![e_2]\!]; \\
 & & \ \ \mathit{return}\ (i + j)\}
\end{array}
}
$$

This notation is similar to the "do" syntax in Haskell 1.3 (Peterson & Hammond, 1996), and is also somewhat similar to monad comprehensions (Wadler, 1990). It is important to remember that, despite the imperative feel, the monadic semantics is still made up of lambda abstractions and applications. We use *bind* and its short-hand notation interchangeably, depending on whichever is more convenient in a given context.

2 Modular Monadic Semantics

In this section, we present the modular monadic semantics of our source language. Compared with traditional denotational semantics, our approach captures individual programming language features using modular building blocks.

Figure 2 shows how our modular monadic semantics is organized. High-level features are defined based on a set of "kernel-level" operations. The expression $e_1 := e_2$, for example, is interpreted by the low-level primitive operation *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* (Moggi, 1990) (Liang *et al.*, 1995). 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. However, monad transformers are defined as higher-order functions and our monadic semantics is no more than a structured version of denotational semantics, so conventional reasoning methods (such as β conversion) apply.

The modular monadic semantics is composed of two parts:

Modular Semantic Building Blocks Semantic building blocks (represented by rectangular blocks in Figure 2) define the monadic semantics of individual source language features. Semantic building blocks are independent of each other, although they are based on a common set of kernel-level operations. Two building blocks may be supported by the same kernel-level operation. For example, both assignments and lazy evaluation may use the same store.

Monad Transformers Monad transformers define the kernel-level operations in a modular way. Multiple monad transformers can be composed to form the underlying monad used by all semantic building blocks. In Figure 2, monad transformers that support environment, continuations, store, etc. are used to construct the underlying monad.

Modular semantic building blocks and monad transformers are the topics of the following sections.

2.1 Modular Semantic Building Blocks

Each modular semantic building block defines the monadic semantics for a particular source language feature. 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:

$$E \quad : \quad \mathit{Term} \rightarrow M\ \mathit{Value}$$

where M is a monad equipped with two basic operations:

$$\begin{array}{lcl} \mathit{bind} & : & M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b \\ \mathit{return} & : & a \rightarrow M\ a \end{array}$$

Value is the domain sum of basic values and functions; and $M\ \mathit{Value}$ represents computations that return *Value* as result. Functions map computations to computations:

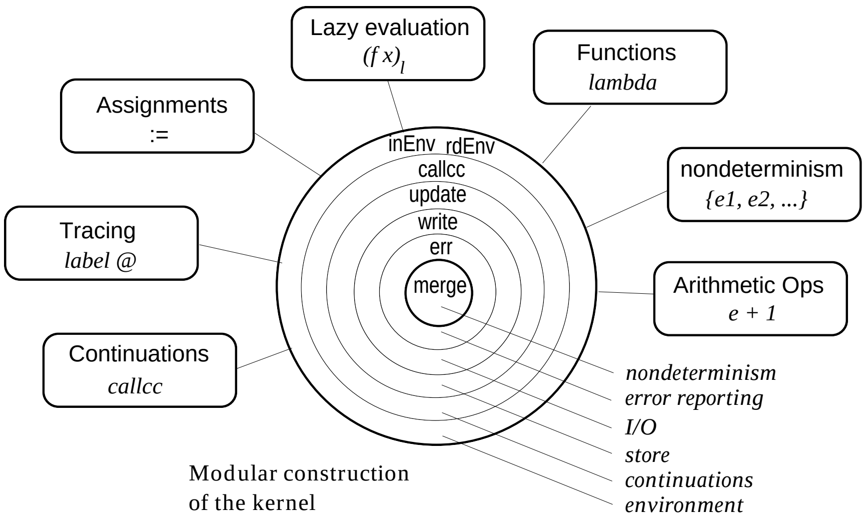

Fig. 2. The organization of modular monadic semantics

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

As will be seen, this generality allows us to model call-by-name, call-by-value and lazy evaluation with only one kind of lambda abstraction (but 3 kinds of function application) in the source language.

In this section, we present the semantic building blocks needed for our source language. The monad operations *return* and *bind* are the basic operations used by every building block. In addition, each semantic building block depends on several other kernel-level operations that are specific to its purpose.

2.1.1 The Arithmetic Building Block

The semantics for arithmetic expressions is as follows:

```
$E[\![n]\!]$           =  $\mathit{return}\ (\mathbf{in}_{Int}\ n)$
$E[\![e_1 + e_2]\!]$   =  { $v_1 \leftarrow E[\![e_1]\!]$;
                            $v_2 \leftarrow E[\![e_2]\!]$;
                            if ($\mathbf{is}_{Int}\ v_1$ and $\mathbf{is}_{Int}\ v_2$) then
                                $\mathit{return}\ (\mathbf{in}_{Int}\ (\mathbf{out}_{Int}\ v_1 + \mathbf{out}_{Int}\ v_2))$
                            else
                                $\mathit{err}$ “type error” }
```

$\mathbf{in}_{Int}$ is the injection function from *Int* to the *Value* domain, whereas $\mathbf{out}_{Int}$ is the projection function from the *Value* domain to *Int*. The kernel-level function (to be defined later):

$$\mathit{err}\ :\ \mathit{String} \to M\ a$$

reports error conditions (which, in this case, are type errors). In other words, *err* is an operation supported by the underlying monad M. For clarity, from now on we omit domain injection/projection and type checking.

$E[\![n]\!]$ returns the number n (injected into the *Value* domain) as the result of a trivial computation. To evaluate $e_1 + e_2$, we evaluate e_1 and e_2 in turn, and then sum the results.

2.1.2 The Function Building Block

In denotational semantics, functions are supported using an environment—a mapping from variable names to their denotation. We introduce an environment *Env* which maps variable names to computations,[†] and two kernel-level operations that retrieve the current environment and perform a computation in a given environment, respectively:

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

The definitions of *rdEnv* and *inEnv* are given later. The semantics for variables, function abstraction, call-by-name and call-by-value are as follows:

$$\begin{array}{lcl} E[\![v]\!] & = & \{\rho \leftarrow rdEnv; \rho\ [\![v]\!]\} \\ E[\![\lambda v.e]\!] & = & \{\rho \leftarrow rdEnv; return(\lambda c.inEnv\ \rho[c/[\![v]\!]]\ E[\![e]\!])\} \\ E[\![(e_1\ e_2)_n]\!] & = & \{f \leftarrow E[\![e_1]\!]; \rho \leftarrow rdEnv; f(inEnv\ \rho\ E[\![e_2]\!])\} \\ E[\![(e_1\ e_2)_v]\!] & = & \{f \leftarrow E[\![e_1]\!]; v \leftarrow E[\![e_2]\!]; f(return\, v)\} \end{array}$$

Because there is no risk of confusion, we drop the parentheses around $\rho[c/v]$ and $E[\![e]\!]$ in the application of *inEnv*.

The difference between call-by-value and call-by-name is clear: the former reduces the argument before invoking the function,[‡] whereas the latter packages the argument with the current environment to form a closure.

2.1.3 The References and Assignment Building Block

Imperative features can be supported using a store—a mapping from locations (of type *Loc*) to computations. Three functions allocate, read from and write to the memory cells in the store:

$$\begin{array}{lcl} alloc & : & M\ \ Loc \\ read & : & Loc \to M\ \ Value \\ write & : & (Loc,\ M\ \ Value) \to M\ \ () \end{array}$$

The monadic semantics for references and assignment is as follows:

$$\begin{array}{lcl} E[\![\mathrm{ref}\ e]\!] & = & \{v \leftarrow E[\![e]\!]; l \leftarrow alloc; write\ (l, return\ \ v); return\ \ l\} \\ E[\![\mathrm{deref}\ e]\!] & = & \{l \leftarrow E[\![e]\!]; read\ l\} \\ E[\![e_1 := e_2]\!] & = & \{l \leftarrow E[\![e_1]\!]; v \leftarrow E[\![e_2]\!]; write\ (l, return\ \ v)\} \end{array}$$

To create a reference, we evaluate the expression, allocate a new memory cell, and store in the location of the memory cell a trivial computation that returns the value of the expression. The argument of deref evaluates to a location, at which the stored value can be read. To assign an expression to a location, we evaluate the expression, and update the location with a trivial computation that returns the value of the expression.

Note that we only store trivial computations. We could alternatively give the semantics for references and assignment using a store that maps locations to values, rather than locations to computations. The reason we store computations is to simplify the overall

[†] We do not need an environment that maps names to computations in order to support call-by-value. However, we need such an environment to support call-by-name and lazy evaluation. We discuss this issue in more detail in Section 2.1.8.

[‡] To be precise, the call-by-value semantics is only preserved when the underlying monad enforces an evaluation order dependency. This is true of the continuation, state, and error monads. However, the identity and environment monads do not actually force the evaluation of c_1 before c_2 in $\{x \leftarrow c_1; c_2\}$.

presentation; in particular, it allows us to avoid introducing a separate kernel-level store operation for our next feature: lazy evaluation.

2.1.4 The Lazy Evaluation Building Block

Using the same store for references and assignments, we can implement lazy evaluation whose operational semantics implies *caching* of results.

$$\begin{array}{lcl} E[\![(e_1\ e_2)_l]\!] & = & \{\ f \leftarrow E[\![e_1]\!];\\ & & \quad l \leftarrow \mathit{alloc};\\ & & \quad \rho \leftarrow \mathit{rdEnv};\\ & & \quad \mathbf{let}\ \mathit{thunk} = \{\ v \leftarrow \mathit{inEnv}\ \rho\ E[\![e_2]\!];\\ & & \qquad\qquad\qquad \_ \leftarrow \mathit{write}\ (l, \mathit{return}\ v);\\ & & \qquad\qquad\qquad \mathit{return}\ v\ \}\\ & & \quad \mathbf{in}\ \{\ \_ \leftarrow \mathit{write}\ (l, \mathit{thunk});\\ & & \qquad\quad f\ (\mathit{read}\ l)\ \}\ \} \end{array}$$

Before entering the function, we allocate a memory cell and store a thunk (a computation that updates itself) in it. After the argument is first evaluated, the result is stored back to the memory cell, overwriting the thunk itself.

2.1.5 The Program Tracing Building Block

Given a kernel-level function:

$$\mathit{output} \quad : \quad \mathit{String} \to M\ ()$$

that prints out a string, we can support tracing. Labels attached to expressions cause a “trace record” to be invoked whenever that expression is evaluated:

$$\begin{array}{lcl} E[\![l\ @\ e]\!] & = & \{\ \_ \leftarrow \mathit{output}\ (\text{“enter ”} +\!\!+\ l);\\ & & \quad v \leftarrow E[\![e]\!];\\ & & \quad \_ \leftarrow \mathit{output}\ (\text{“leave ”} +\!\!+\ l);\\ & & \quad \mathit{return}\ v\ \} \end{array}$$

Here we see that some of the features of monitoring semantics (Kishon *et al.*, 1991) are easily incorporated into our framework.

2.1.6 The Continuation Building Block

The continuation is a powerful mechanism for modeling control flow in denotational semantics (Stoy, 1977). In addition, *callcc* (“call with current continuation”) is a useful programming language construct, popularized by its use in Scheme (Clinger & Rees, 1991). Here is a simple example to show how *callcc* works:

$$\mathit{callcc}\ (\lambda k.(k\ 100)_v) \implies 100$$

When applied to a function, *callcc* captures the current continuation, and passes the continuation as the argument k. The continuation itself is captured as a function. When captured continuation is later applied to the value 100, the control flow is transferred back to the point where the continuation was initially captured. The value (100) passed to the continuation is the result returned from *callcc*.

The power of *callcc* lies in that the captured continuation does not have to be invoked immediately. We may store the continuation into data structures, perform other computations, and then invoke the stored continuation to transfer the control back to where we issued *callcc*. For this reason, *callcc* can be used to model a wide variety of non-local control flow, including, for example, catch/throw, error handling, coroutines, and thread context switches. Scheme (Clinger & Rees, 1991) and SML (Milner *et al.*, 1990) incorporate *callcc* as a language feature.

As expected, the kernel-level operation *callcc* takes a function argument that in turn takes a continuation:

$$\mathit{callcc} \quad : \quad ((\mathit{Value} \rightarrow M \;\; \mathit{Value}) \rightarrow M \;\; \mathit{Value}) \rightarrow M \;\; \mathit{Value}$$

We define the semantics of source-level *callcc* as a function expecting another function as an argument, to which the current continuation is passed:

$$E[\![\mathrm{callcc}]\!] \quad = \quad \mathit{return}\ (\lambda f.\{f' \leftarrow f; \mathit{callcc}(\lambda k.f'(\lambda a.\{x \leftarrow a; kx\}))\})$$

The result of $E[\![\mathrm{callcc}]\!]$ is a trivial computation that returns a function. The argument of the function, f, evaluates to the current continuation (f').

2.1.7 The Nondeterminism Building Block

Given a kernel-level function:

$$\mathit{merge} \quad : \quad \mathit{List}\ (M \;\; a) \rightarrow M \;\; a$$

that merges a list of computations into a single (nondeterministic) computation, nondeterminism semantics can be expressed as:

$$E[\![\{e_0, e_1, \ldots\}]\!] \quad = \quad \mathit{merge}\ [E[\![e_0]\!], E[\![e_1]\!], \ldots]$$

$E[\![e_0]\!]$, $E[\![e_1]\!]$, etc. are a list of computations denoting the nondeterministic behavior.

2.1.8 Alternative Definitions of the Environment and Store

In the building blocks presented so far, we have used one environment that maps variable names to computations, and used one store that maps locations to computations. As we have pointed out, this generality is not necessary for some of the building blocks. For example, the call-by-value semantics only needs an environment that maps variable names to values, whereas the reference and assignment semantics only needs a store for values.

Modular monadic semantics is flexible enough that we can easily introduce multiple environments and stores, so that each building block is supported by exactly the right set of operations. To specify call-by-value functions, for example, we can use an environment that maps variable names to values. If we later add call-by-name functions, we simply add a new environment that maps variable names to computations. Similarly for the reference and assignment building block, we can introduce a store that maps locations to values, separate from the requirements of lazy evaluation.

If we were to store variables in two separate environments, we would then need to distinguish, at the source language level, call-by-value functions from call-by-name functions. Thus instead of using one syntax for all three kinds of function abstractions (as in Section 1.5), we would need to have two separate syntactic constructs: one for call-by-value, the other for call-by-name and lazy evaluation. Variables would then be stored in either of the two environments, depending on what kind of function abstraction the variable is introduced in.

We do not present the details of designing a modular semantics with multiple environments and stores. Instead, we emphasize that the simplifications we made in previous sections to ease the presentation do not fundamentally limit the modularity of our approach.

Feature	**Function**
Error reporting	$err : String \to M\ a$
Environment	$rdEnv : M\ Env$ $inEnv : Env \to M\ a \to M\ a$
Store	$alloc : M\ Loc$ $read : Loc \to M\ Value$ $write : (Loc, M\ Value) \to M\ ()$
Output	$output : String \to M\ ()$
Continuations	$callcc : ((a \to M\ b) \to M\ a) \to M\ a$
Nondeterminism	$merge : List\ (M\ a) \to M\ a$

Table 1. *Monad operations used in the semantics*

2.2 Monads With Operations

Semantic building blocks depend on other kernel-level operations in addition to *unit* and *bind*. From the last section, it is clear that the operations listed in Table 1 must be supported.

If we were writing the semantics in the traditional way, now would be the time to set up the domains and define the functions listed in the table. The major drawback of such a monolithic approach is that we have to take into account all other features when we define an operation for one specific feature. When we define *callcc*, for example, we have to decide how it interacts with the store and environment etc. And, if we later want to add more features, the semantic domains and all kernel-level functions may have to be redefined.

Monad transformers, on the other hand, allow us to capture individual language features. Furthermore, the concept of *lifting* allows us to account for the interactions between various features. Monad transformers and lifting are the topics of the next two sections.

To simplify the set of operations, we note that both the store and output (used by the program tracing building block) have to do with some notion of *state*. Thus we could define *alloc*, *read*, *write*, and *output* in terms of the function:

$$update \quad : \quad (s \to s) \to M\ s$$

for some suitably chosen state type *s*. We can read the state by passing *update* the identity function, and update the state by passing it a state transformer. For example, we can model output by using *String* as the state type:

$$\begin{array}{lcl} output & : & String \to m\ () \\ output\ msg & = & \{\ \_ \leftarrow update\ (\lambda\ sofar.sofar \mathbin{++} msg); \\ & & \quad return\ ()\} \end{array}$$

The underscore (_) indicates that the return value of *update* is ignored.

2.3 Monad Transformers

For an intuitive understanding of monad transformers, consider the merging of a state monad with an arbitrary monad, an example which originally appeared in Moggi's note (Moggi, 1990):

$$\textbf{type}\ \mathit{StateT}\ s\ m\ a \quad = \quad s \to m\ (s, a)$$

The type variable m represents a type constructor. We later show that, if m is a monad, then so is *StateT* s m. Therefore *StateT* s is a monad transformer. For example, if we substitute the identity monad:

$$\textbf{type}\ \mathit{Id}\ a \quad = \quad a$$

for m in the above monad transformer, then we arrive at:

$$\begin{array}{lcl} \mathit{StateT}\ s\ \mathit{Id}\ a & = & s \to \mathit{Id}\ (s, a) \\ & = & s \to (s, a) \end{array}$$

which is the standard state monad found, for example, in Wadler's work (Wadler, 1992).

We formally define monad transformers in Section 3.1.2. For now we note that a monad transformer t has a number of capabilities:

First, it transforms any monad m to monad t m. Functions $\mathit{return}_{t\ m}$ and $\mathit{bind}_{t\ m}$ are naturally defined in terms of return_m and bind_m.

Second, it can embed any computation in monad m as a computation in monad t m. Every monad transformer is equipped with a function:

$$\mathit{lift}_t \quad : \quad m\ a \to t\ m\ a$$

which maps any computation in monad m to a computation in monad t m.

Third, it adds operations (i.e., introduces new features) to a monad. The *StateT* monad transformer, for example, adds state s to the monad it is applied to, and the resulting monad accepts *update* as a legitimate operation.

Lastly, monad transformers compose easily. For example, applying both *StateT* s_1 and *StateT* s_2 to the identity monad, we get:

$$\begin{array}{lcl} \mathit{StateT}\ s_1\ (\mathit{StateT}\ s_2\ \mathit{Id})\ a & = & s_1 \to (\mathit{StateT}\ s_2\ \mathit{Id})\ (s_1, a) \\ & = & s_1 \to s_2 \to (s_2, (s_1, a)), \end{array}$$

which is the expected type signature for transforming both states s_1 and s_2. The observant reader will note, however, an immediate problem: in the resulting monad, which state does *update* act upon? In general, this is the problem of *lifting* monad operations through transformers, and is addressed in the next section.

The remainder of this section introduces the monad transformers that cover all the features listed in Table 1. Some of these (*StateT*, *ContT*, and *ErrorT*) appear in an abstract form in Moggi's note (Moggi, 1990). The *environment* monad is similar to the *state reader* by Wadler (Wadler, 1990). The *state* and *environment* monad transformers are related to ideas found in Jones and Duponcheel's work (Jones, 1993) (Jones & Duponcheel, 1993).

We attach subscripts to monadic operations to distinguish between the different monads they operate on. Some monad transformers use two additional functions: *map* and *join*. These functions, which can be used in any monad, are easily defined in terms of *return* and *bind*:

$$\begin{array}{lcl} \mathit{map}_m & : & (a \to b) \to m\ a \to m\ b \\ \mathit{map}_m\ f\ e & = & \mathit{bind}_m\ e\ (\lambda a.\ \mathit{return}_m\ (f\ a)) \end{array}$$

$$\begin{array}{lcl} \mathit{join}_m\ z & : & m\ (m\ a) \to m\ a \\ \mathit{join}_m\ z & = & \mathit{bind}_m\ z\ (\lambda a.a) \end{array}$$

2.3.1 The State Monad Transformer

The state monad transformer introduces an updatable state into an existing monad. The resulting monad accepts an additional operation *update*, and is called a *state monad*.

Previously, we described the state monad transformer with a type definition:

$$\textbf{type}\ \mathit{StateT}\ s\ m\ a \quad = \quad s \rightarrow m\ (s, a)$$

To complete the definition, we must also provide the *return* and *bind* functions for *StateT s m*:

$$\begin{aligned}
\mathit{return}_{\mathit{StateT}\ s\ m} &= \lambda s.\, \mathit{return}_m\ (s, x) \\
\mathit{bind}_{\mathit{StateT}\ s\ m}\ m\ k &= \lambda s_0 . \mathit{bind}_m\ (m\ s_0)\ (\lambda(s_1, a).k\ a\ s_1)
\end{aligned}$$

Given these definitions, if return_m, bind_m, and m form a monad, then so do $\mathit{return}_{\mathit{StateT}\ s\ m}$, $\mathit{bind}_{\mathit{StateT}\ s\ m}$ and *StateT s m*. A more formal characterization of the relationships between m and *StateT s m* is given in Section 3.

Next, we define the *lift* function, which simply performs the computation in the new context and preserves the state.

$$\begin{aligned}
\mathit{lift}_{\mathit{StateT}\ s} &: m\ a \rightarrow \mathit{StateT}\ s\ m\ a \\
\mathit{lift}_{\mathit{StateT}\ s}\ c &= \lambda s.\{x \leftarrow c; \mathit{return}_m\ (s, x)\}_m
\end{aligned}$$

Finally, a state monad must support the *update* operation, which transforms the state using the given *f*, and returns the old state:

$$\begin{aligned}
\mathit{update}_{\mathit{StateT}\ s\ m} &: (s \rightarrow s) \rightarrow \mathit{StateT}\ s\ m\ s \\
\mathit{update}_{\mathit{StateT}\ s\ m}\ f &= \lambda s.\, \mathit{return}_m\ (f\ s, s)
\end{aligned}$$

2.3.2 The Environment Monad Transformer

EnvT r transforms any monad into an *environment monad* that supports *inEnv* and *rdEnv*. The definition of *bind* shows that two subsequent computation steps run under the same environment ρ (of type r). (Compare this with the state monad, where the second computation is run in the state returned by the first computation.)

$$\textbf{type}\ \mathit{EnvT}\ r\ m\ a \quad = \quad r \rightarrow m\ a$$

$$\begin{aligned}
\mathit{return}_{\mathit{EnvT}\ r\ m}\ a &= \lambda\rho.\, \mathit{return}_m\ a \\
\mathit{bind}_{\mathit{EnvT}\ r\ m}\ m\ k &= \lambda\rho.\mathit{bind}_m\ (m\ \rho)\ (\lambda a.k\ a\ \rho)
\end{aligned}$$

The result of lifting a computation through the environment monad is a computation that ignores its environment.

$$\begin{aligned}
\mathit{lift}_{\mathit{EnvT}\ r} &: m\ a \rightarrow \mathit{EnvT}\ r\ m\ a \\
\mathit{lift}_{\mathit{EnvT}\ r}\ c &= \lambda\rho.c
\end{aligned}$$

InEnv ignores the environment carried inside the monad, and performs the computation in the given environment.

$$\begin{aligned}
\mathit{inEnv}_{\mathit{EnvT}\ r\ m} &: r \rightarrow \mathit{EnvT}\ r\ m\ a \rightarrow \mathit{EnvT}\ r\ m\ a \\
\mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho\ m &= \lambda\rho'.m\ \rho
\end{aligned}$$

$$\begin{aligned}
\mathit{rdEnv}_{\mathit{EnvT}\ r\ m} &: \mathit{EnvT}\ r\ m\ r \\
\mathit{rdEnv}_{\mathit{EnvT}\ r\ m} &= \lambda\rho.\, \mathit{return}_m\ \rho
\end{aligned}$$

2.3.3 The Error Monad Transformer

Monad *Err* completes a series of computations if all succeed, or aborts as soon as an error occurs. The monad transformer *ErrT* transforms a monad into an *error monad* that supports *err* as a valid operation.

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Err}\ a & = & \mathit{Ok}\ a \mid \mathit{Err}\ \mathit{String} \\
\textbf{type}\ \mathit{ErrT}\ m\ a & = & m\ (\mathit{Err}\ a)
\end{array}$$

$$\begin{array}{lcl}
\mathit{return}_{\mathit{ErrT}\ m}\ a & = & \mathit{return}_m\ (\mathit{Ok}\ a) \\
\mathit{bind}_{\mathit{ErrT}\ m}\ m\ k & = & \mathit{bind}_m\ m\ (\lambda a.\textbf{case}\ a\ \textbf{of} \\
 & & \qquad (\mathit{Ok}\ x) \quad \to \quad k\ x \\
 & & \qquad (\mathit{Err}\ \mathit{msg}) \quad \to \quad \mathit{return}_m\ (\mathit{Err}\ \mathit{msg}))
\end{array}$$

To lift a computation across *ErrT*, we tag the result with *Ok*:

$$\begin{array}{lcl}
\mathit{lift}_{\mathit{ErrT}} & : & m\ a \to \mathit{ErrT}\ m\ a \\
\mathit{lift}_{\mathit{ErrT}} & = & \mathit{map}_m\ \mathit{Ok}
\end{array}$$

The semantic function *err* throws away any intermediate result, and returns the error value *Err*.

$$\begin{array}{lcl}
\mathit{err} & : & \mathit{String} \to \mathit{ErrT}\ m\ a \\
\mathit{err} & = & \mathit{return}_m \cdot \mathit{Err}
\end{array}$$

2.3.4 The Continuation Monad Transformer

We define the continuation monad transformer as:

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

$$\begin{array}{lcl}
\mathit{return}_{\mathit{ContT}\ c\ m}\ x & = & \lambda k.k\ x \\
\mathit{bind}_{\mathit{ContT}\ c\ m}\ m\ f & = & \lambda k.m\ (\lambda a.f\ a\ k)
\end{array}$$

ContT introduces an additional continuation argument (of type $a \to m\ c$), where c is the answer type. By the above definitions of *return* and *bind*, all computations in monad *ContT c m* are carried out in the continuation passing style.

Lift for *ContT c m* turns out to be the same as bind_m. (Indeed they have the same type signature.)

$$\begin{array}{lcl}
\mathit{lift}_{\mathit{ContT}\ c} & : & m\ a \to \mathit{ContT}\ c\ m\ a \\
\mathit{lift}_{\mathit{ContT}\ c} & = & \mathit{bind}_m
\end{array}$$

ContT transforms any monads to a *continuation monad*, which supports an additional operation *callcc*. *Callcc* f invokes the computation in f, passing it a continuation that, once applied, throws away the current continuation k' and invokes the captured continuation k.

$$\begin{array}{lcl}
\mathit{callcc}_{\mathit{ContT}\ c\ m} & : & ((a \to \mathit{ContT}\ c\ m\ b) \to \mathit{ContT}\ c\ m\ a) \to \mathit{ContT}\ c\ m\ a \\
\mathit{callcc}_{\mathit{ContT}\ c\ m}\ f & = & \lambda k.f\ (\lambda a.\lambda k'.k\ a)\ k
\end{array}$$

2.3.5 The List Monad

In denotational semantics, nondeterminism is usually captured by a list of all possible results. It is known that lists compose with a special kind of monads called *commutative monads* (Jones & Duponcheel, 1993). It is not clear, however, if lists compose with arbitrary monads. Since many useful monads (e.g. state, error and continuation monads) are not commutative, we cannot define a list monad transformer—one which adds the operation *merge* to any monad.

Fortunately, every other monad transformer we have considered in this paper properly transforms arbitrary monads. We thus can use lists as the *base* monad, to which other transformers can be applied. We recall the definition of the well-known list type and its monadic operations:

```
data List a  =  a : List a       -- Cons cell
             |  [ ]              -- Nil

return_List x  =  [x]
bind_List m k  =  case m of
                      [ ]       →  [ ]
                      (x : xs)  →  k x ++ (bind_List xs k)
```

The *merge* function of the *List* monad is the well-known list concatenation operation:

```
merge_List          :  List (List a) → List a
merge_List [ ]         =  [ ]
merge_List (x : xs)    =  x ++ merge_List xs
```

2.4 Liftings

We have introduced monad transformers that add useful operations to a given monad, but we have not addressed how these operations can be carried through other layers of monad transformers. This process is called the *lifting* of operations.

Lifting an operation f in monad m through a monad transformer t results in an operation whose type signature can be derived by substituting all occurrences of m in the type of f with $t\ m$. For example, lifting:

$$\mathit{inEnv} \ : \ r \to m\ a \to m\ a$$

through t results in an operation with type:

$$\mathit{inEnv} \ : \ r \to t\ m\ a \to t\ m\ a$$

Moggi (Moggi, 1990) studied the problem of lifting under a categorical context. The objective was to identify liftable operations from their type signatures. Unfortunately, many useful operations such as *merge*, *inEnv* and *callcc* failed to meet Moggi's criteria, and were left unsolved.

Instead, we consider how to lift these difficult cases individually. This allows us to make use of their definitions (rather than just their types), and to find ways to lift them through all of the monad transformers studied so far.

This is exactly where monad transformers provide us with an opportunity to study how various programming language features interact. The easy-to-lift cases correspond to features that are independent in nature, while the more involved cases require a deeper analysis of monad structures to clarify the semantics.

An unfortunate consequence of our approach is that, as we consider more monad transformers, the number of possible liftings grows quadratically. It seems, however, that there are not too many different kinds of monad transformers (although there may be many *instances* of the same monad transformer such as *StateT*). The monad transformers that we have introduced so far are able to model almost all commonly known features of sequential languages. [§]

Some operations are more difficult to lift than others. In particular, *inEnv* and *callcc* require special attention. We first list the easy cases, followed by the rest. Although we present a number of liftings in this section, we defer a formal explanation of why they are the desirable ones to Section 3.

2.4.1 The Easy Cases

RdEnv, *err* and *update* take a non-monadic type, and return a computation. They are handled by *lift*. For any monad transformer t applied to monad m, we have:

$$\begin{array}{lcl} \mathit{rdEnv}_{t\ m} & = & \mathit{lift}_t\ \mathit{rdEnv}_m \\ \mathit{err}_{t\ m} & = & \mathit{lift}_t \cdot \mathit{err}_m \\ \mathit{update}_{t\ m} & = & \mathit{lift}_t \cdot \mathit{update}_m \end{array}$$

Because *List* always is the base monad, we only have to consider cases when (possibly a sequence of) monad transformers are applied to *List*:

$$\mathit{merge}_{(t_1 \ldots (t_n\ \mathit{List}) \ldots)} \quad = \quad \mathit{join}_{(t_1 \ldots (t_n\ \mathit{List}) \ldots)} \cdot \mathit{lift}_{t_1} \cdot \ldots \cdot \mathit{lift}_{t_n}$$

2.4.2 Lifting Callcc

The crucial issue in lifting *callcc* through a monad transformer, for example, *EnvT* r, is to specify how it interacts with the newly introduced environment r. The following lifting discards the *current* environment ρ' upon invoking the captured continuation k. The execution will continue in the environment ρ captured when *callcc* was first invoked. This is indeed how SML's *callcc* normally interacts with the environment.

$$\begin{array}{lcl} \mathit{callcc}_{\mathit{EnvT}\ r\ m} & : & ((a \to r \to m\ b) \to r \to m\ a) \to r \to m\ a \\ \mathit{callcc}_{\mathit{EnvT}\ r\ m} & = & \lambda\rho.\mathit{callcc}_m(\lambda k.f(\lambda a.\lambda\rho'.ka)\rho) \end{array}$$

In lifting *callcc* through *StateT*, we have a choice of passing either the current state s_1 or the captured state s_0. The former is the usual semantics for *callcc*, and the latter is useful in Tolmach and Appel's approach to debugging (Tolmach & Appel, 1990).

$$\begin{array}{lcl} \mathit{callcc}_{\mathit{StateT}\ s\ m} & : & ((a \to s \to m(s,b)) \to s \to m(s,a)) \to s \to m(s,a) \\ \mathit{callcc}_{\mathit{StateT}\ s\ m}\ f & = & \lambda s_0.\mathit{callcc}_m\ (\lambda k.f\ (\lambda a.\lambda s_1.k\ (s_1,a))\ s_0) \end{array}$$

[§] An example of the features we cannot model is concurrent computation in multi-threaded programs. In addition, the state monad transformer is more general than what is needed to model output. The *output monad transformer* (Moggi, 1990) is also able to support the *output* operation:

$$\begin{array}{lcl} \textbf{type}\ \mathit{OutputT}\ m\ a & = & m\ (\mathit{String}, a) \\ \mathit{return}_{\mathit{OutputT}\ m}\ x & = & \mathit{return}_m\ ("\,", x) \\ \mathit{bind}_{\mathit{OutputT}\ m}\ m\ k & = & \{(o_1,a) \leftarrow m;\ (o_2,b) \leftarrow k\ a;\ \mathit{return}_m\ (o_1 +\!\!+\ o_2, b)\}_m \\ \mathit{lift}_{\mathit{OutputT}} & : & m\ a \to \mathit{OutputT}\ m\ a \\ \mathit{lift}_{\mathit{OutputT}}\ c & = & \{x \leftarrow c;\ \mathit{return}_m\ ("\,", x)\}_m \\ \mathit{output}_{\mathit{OutputT}\ m} & : & \mathit{String} \to \mathit{OutputT}\ m \\ \mathit{output}_{\mathit{OutputT}\ m}\ s & = & \mathit{return}_m\ (s, ()) \end{array}$$

Investigating the properties of *OutputT* and its relationship with *StateT* is a topic for future research.

The above shows the usual *callcc* semantics, and can be changed to the "debugging" version by instead passing (s_0, a) to k:

$$\mathit{callcc}_{\mathit{StateT}\ s\ m}\ f \quad = \quad \lambda s_0.\mathit{callcc}_m\ (\lambda k.f\ (\lambda a.\lambda s_1.k\ (s_0, a))\ s_0)$$

callcc can be lifted through *ErrT* as follows:

$$\begin{aligned} \mathit{callcc}_{\mathit{ErrT}\ m} &: ((a \to m(\mathit{Err}\ b)) \to m(\mathit{Err}\ a)) \to m(\mathit{Err}\ a) \\ \mathit{callcc}_{\mathit{ErrT}\ m}\ f &= \mathit{callcc}_m(\lambda k.f(\lambda a.k(\mathit{Ok}\ a))) \end{aligned}$$

2.4.3 Lifting InEnv

The liftings of *inEnv* through *EnvT* and *StateT* are similar:

$$\begin{aligned} \mathit{inEnv}_{\mathit{EnvT}\ r'\ m} &: r \to (r' \to m\ a) \to r' \to m\ a \\ \mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho\ e &= \lambda \rho'.\mathit{inEnv}_m\ \rho\ (e\ \rho') \end{aligned}$$

$$\begin{aligned} \mathit{inEnv}_{\mathit{StateT}\ s\ m} &: r \to (s \to m\ (s, a)) \to s \to m\ (s, a) \\ \mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho\ e &= \lambda s.\mathit{inEnv}_m\ \rho\ (e\ s) \end{aligned}$$

A function of type:

$$m\ a \to m\ a$$

maps m (*Err* a) to m (*Err* a), thus *inEnv* stays the same after being lifted through *ErrT*.

We do not know of a desirable way to lift *inEnv* through *ContT*. This means that we always have to apply the continuation monad transformer before we apply environment monad transformers. In the following lifting, for example, the environment is not restored when c invokes k, and would thus reflect the history of dynamic execution.

$$\begin{aligned} \mathit{inEnv}_{\mathit{ContT}\ c\ m}\ \rho\ c &= \lambda k.\mathit{inEnv}_m\ \rho\ (c\ k) \\ \mathit{rdEnv}_{\mathit{ContT}\ c\ m} &= \mathit{lift}\ \mathit{rdEnv}_m \end{aligned}$$

2.5 Summary

Monad transformers and lifting are summarized in Figures 3 and 4. The most problematic case is the continuation monad transformer *ContT*. Not only are operations relatively hard to lift though *ContT*, the *callcc* operation also requires more work to lift through other monad transformers.

Equipped with the monad transformers, we can construct the underlying monad M to support all of the semantic building blocks in Section 2.1:

```
type M a  =  EnvT Env                    (environment)
               (ContT Answer             (continuation)
                  (StateT Store          (store)
                     (StateT IO          (input/output)
                        (ErrT            (error reporting)
                           List)))) a    (nondeterminism)
```

Env, *Answer*, *Store*, and *IO* are the types of environment, answer, store, and I/O channels, respectively. The order of some monad transformers can be changed. However, because of the limitations in lifting *inEnv* through *ContT*, we cannot exchange the order of *EnvT* and *ContT*.

By using a series of abstractions, modular monadic semantics turns the monolithic structure of traditional denotational semantics into reusable components. The modularity is manifested at two levels, high-level monadic building blocks and low-level monad transformers.

State:

$$\textbf{type } \mathit{StateT}\ s\ m\ a = s \to m\ (s, a)$$

$$\begin{aligned}
&\mathit{return}_{\mathit{StateT}\ s\ m}\ a = \lambda s.\ \mathit{return}_m(s, a)\\
&\mathit{bind}_{\mathit{StateT}\ s\ m}\ e\ k =\\
&\qquad \lambda s.\{(s', a) \leftarrow es; kas'\}_m
\end{aligned}$$

$$\mathit{update}\ f = \lambda s.\ \mathit{return}_m(fs, s)$$

$$\mathit{lift}_{\mathit{StateT}\ s} e = \lambda s.\{a \leftarrow e; \mathit{return}_m(s, a)\}_m$$

Environment:

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

$$\begin{aligned}
&\mathit{return}_{\mathit{EnvT}\ r\ m}\ a = \lambda \rho.\ \mathit{return}_m\ a\\
&\mathit{bind}_{\mathit{EnvT}\ r\ m}\ e\ k =\\
&\qquad \lambda \rho.\{a \leftarrow e\rho; ka\rho\}_m
\end{aligned}$$

$$\begin{aligned}
\mathit{rdEnv} &= \lambda \rho.\ \mathit{return}_m\ \rho\\
\mathit{inEnv}\ \rho\ c &= \lambda \rho'.c\ \rho
\end{aligned}$$

$$\mathit{lift}_{\mathit{EnvT}\ r} e = \lambda \rho.e$$

Errors:

$$\begin{aligned}
&\textbf{type } \mathit{Err}\ a = \mathit{Ok}\ a \mid \mathit{Err}\ \mathit{String}\\
&\textbf{type } \mathit{ErrT}\ m\ a = m\ (\mathit{Err}\ a)
\end{aligned}$$

$$\begin{aligned}
&\mathit{return}_{\mathit{ErrT}\ m} = \mathit{return}_m \cdot \mathit{Ok}\\
&\mathit{bind}_{\mathit{ErrT}\ m} e\ k =\\
&\qquad \{\ a \leftarrow e;\\
&\qquad\quad \textbf{case } a \textbf{ of}\\
&\qquad\qquad \mathit{Ok}\ x \ \to\ kx\\
&\qquad\qquad \mathit{Err}\ s \ \to\ \mathit{return}_m(\mathit{Err}\ s)\}
\end{aligned}$$

$$\mathit{err} = \mathit{return}_m \cdot \mathit{Err}$$

$$\mathit{lift}_{\mathit{ErrT}} = \mathit{map}_m\, \mathit{Ok}$$

Continuation:

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

$$\begin{aligned}
&\mathit{return}_{\mathit{ContT}\ c\ m}\ a = \lambda k.ka\\
&\mathit{bind}_{\mathit{ContT}\ c\ m}\ e\ f = \lambda k.e(\lambda a.fak)
\end{aligned}$$

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

$$\mathit{lift}_{\mathit{ContT}\ c} = \mathit{bind}_m$$

Fig. 3. Monad transformers

We have, however, only achieved part of our goal. Without a theory of monads and monad transformers, we would have to unfold the definitions of all kernel-level monadic operations (such as *bind* and *inEnv*) to reason about semantic building blocks and the source language. In the next section, we present a theory that enables us to perform equational reasoning at a higher level with a set of laws and axioms.

3 A Theory of Monads and Monad Transformers

The purpose of developing a theory for monads and monad transformers is to reason about the monadic semantics without having to unfold the definitions of kernel-level monadic operations such as *bind*, *inEnv*, etc. Unfolding the monadic operations would defeat the purpose of the modular abstraction mechanism. Instead, we make it possible to perform equational reasoning at a high level by providing a set of properties directly associated with various monadic operations. An example in Section 5 further demonstrates that reasoning in the monadic framework offers modular proofs and more general results. In this section, we concentrate on the fundamental properties of monads and monad transformers.

We begin with the formal definition of monads and monad transformers, based on Moggi's and Walder's earlier work. The main topics of this section are how monadic axioms capture the properties of individual programming language features, and how natural liftings preserve existing features and capture the interactions between the newly added feature and existing features. The section ends with a discussion of the order of composing monad transformers.

Functions *err*, *update* and *rdEnv* are easily lifted using *lift*:

$$\begin{array}{lcl} err_{t\ m} & = & lift_{t\ m} \cdot err_m \\ update_{t\ m} & = & lift_{t\ m} \cdot update_m \\ rdEnv_{t\ )} & = & lift_{t\ m} rdEnv_m \end{array}$$

List can only be the base monad:

$$merge_{(t_1 \ldots (t_n\ List) \ldots)} \quad = \quad join_{(t_1 \ldots (t_n\ List) \ldots)} \cdot lift_{t_1} \cdot \ldots \cdot lift_{t_n}$$

Liftings of *callcc* and *inEnv*:

	$callcc_{t\ m}\ f$	$inEnv_{t\ m}\ \rho\ e$
$EnvT\ r\ m$	$\lambda\rho.callcc_m(\lambda k.f(\lambda a.\lambda\rho'.ka)\rho)$	$\lambda\rho'.inEnv_m\rho(e\rho')$
$StateT\ s\ m$	$\lambda s_0.callcc_m(\lambda k.f(\lambda a.\lambda s_1.k(s_0,a))s_0)$	$\lambda s.inEnv_m\rho(es)$
$ErrT\ m$	$callcc_m\ (\lambda k.f(\lambda a.k(Ok\ a)))$	$inEnv_m\ \rho\ e$

Fig. 4. Liftings

3.1 Monad and Monad Transformers

In this section we give a formal definition of monads and monad transformers.

3.1.1 Monads

Definition 3.1
A **monad** is a triple $(m, return_m, bind_m)$ consisting of a type constructor and two functions that satisfy the following laws (Moggi, 1990):

$$\begin{array}{rcll} \{b \leftarrow return\ a; k\ b\} & = & k\ a & \text{(left unit)} \\ \{a \leftarrow e; return\ a\} & = & e & \text{(right unit)} \\ \{v_1 \leftarrow e_1; \{v_2 \leftarrow e_2; e_3\}\} & = & \{v_2 \leftarrow \{v_1 \leftarrow e_1; e_2\}; e_3\} & \text{(associativity)} \end{array}$$

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 imperative programming languages.

Note that in the associativity law, e_1 is in the scope of v_2 on the right hand side but not so on the left hand side. In applying this law, we must make sure that there is no unwanted name capture.

The type constructors *Id* and *List* introduced in Section 2 are well-known monads (presented in, for example, (Wadler, 1990)):

Proposition 3.1
Id and *List* are monads.

3.1.2 Monad Transformers

To capture monad transformers formally, we first introduce *monad morphisms* (Moggi, 1990):

Definition 3.2

A **monad morphism** f between monads m and m' is a function of type:

$$f : m\ a \rightarrow m'\ a$$

satisfying:

$$\begin{array}{rcl} f.\,return_m & = & return_{m'} \\ f\ (bind_m\ m\ k) & = & bind_{m'}\ (f\ m)\ (f \cdot k) \end{array}$$

Note that f is polymorphic in a. We can now define monad transformers as follows:

Definition 3.3
A **monad transformer** consists of a type constructor t and an associated function $lift_t$, where t maps any given monad (m, $return_m$, $bind_m$) to a new monad ($t\ m$, $return_{t\ m}$, $bind_{t\ m}$). Furthermore, $lift_t$ is a monad morphism between m and $t\ m$:

$$lift_t : m\ a \rightarrow t\ m\ a$$

Therefore lifting a trivial computation results in a trivial computation; lifting a sequence of computations is equivalent to first lifting them individually, and then combining them in the lifted monad.
The type constructors listed in Figure 3 satisfy the above definition.

Proposition 3.2
EnvT r, *StateT* s, *ErrT*, and *ContT* c are monad transformers.

It is well known that these type constructors transform monads to monads. "*EnvT* r" is the composable reader monad presented in (Jones & Duponcheel, 1993). The remaining three were discovered by Moggi (Moggi, 1990). Appendix A contains detailed proofs that the corresponding *lift* functions are indeed monad morphisms.

Monad transformers compose with each other (a property that follows immediately from the definition of monad morphisms):

Proposition 3.3
Given monad transformers t_1 and t_2, $t_1 \cdot t_2$ is a monad transformer with:

$$\begin{array}{l} \textbf{type}\ (t_1 \cdot t_2)\ m\ a \ = \ t_1\ (t_2\ m)\ a \\ lift_{(t_1 \cdot t_2)} \ = \ lift_{t_1} \cdot lift_{t_2} \end{array}$$

3.2 Environment Axioms

Environments have a profound impact on programming language semantics and compilation. For example, lexically scoped languages fit well into the environment model. The monadic framework provides us a way to capture the essential properties of environments as follows:

Proposition 3.4
The environment operations, *rdEnv* and *inEnv* satisfy the following axioms:

$$\begin{array}{rcll} (inEnv\ \rho) \cdot return & = & return & \text{(unit)} \\ inEnv\ \rho\ \{v \leftarrow e_1; e_2\} & = & \{v \leftarrow inEnv\ \rho\ e_1; inEnv\ \rho\ e_2\} & \text{(distribution)} \\ inEnv\ \rho\ rdEnv & = & return\,\rho & \text{(cancellation)} \\ inEnv\ \rho'\ (inEnv\ \rho\ e) & = & inEnv\ \rho\ e & \text{(overriding)} \end{array}$$

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 supersedes an outer one (the overriding law). The distribution law, for example, is what distinguishes the environment from a store. A store does not distribute across a sequence of computations. It is updated as the computation progresses.

We can prove the environment axioms by first verifying that they hold after the environment monad transformer is applied, and then by making sure that they are preserved through the liftings of *rdEnv* and *inEnv*. A detailed proof of these results is included in Appendix A.

In Section 5, we will present an example that uses the environment axioms to prove a property about compiling the source language.

The environment axioms provide an answer to the question: "what constitutes an environment?" We expect that useful monadic axioms can be derived for other features, following the earlier efforts on state (Hudak & Bloss, 1985) (Peyton Jones & Wadler, 1993) (Chen & Hudak, 1997), continuations (Felleisen *et al.*, 1986) (Felleisen & Hieb, 1992) and exceptions (Spivey, 1990).

3.3 Natural Liftings

In this section, we investigate what conditions a desirable lifting must satisfy. First we will formalize how types are transformed in the lifting process. We will then introduce the *natural lifting condition* and verify that the liftings we constructed in Section 2.4 are indeed natural.

3.3.1 Lifting Types

How does its type change when an operation is lifted? The set of operations we consider has the following types in monad m:

$$\begin{array}{rcll}
\tau & ::= & A & \text{(type constants)} \\
 & | & a & \text{(type variables)} \\
 & | & \tau \rightarrow \tau & \text{(functions)} \\
 & | & (\tau, \tau) & \text{(products)} \\
 & | & \mathit{List}\ \tau & \text{(lists)} \\
 & | & m\ \tau & \text{(computations)}
\end{array}$$

When an operation is lifted through the monad transformer t, its new type can be derived by substituting all occurrences of m in the type with $t\ m$. Formally, $\lceil\cdot\rceil_t$ is the mapping of types across the monad transformer t:

$$\begin{array}{lcl}
\lceil A\rceil_t & = & A \\
\lceil a\rceil_t & = & a \\
\lceil \tau_1 \rightarrow \tau_2\rceil_t & = & \lceil \tau_1\rceil_t \rightarrow \lceil \tau_2\rceil_t \\
\lceil (\tau_1, \tau_2)\rceil_t & = & (\lceil \tau_1\rceil_t, \lceil \tau_2\rceil_t) \\
\lceil \mathit{List}\ \tau\rceil_t & = & \mathit{List}\ \lceil \tau\rceil_t \\
\lceil m\ \tau\rceil_t & = & t\ m\ \lceil \tau\rceil_t
\end{array}$$

3.3.2 Natural Lifting Condition

What properties should a particular lifting satisfy? Recall that in Section 2.4.3, we noted that the following was not a desirable lifting of *inEnv* through *ContT*:

$$\mathit{inEnv}_{\mathit{ContT}\ c\ m}\ r\ c \quad = \quad \lambda k.\mathit{inEnv}_m\ r\ (c\ k)$$

The problem is that the environment is not restored when c invokes k, which is equivalent to, for example, not popping off the arguments after a function returns. This lifting is not desirable because a new feature (a continuation) has disrupted an existing feature (the environment).

Intuitively, any programs not using the added feature should behave in the same way after a monad transformer is applied. The monad morphism property of *lift* ensures that single computations are properly lifted. But some operations, such as *callcc*, have more complex types—they take computations as arguments. Thus we extend Moggi's original definition and define **natural liftings** as a family of relations $\mathcal{L}_\tau$, indexed by type τ:

Definition 3.4
$\mathcal{L}_\tau$ is a **natural lifting** of operations of type τ along the monad transformer t if it satisfies:

$$\begin{array}{lcll}
\mathcal{L}_\tau & : & \tau \rightarrow \lceil \tau \rceil_t & \\
 & & & \\
\mathcal{L}_A & = & id & (1) \\
\mathcal{L}_a & = & id & (2) \\
\mathcal{L}_{\tau_1 \rightarrow \tau_2} & = & \lambda f.f' \textit{ satisfying:} & \\
 & & \quad \forall \mathcal{L}_{\tau_1}, \exists \mathcal{L}_{\tau_2}, \textit{such that: } f' \cdot \mathcal{L}_{\tau_1} = \mathcal{L}_{\tau_2} \cdot f & (3) \\
\mathcal{L}_{(\tau_1, \tau_2)} & = & \lambda(a,b).(\mathcal{L}_{\tau_1}\ a, \mathcal{L}_{\tau_2}\ b) & (4) \\
\mathcal{L}_{List\ \tau} & = & map_{List}\ \mathcal{L}_\tau & (5) \\
\mathcal{L}_{m\ \tau} & = & lift_t \cdot (map_m\ \mathcal{L}_\tau) & (6)
\end{array}$$

Despite the similarity between cases 5 and 6, case 5 is in fact more similar to case 4. Both cases 4 and 5 map τ across the some basic data type. In case 6, m is the monad on which the monad transformer t is applied.

Constant types (such as integer) and polymorphic types do not depend on any particular monad. (See cases 1 and 2.) On the other hand, we expect a lifted function, when applied to a value lifted from the domain of the original function, to return a lifting of the result of applying the original function to the unlifted value. This relationship is precisely captured by equation 3, which corresponds to the following commuting diagram:

$$\begin{tikzcd}
\tau_1 \arrow[r, "f"'] \arrow[d, "\mathcal{L}_{\tau_1}"'] & \tau_2 \arrow[d, "\mathcal{L}_{\tau_2}"] \\
\lceil \tau_1 \rceil_t \arrow[r, "f'"] & \lceil \tau_2 \rceil_t
\end{tikzcd}$$

The liftings of tuples and lists are straightforward. Finally, the *lift* operator that comes with the monad transformer m lifts computations in m. Note that $\mathcal{L}_\tau$ is mapped to the result of the computation, which may involve other computations.

The above does *not* provide a constructive definition for a type-parametric lifting function $\mathcal{L}$. The "satisfying" clause in the third equation specifies a constraint, rather than a definition of f'. That is why we define $\mathcal{L}$ as a relation rather than a function. In practice, we first find out by hand how to lift an operation through particular monad transformers, and then use the above equations to verify that such a lifting is indeed natural.

3.3.3 Verifying Natural Liftings

We now verify the natural lifting condition for the liftings in Section 2.4. The easy cases (*update*, *err* and *rdEnv*) are covered by the following theorem by Moggi (Moggi, 1990):

Proposition 3.5

If function f's domain does not involve any monadic type, then:

$$\mathit{lift}_t \cdot f$$

is a natural lifting of f through any monad transformer t.

Proof: Since the domain type (call it τ) does not involve the monad, the lifting of τ is τ itself. The above theorem follows from the commutativity of the following diagram:

$$\begin{tikzcd}
\tau \arrow[r, "f"'] \arrow[d, "id"'] & \tau' \arrow[d, "\mathit{lift}_t"] \\
\tau \arrow[r, "\mathit{lift}_t \cdot f"] & \lceil \tau' \rceil_t
\end{tikzcd}$$

□

We address the remaining cases (*merge*, *inEnv* and *callcc*) separately.

Proposition 3.6

$$\mathit{merge}_{(t_1 \ldots (t_n\ \mathit{List}) \ldots)} = \mathit{join}_{(t_1 \ldots (t_n\ \mathit{List}) \ldots)} \cdot \mathit{lift}_{t_1} \cdots \mathit{lift}_{t_n}$$

is a natural lifting of $\mathit{merge}_{\mathit{List}}$.

To prove that the lifting for *merge* is natural, we need the following property of *map* and *join*:

Lemma 3.1
If t is a monad transformer, m a monad, then:

$$\mathit{lift}_t \cdot \mathit{join}_m = \mathit{join}_{t\ m} \cdot \mathit{lift}_t \cdot (\mathit{map}_m\ \mathit{lift}_t)$$

Proof:

$$\begin{array}{rcll}
\mathit{lift}_t\ (\mathit{join}_m e) & = & \mathit{lift}_t\ \{a \leftarrow e;\ a\}_m & (\mathit{join}) \\
& = & \{a \leftarrow \mathit{lift}_t e;\ \mathit{lift}_t a\}_{t\ m} & \text{(monad morphism)} \\
& = & \{a \leftarrow \mathit{lift}_t e;\ b \leftarrow \mathit{return}_{t\ m}(\mathit{lift}_t a);\ b\}_{t\ m} & \text{(left unit)} \\
& = & \mathit{join}_{t\ m}\ \{a \leftarrow \mathit{lift}_t e;\ \mathit{return}_{t\ m}(\mathit{lift}_t a)\}_{t\ m} & (\mathit{join}) \\
& = & \mathit{join}_{t\ m}\ \{a \leftarrow \mathit{lift}_t e;\ \mathit{lift}_t(\mathit{return}_m(\mathit{lift}_t a))\}_{t\ m} & \text{(monad morphism)} \\
& = & \mathit{join}_{t\ m}\ (\mathit{lift}_t\ \{a \leftarrow e;\ \mathit{return}_m(\mathit{lift}_t a)\}_m) & \text{(monad morphism)} \\
& = & \mathit{join}_{t\ m}\ (\mathit{lift}_t\ (\mathit{map}_m\ \mathit{lift}_t\ e)) & (\mathit{map})
\end{array}$$

□

We can now prove Proposition 3.6 by verifying that the following diagram commutes:

$$\begin{tikzcd}
\mathit{List}\ (\mathit{List}\ a) \arrow[r, "\mathit{merge}_{\mathit{List}}"'] \arrow[d, "\mathit{map}_{\mathit{List}}\ (\mathit{lift}_{t_1} \cdots \mathit{lift}_{t_n})"'] & \mathit{List}\ a \arrow[d, "\mathit{lift}_{t_1} \cdots \mathit{lift}_{t_n}"] \\
\mathit{List}\ ((t_1 \ldots (t_n\ \mathit{List}) \ldots)\ a) \arrow[r, "\mathit{merge}_{(t_1 \ldots (t_n\ \mathit{List}) \ldots)}"] & (t_1 \ldots (t_n\ \mathit{List}) \ldots)\ a
\end{tikzcd}$$

Indeed we have:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{merge}_{(t_1\ldots(t_n\ \mathit{List})\ldots)} \cdot \mathit{map}_{\mathit{List}}\ (\mathit{lift}_{t_1} \cdots \mathit{lift}_{t_n})} \\ = & \mathit{merge}_{(t_1\cdots t_n)\ \mathit{List}} \cdot \mathit{map}_{\mathit{List}}\ \mathit{lift}_{t_1\cdots t_n} & (3.3) \\ = & \mathit{lift}_{t_1\cdots t_n} \cdot \mathit{merge}_{\mathit{List}} & (3.1) \\ = & \mathit{lift}_{t_1} \cdots \mathit{lift}_{t_n} \cdot \mathit{merge}_{\mathit{List}} & (3.3) \end{array}$$

□

Proposition 3.7

$$\begin{array}{lll} \mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\rho\ e & = & \lambda\rho'.\mathit{inEnv}_m\ \rho\ (e\ \rho') \\ \mathit{inEnv}_{\mathit{StateT}\ s\ m}\rho\ e & = & \lambda s.\mathit{inEnv}_m\ \rho\ (e\ s) \\ \mathit{inEnv}_{\mathit{ErrT}\ m}\rho\ e & = & \mathit{inEnv}_m\ \rho\ e \end{array}$$

are natural liftings of inEnv_m.

Proof: For $\mathit{inEnv}_{t\ m}$ to be a natural lifting, we need to prove that:

$$\mathit{inEnv}_{t\ m}\rho \cdot \mathit{lift}_t = \mathit{lift}_t \cdot \mathit{inEnv}_m\rho$$

Indeed we have:

$$\begin{array}{llll} \mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\rho\ (\mathit{lift}_{\mathit{EnvT}\ r'\ m}e) & = & \lambda\rho'.\mathit{inEnv}_m\rho(\mathit{lift}_{\mathit{EnvT}\ r'\ m}e\rho') & (\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}) \\ & = & \lambda\rho'.\mathit{inEnv}_m\rho e & (\mathit{lift}_{\mathit{EnvT}\ r'\ m}) \\ & = & \mathit{lift}_{\mathit{EnvT}\ r'\ m}(\mathit{inEnv}_m\rho e) & (\mathit{lift}_{\mathit{EnvT}\ r'\ m}) \end{array}$$

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{inEnv}_{\mathit{StateT}\ s\ m}\rho\ (\mathit{lift}_{\mathit{StateT}\ s\ m}e)} \\ = & \lambda s.\mathit{inEnv}_m\rho(\mathit{lift}_{\mathit{StateT}\ s\ m}es) & (\mathit{inEnv}_{\mathit{StateT}\ s\ m}) \\ = & \lambda s.\mathit{inEnv}_m\rho\{a \leftarrow e; \mathit{return}_m(s,a)\}_m & (\mathit{lift}_{\mathit{StateT}\ s\ m}) \\ = & \lambda s.\{a \leftarrow \mathit{inEnv}_m\rho e; \mathit{return}_m(s,a)\}_m & (\text{Prop. } 3.4) \\ = & \mathit{lift}_{\mathit{StateT}\ s\ m}(\mathit{inEnv}_m\rho e) & (\mathit{lift}_{\mathit{StateT}\ s\ m}) \end{array}$$

$$\begin{array}{llll} \mathit{inEnv}_{\mathit{ErrT}\ m}\rho\ (\mathit{lift}_{\mathit{ErrT}\ m}e) & = & \mathit{inEnv}_m\rho(\mathit{lift}_{\mathit{ErrT}\ m}e) & (\mathit{inEnv}_{\mathit{ErrT}\ m}) \\ & = & \mathit{inEnv}_m\rho\{a \leftarrow e; \mathit{return}_m(\mathit{Ok}\ a)\}_m & (\mathit{lift}_{\mathit{ErrT}\ m}, \mathit{map}_m) \\ & = & \{a \leftarrow \mathit{inEnv}_m\rho e; \mathit{return}_m(\mathit{Ok}\ a)\}_m & (\text{Prop. } 3.4) \\ & = & \mathit{lift}_{\mathit{ErrT}\ m}(\mathit{inEnv}_m\rho e) & (\mathit{lift}_{\mathit{ErrT}\ m}) \end{array}$$

□

Proposition 3.8

$$\begin{array}{llll} (a) & \mathit{callcc}_{\mathit{EnvT}\ r\ m} & = & \lambda\rho.\mathit{callcc}_m(\lambda k.f(\lambda a.\lambda\rho'.ka)\rho) \\ (b) & \mathit{callcc}_{\mathit{ErrT}\ m}\ f & = & \mathit{callcc}_m(\lambda k.f(\lambda a.k(\mathit{Ok}\ a))) \\ (c) & \mathit{callcc}_{\mathit{StateT}\ s\ m}\ f & = & \lambda s_0.\mathit{callcc}_m\ (\lambda k.f\ (\lambda a.\lambda s_1.k\ (s_0,a))\ s_0) \end{array}$$

are natural liftings of callcc_m.

Proof: To prove Proposition 3.8a, we apply Definition 3.4 to the type of *callcc*, and arrive at the following lemma:

Lemma 3.2

$$\mathit{callcc}_{t\ m} \text{ is a natural lifting of } \mathit{callcc}_m$$
$$\text{iff:}$$
$$\forall f, f'.(\forall k.f'(\mathit{lift}_t \cdot k) = \mathit{lift}_t(fk)) \quad \Rightarrow \quad \mathit{callcc}_{t\ m}f' = \mathit{lift}_t(\mathit{callcc}_m f)$$

Using Lemma 3.2, it is easy to show that $\mathit{callcc}_{\mathit{EnvT}\ r\ m}$ is a natural lifting of callcc_m:

$$\begin{array}{llll} \mathit{callcc}_{\mathit{EnvT}\ r\ m}f' & = & \lambda\rho.\mathit{callcc}_m(\lambda k.f'(\lambda a.\lambda\rho'.ka)\rho) & (\mathit{callcc}_{\mathit{EnvT}\ r\ m}) \\ & = & \lambda\rho.\mathit{callcc}_m(\lambda k.f'(\lambda a.\mathit{lift}_{\mathit{EnvT}\ r}(ka))\rho) & (\mathit{lift}_{\mathit{EnvT}\ r}) \\ & = & \lambda\rho.\mathit{callcc}_m(\lambda k.\mathit{lift}_{\mathit{EnvT}\ r}(fk)\rho) & (\text{prerequisite of } 3.2) \\ & = & \lambda\rho.\mathit{callcc}_m(\lambda k.fk) & (\mathit{lift}_{\mathit{EnvT}\ r}) \\ & = & \mathit{lift}_{\mathit{EnvT}\ r}(\mathit{callcc}_m f) & (\mathit{lift}_{\mathit{EnvT}\ r}) \end{array}$$

Paterson (Paterson, 1995) showed a simple proof for the naturalness of $callcc_{ErrT\ m}$ using the free theorem (Wadler, 1989) for *callcc*:

$$\begin{aligned}
&\forall g, h, f, f'.\\
&\quad (\forall k, k'.k' \cdot g = map\ h \cdot k \Rightarrow f'k' = map\ g\ (fk)) \Rightarrow\\
&\qquad callcc f' = map\ g\ (callcc f)
\end{aligned}$$

By specializing f' to $\lambda k.f''(k \cdot g)$, we can transform the free theorem to:

Lemma 3.3

$$\begin{aligned}
&\forall g, h, f, f''.\\
&\quad (\forall k, f''(map\ h \cdot k) = map\ g\ (fk)) \Rightarrow\\
&\qquad callcc(\lambda k.f''(k \cdot g)) = map\ g\ (callcc f)
\end{aligned}$$

We now use Lemma 3.3 to prove $callcc_{ErrT\ m}$ is a natural lifting. Letting:

$$g = h = Ok$$

we have:

$$\begin{aligned}
callcc_{ErrT\ m} f'' &= callcc_m(\lambda k.f(k \cdot Ok)) && (callcc_{ErrT\ m})\\
&= map_m Ok\ (callcc_m\ f) && (\text{free theorem and prerequisite in 3.2})\\
&= lift_{ErrT}\ (callcc_m\ f) && (lift_{ErrT})
\end{aligned}$$

Thus $callcc_{ErrT\ m}$ is a natural lifting, following Lemma 3.2.

The free theorem, however, is not powerful enough to prove the naturalness of $callcc_{StateT\ s\ m}$. Instead, we introduce the following lemma, which is a slight variation of the free theorem:

Lemma 3.4

$$\begin{aligned}
&\forall g, h, f, f', s_0.\\
&\quad (\forall k, f'(\lambda x.\lambda s.map\ (\lambda x.h(s,x))\ (kx))s_0 = map\ g\ (fk) \Rightarrow\\
&\qquad callcc\ (\lambda k.f'(\lambda x.\lambda s.k(gx))s_0) = map\ g\ (callcc\ f)
\end{aligned}$$

The proof of the lemma is in Appendix A. We will apply Lemma 3.4 with the following specialized definitions to prove $callcc_{StateT\ s\ m}$ is a natural lifting:

$$\begin{aligned}
g &= \lambda x.(s_0, x)\\
h &= \lambda x.x
\end{aligned}$$

The proof is carried out in two steps. First, we verify the prerequisite of Lemma 3.4, using the prerequisite of Lemma 3.2.

$$\begin{aligned}
&f'(\lambda x.\lambda s.map_m\ (\lambda x.h(s,x))\ (kx))s_0\\
&\quad = f'(\lambda x.\lambda s.map_m\ (\lambda x.(s,x))\ (kx))s_0 && (h)\\
&\quad = f'(\lambda x.lift_{StateT\ s}\ (kx))s_0 && (lift_{StateT\ s})\\
&\quad = lift_{StateT\ s}\ (fk)s_0 && (\text{prerequisite of 3.2})\\
&\quad = map_m\ g\ (fk) && (lift_{StateT\ s})
\end{aligned}$$

Second, we use the result of Lemma 3.4 to establish the the sufficient and necessary condition in Lemma 3.2:

$$\begin{aligned}
callcc_{StateT\ s\ m}\ f'\ s_0 &= callcc_m(\lambda k.f'(\lambda a.\lambda s_1.k(s_0,a))s_0) && (callcc_{StateT\ s\ m})\\
&= map_m(\lambda a.(s_0,a))\ (callcc_m\ f) && (3.4)\\
&= lift_{StateT\ s}\ (callcc_m\ f) && (lift_{StateT\ s})
\end{aligned}$$

Apply the above to Lemma 3.2, we have proved that $callcc_{StateT\ s\ m}$ is a natural lifting.
□

So far we have established that *all* the liftings in Figure 4 are natural. Note that the following lifting of $callcc_{StateT\ s\ m}$:

$$callcc_{StateT\ s\ m}\ f \quad = \quad \lambda s_0.callcc_m\ (\lambda k.f\ (\lambda a.\lambda s_1.k\ (s_1, a))\ s_0)$$

which passes the current state to the continuation, is *not* natural. Here is a counter-example discovered by Paterson (Paterson, 1995). Let:

$$f'k = lift_{StateT\ s}(f(\lambda x.bind(kxs_1)(\lambda(s', x).\ return\, x)))$$

For any state s_1, f' and f meet the condition:

$$\forall k.f'(lift_{StateT\ s} \cdot k) = lift_{StateT\ s}(fk))$$

However:

$$\begin{array}{lcl} callcc_{StateT\ s\ (ContT\ c\ Id)}f'\ s_0\ k & = & f(\lambda x.\lambda k'.k(s_1, x))(\lambda x.k(s_0, x)) \\ lift_{StateT\ s}(callcc_{ContT\ c\ Id}f)\ s_0\ k & = & f(\lambda x.\lambda k'.k(s_0, x))(\lambda x.k(s_0, x)) \end{array}$$

are different.

3.4 Ordering of Monad Transformers

The ordering of monad transformers has an impact on the resulting semantics. For example, we have seen that lifting *callcc* through *StateT* results in a "debugging" semantics. On the other hand, if we apply *ContT* to a state monad, then we get the usual semantics for *callcc*. To demonstrate, we construct two monads:

$$\begin{array}{lcl} \textbf{type}\ M1\ a & = & ContT\ c\ (StateT\ Int\ Id)\ a \\ \textbf{type}\ M2\ a & = & StateT\ Int\ (ContT\ c\ Id)\ a \end{array}$$

The program segment:

$$callcc(\lambda k.\{\_ \leftarrow update(\lambda x.x + 1); k0\})$$

expands to:

$$\lambda k.\lambda s_0.k\ 0\ (s_0 + 1)$$

in $M1$, but to:

$$\lambda s_0.\lambda k.k\ (s_0, 0)$$

in $M2$.

The key difference is that one combination captures the state in the continuation, whereas the other combination does not.

In general we can swap the ordering of some monad transformers (such as between *StateT* and *EnvT*), but doing so to others (such as *ContT*) may effect semantics. This is consistent with earlier experience in combining monads (King & Wadler, 1993), and, in practice, provides us with an opportunity to fine tune the resulting semantics.

4 Modular Monadic Interpreters

We can transform a denotational semantics description into an executable interpreter by translating the mathematical notations into corresponding programming constructs. Modern functional languages such as Haskell (Hudak *et al.*, 1992) or SML (Milner *et al.*, 1990) are particularly suitable because these languages offer features such as algebraic data types and higher-order functions that match well with the mathematical notations used in denotational semantics.

While the static type system in Haskell or SML is capable of implementing traditional

```
type Term = OR TermA       -- arithmetic
          ( OR TermF       -- functions
          ( OR TermR       -- assignment
          ( OR TermL       -- lazy evaluation
          ( OR TermT       -- tracing
          ( OR TermC       -- callcc
               TermN       -- nondeterminism
             )))))

type M = EnvT Env          -- environment
       ( ContT Answer      -- continuations
       ( StateT Store      -- memory cells
       ( StateT String     -- trace output
       ( ErrT              -- error reporting
         List              -- multiple results
         ))))

type Value = OR Int        -- integers
           ( OR Loc        -- memory locations
           ( OR Fun        -- functions
                ()))
```

Fig. 5. Gofer specification of a modular interpreter

denotational semantics, implementing monadic modular semantics in a strongly typed language has proved to be a challenge. For example, Steele (Steele Jr., 1994) reported numerous difficulties when he built a modular monadic interpreter in Haskell. Although the Haskell type system can implement individual monads and monad transformers as type constructors, modular monadic semantics requires the type system to capture relationships among different monads and monad transformers.

We have successfully implemented a modular monadic interpreter in *Gofer* (Jones, 1991), whose *constructor classes* and *multi-parameter type classes* provide just the added power over Haskell's type classes[¶] to allow precise and convenient expression of the typing relationships. Figure 5 gives the high-level definition of the interpreter for our source language. The rest of the section will explain how the type declarations expand into a full interpreter. For now just note that `OR` is equivalent to the domain sum operator, and that `Term`, `Value` and `M` denote the abstract syntax, runtime values, and the interpreter monad, respectively.

4.1 Extensible Union Types

We begin with a discussion of a key idea in our implementation: how values and terms may be expressed as *extensible union types*. This facility has nothing to do with monads.

The disjoint union of two types is implemented by the data type `OR`:

```
data OR a b = L a
            | R b
```

[¶] The newly defined Haskell 1.3 (Peterson & Hammond, 1996) supports constructor classes (but not multi-parameter type classes).

where `L` and `R` are used to perform the conventional injection of a summand type into the union; conventional pattern-matching is used for projection. However, such injections and projections only work if we know the exact structure of the union. When building modular interpreters, an extensible union may be arbitrarily nested or extended. We would like a *single* pair of injection and projection functions to work on all such constructions.

To achieve this, we define a multi-parameter type class to implement the summand/union type relationship, which we refer to as a “subtype” relationship:

```
class SubType sub sup where
  inj :: sub -> sup                     -- injection
  prj :: sup -> Maybe sub               -- projection

data Maybe a = Just a
             | Nothing
```

The `Maybe` data type is used because the projection function may fail. We can now express the relationships between the summand and union types:

```
instance SubType a (OR a b) where
  inj       = L
  prj (L x) = Just x
  prj _     = Nothing

instance SubType a b => SubType a (OR c b) where
  inj       = R . inj
  prj (R a) = prj a
  prj _     = Nothing
```

It would appear that we could have a more symmetric instance declaration in place of the second declaration above:

```
instance SubType a (OR b a) where
  inj       = R
  prj (R x) = Just x
  prj _     = Nothing
```

With this declaration, however, the Gofer type system complains that `(OR a a)` is an overlapping instance. The type system cannot determine which of the two injection/projection pairs are applicable if the programmer supplies, for example, `(OR Int Int)` as the union type.

Now we can see how the `Value` domain used in Figure 5, for example, is actually constructed:

```
type Value = OR Int (OR Loc (OR Fun ()))
type Fun   = M Value -> M Value
```

With these definitions the Gofer type system will infer that `Int`, `Loc`, and `Fun` are all “subtypes” of `Value`, and the coercion functions `inj` and `prj` will be generated automatically.[‖]

[‖] Most of the typing problems Steele (Steele Jr., 1994) encountered disappear with the use of our extensible union types; in particular, there is no need for Steele’s “towers” of data types.

4.2 Interpreter Building Blocks

As seen in Figure 5, the `Term` type is also constructed as an extensible union (of subterm types). We define additionally a class `InterpC` to characterize the term types that we wish to interpret:

```
class InterpC t where
  interp :: t -> M Value
```

The behavior of the evaluation function `interp` on unions of terms is given in the obvious way:

```
instance (InterpC t1, InterpC t2) =>
                  InterpC (OR t1 t2) where
  interp (L t) = interp t
  interp (R t) = interp t
```

The interpreter is just the method associated with the top-level type `Term`:

```
interp :: Term -> M Value
```

The interpreter building blocks are straightforward translations of the semantic building blocks in Section 2.1 into instance declarations. For example, the arithmetic building block can be implemented as follows:

```
data TermA = Num Int
          | Add Term Term

instance InterpC TermA where
  interp (Num x)   = returnInj x
  interp (Add x y) = interp x `bindPrj` \i ->
                     interp y `bindPrj` \j ->
                     returnInj ((i + j) :: Int)

returnInj     = return . inj
m `bindPrj` k = m `bind` \a ->
               case (prj a) of
                 Just x  -> k x
                 Nothing -> err "type error"
```

Note the simple use of `inj` and `prj` to inject/project the integer result into/out of the `Value` domain, regardless of how `Value` is eventually defined (`returnInj` and `bindPrj` make this a tad easier). The `err` function is the error reporting function implemented by the underlying monad.

Appendix B lists Gofer implementation of other interpreter building blocks. They can be similarly translated from the corresponding monadic semantics.

Before discussing how to implement the monad transformers needed to construct the interpreter monad `M`, we introduce Gofer's constructor classes through a motivating example.

4.3 Constructor Classes

Constructor classes (Jones, 1993) support abstraction of common features among type constructors. Haskell, for example, provides the standard `map` function to apply a function to each element of a given list:

```
map :: (a -> b) -> List a -> List b
```

Meanwhile, we can define similar functions for a wide range of other data types. For example:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)

mapTree :: (a -> b) -> Tree a -> Tree b

mapTree f (Leaf x)   = Leaf (f x)
mapTree f (Node l r) = Node (mapTree f l) (mapTree f r)
```

The `mapTree` function has similar type and functionality to those of `map`. With this in mind, it seems a shame that we have to use different names for each of these variants. Indeed, Gofer allows type variables to stand for *type constructors*, on which the Haskell type class system has been extended to support overloading. To solve the problem with `map`, we can introduce a new constructor class `Functor` (in a categorical sense):

```
class Functor f where
  map :: (a -> b) -> f a -> f b
```

Now the standard list (`List`) and the user-defined type constructor `Tree` can be defined as instances of `Functor`:

```
instance Functor List where
  map f [] = []
  map f (x:xs) = f x : map f xs

instance Functor Tree where
  map f (Leaf x)   = Leaf (f x)
  map f (Node l r) = Node (map f l) (map f r)
```

Constructor classes are extremely useful for dealing with multiple instances of monads and monad transformers (which are all type constructors).

4.4 Monads

We follow a well known approach (Jones, 1993) to define monads using a constructor class:

```
class Monad m where
  return :: a -> m a
  bind :: m a -> (a -> m b) -> m b

  map  :: (a -> b) -> (m a -> m b)
  join :: m (m a) -> m a

  map f m = m `bind` \a -> return (f a)
  join m  = m `bind` id
```

`Map` and `join` are conveniently defined as default methods in terms of `bind` and `return`. A specific monad, such as `List`, is an instance of the `Monad` class:

```
instance Monad List where
   return x        = [x]
```

```
    [] `bind` k     = []
    (x:xs) `bind` k = k x ++ (xs `bind` k)
```

The interesting properties of a monad are the additional operations it supports. We can further define subclasses of `Monad`, each containing an additional set of operations. For example, `ListMonad` has one extra operation `merge`:

```
  class Monad m => ListMonad m where
    merge :: List (m a) -> m a
```

The standard list monad `List` implements `merge` as follows:

```
 instance ListMonad List where
 --merge :: List (List a) -> List a
   merge []     = []
   merge (x:xs) = x ++ (merge xs)
```

Other classes of monads, such as `StateMonad`, `EnvMonad`, `ContMonad` and `ErrMonad`, can be similarly defined. (See Appendix C for details.)

4.5 Monad Transformers

We implement monad transformers in the following constructor class definition:

```
 class MonadT t where
   lift :: (Monad m, Monad (t m)) => m a -> t m a
```

To illustrate how individual instances are defined, we use the state monad transformer (*StateT*) as an example. The Gofer implementation of *EnvT*, *ContT*, and *ErrT* can be found in Appendix C.

From Section 2.3 we know that applying monad transformer `StateT s` to monad `m` results in a monad `StateT s m`. Because Gofer only allows us to partially apply a data type, not a type synonym, we introduce a dummy data constructor and define `StateT` as an algebraic data type**:

```
data StateT s m a = StateM (s -> m (s,a))
unStateM (StateM x) = x

instance Monad m => Monad (StateT s m) where
    return x            = StateM (\s -> return (s,x))
    (StateM m) `bind` k =
                 StateM (\s0 -> m s0 `bind` \(s1, a) ->
                              unStateM (k a) s1)
```

The definition follows exactly from Figure 3, except for dealing with the `StateM` data constructor. Note that `bind` and `return` are not recursive functions; the constructor class system automatically infers that the functions appearing on the right are for monad `m`.

Next, we define `StateT s` as a monad transformer:

```
instance MonadT (StateT s) where
 -- lift  :: m a -> StateT s m a
    lift m = StateM (\s -> m `bind` \x -> return (s,x))
```

** Haskell 1.3(Peterson & Hammond, 1996) introduces a `newtype` construct that can be used to avoid the run-time penalty of dummy data constructors such as `StateM`.

We introduce `StateMonad` as a subclass of `Monad` with an additional operation `update`:

```
class Monad m => StateMonad s m where
    update :: (s -> s) -> m s
```

Monad transformer `StateT s` adds the `update` function on `s` to any monad `m`:

```
instance Monad m => StateMonad s (StateT s m) where
    update f = StateM (\s -> return (f s, s))
```

Finally, we can lift `update` through any monad transformer by composing it with `lift` (see Proposition 3.5):

```
instance (StateMonad s m, MonadT t) =>
                            StateMonad s (t m) where
    update = lift . update
```

As another example of lifting, we can apply any monad transformer to `List` and obtain a `ListMonad` (see Proposition 3.6):

```
 instance (MonadT t, Monad m) => ListMonad (t m) where
   merge = join . lift
```

4.6 Summary

We have shown that modular interpreter building blocks and monad transformers can be implemented using two key features in Gofer type system: *constructor classes* and *multi-parameter type classes*. Our approach offers several benefits. First, it allows us to experiment with and debug our ideas. Second, the overloading mechanism greatly facilitates representing multiple instances of monads and monad transformers, eliminating the need for subscripts. Third, type checking guarantees that we have enough features in the underlying monad to support the set of building blocks needed for our source language. For example, if we had instead constructed the monad `M` in figure 5 without the `StateT String` monad transformer:

```
type M = EnvT Env        -- environment
       ( ContT Answer    -- continuations
       ( StateT Store    -- memory cells
                         -- missing state component for IO
       ( ErrT            -- error reporting
         List            -- multiple results
         )))
```

then the Gofer type system would complain that `StateMonad String M` cannot be inferred from the definition of `M`.

5 Compilation

In this section we investigate how to compile the source language from its monadic semantics specification. The target language we consider is fairly high-level, providing support for closures, tagged data structures, basic control-flow (such as conditionals) and garbage collection. How to implement a back-end that efficiently supports such target languages has been investigated by a number of compiler research efforts (e.g., the techniques developed for T (Kranz *et al.*, 1986), SML/NJ (Appel, 1992), and Haskell (Peyton Jones, 1992)).

Even though we do not tackle the problem of building compiler back-ends, our work provides insights into how we may build a *common back-end* capable of supporting a variety of source languages. Writing separate back-ends for different source languages leads to duplication of efforts. On the other hand, a common back-end has the following benefits:

- It simplifies the task of constructing compilers.
- It allows multiple source languages to interoperate by freely exchanging compatible runtime data.

Modular monadic semantics fits well with a common back-end, because it is suitable for specifying multiple source languages, and, as will be seen, it leads to an efficient and provably correct compilation scheme. This is achieved in several steps. First, we require that our semantics 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 inductive proofs on programs possible. In practice, this guarantees that, given any abstract syntax tree, we can recursively unfold all calls to the interpreter, effectively removing runtime dispatch on the abstract syntax tree.

Our second step is to simplify the resulting monadic style code composed out of various monadic operations (such as *bind* and *inEnv*). As will be seen in Section 5.1, monad laws are useful in simplifying code; and environment axioms can be used to eliminate the costly interpretive overhead of environment lookups. In Section 5.2, we formally prove that all environment lookups can be removed.

The final step (Section 5.3) is to map monadic-style intermediate code to the target language. The main focus is on how to utilize the built-in target language features.

5.1 Using Monad Laws to Transform Programs

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

```
$E[\![((\lambda v.v+1)\ 2)_v]\!] =$
  $\{\ f \leftarrow \{\ \rho \leftarrow rdEnv;$
              $return\ (\lambda x.inEnv\ \rho[x/[\![v]\!]]\ \{\ i \leftarrow \{\ \rho \leftarrow rdEnv;$
                                                  $\rho[\![v]\!]\ \};$
                                          $j \leftarrow return\, 1;$
                                          $return\ (i+j)\ \})\ \};$
     $v \leftarrow return\, 2;$
     $f(return\ \ v)\ \}$
```

Even without any further simplifications, the above code is clear enough to describe the computation. By applying monad laws we can simplify it to:

```
$\{\ \rho \leftarrow rdEnv;$
   $(\lambda x.inEnv\ \rho[x/[\![v]\!]]\ \{\ \rho \leftarrow rdEnv;$
                              $i \leftarrow \rho[\![v]\!];$
                              $return\ (i+1)\ \})\ (return\ \ 2)\ \}$
```

By applying the distribution, unit and cancellation environment axioms, followed by the unit monad law, we can further transform the example code to:

```
$\{\ \rho \leftarrow rdEnv;$
   $(\lambda x.\{i \leftarrow inEnv\ \rho[x/[\![v]\!]]\ x; return\ (i+1)\})\ (return\ \ 2)\ \}$
```

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

Note that the true computation in the original expression "$((\lambda x.x + 1)\ 2)_v$" is left unreduced. With traditional denotational semantics, it is harder to distinguish the redexes introduced by the compilation process from computations in the source program. In the above example, we could safely further reduce the intermediate code:

$$\begin{array}{lll} & (\lambda x.\{i \leftarrow x; \mathit{return}\ (i+1)\})(\mathit{return}\ 2) & \\ \Rightarrow & \{i \leftarrow \mathit{return}\ 2; \mathit{return}\ (v+1)\} & (\beta) \\ \Rightarrow & \mathit{return}\ 3 & \text{(left unit)} \end{array}$$

However, in general, unrestricted reductions for arbitrary source programs could result in unwanted compile-time exceptions, such as in "$((\lambda x.10/x)\ 0)_v$."

5.2 A Natural Semantics

We successfully transformed away the explicit environment in the above example, but can we do the same for arbitrary source programs? If that is possible, we will have an effective compilation scheme that uses the target language environment for the source language, without any interpretive overhead.

It turns out that we can indeed prove such a general result by using monad laws and environment axioms. Following Wand (Wand, 1990), we define a "natural semantics" that translates source language variables to lexical variables in the meta-language, and we prove that it is equivalent to the standard semantics.

5.2.1 Definition of a Natural Semantics

We adopt Wand's definition of a *natural semantics* (which differs from Kahn's notion (Clément *et al.*, 1986)) to our functional sub-language. For any source language variable name v, we assume there is a corresponding variable name $\overline{v}$ in the meta-language, and $\overline{\rho}$ is an environment that maps variable name v to $\overline{v}$.

Definition 5.1
The *natural semantics* for the source language is defined as follows:

$$\begin{array}{lcl} N[\![v]\!] & = & \overline{v} \\ N[\![\lambda v.e]\!] & = & \text{return}(\lambda\overline{v}.\text{inEnv}\ \overline{\rho}\ N[\![e]\!]) \\ N[\![(e_1\ e_2)_n]\!] & = & \{f \leftarrow N[\![e_1]\!]; f(\text{inEnv}\ \overline{\rho}\ N[\![e_2]\!])\} \\ N[\![(e_1\ e_2)_v]\!] & = & \{f \leftarrow N[\![e_1]\!]; v \leftarrow N[\![e_2]\!]; f(\text{return}\, v)\} \\ N[\![(e_1\ e_2)_l]\!] & = & \{\ f \leftarrow N[\![e_1]\!]; \\ & & \quad l \leftarrow \text{alloc}; \\ & & \quad \textbf{let}\ \text{thunk} = \{\ v \leftarrow \text{inEnv}\ \overline{\rho}\ N[\![e_2]\!]; \\ & & \qquad\qquad\qquad\quad \_ \leftarrow \text{write}\ (l, \text{return}\ \ v); \\ & & \qquad\qquad\qquad\quad \text{return}\ \ v\ \} \\ & & \quad \textbf{in}\ \{\ \_ \leftarrow \text{write}\ (l, \text{thunk}); \\ & & \qquad\quad f\ (\text{read}\ l)\ \}\ \} \end{array}$$

Other source-level constructs, such as +, :=, and callcc, do not explicitly deal with the environment, and have the same natural semantics as the standard semantics.

The natural semantics uses the environment of the meta-language for variables in the source language.

5.2.2 Correspondence between Natural and Standard Semantics

The next theorem, a variation of Wand's (Wand, 1990), states that the standard semantics and natural semantics are equivalent, and thus guarantees that it is safe to implement function calls in the source language using the meta-language environment.

Theorem 5.1
For any source language program e, we have:

$$inEnv\ \overline{\rho}\ E[\![e]\!] = inEnv\ \overline{\rho}\ N[\![e]\!]$$

The detailed proof is in Appendix A. The basic technique is equational reasoning based on the rules of lambda calculus (e.g., β reduction), monad laws, and environment axioms. We establish the theorem for each semantic building block, independent of:

- the existence of other building blocks, and
- the organization of the underlying monad.

Therefore the result holds for each building block as long as the underlying monad provides the necessary kernel-level support so that the monad laws and environment axioms hold. The proof can be reused, even after other features are added into the source language.

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

$$E[\![\lambda v.e]\!] \quad = \quad return(\lambda c.\{\rho \leftarrow rdEnv; inEnv\ \rho[c/[\![v]\!]]\ E[\![e]\!]\}),$$

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

5.2.3 Benefits of Reasoning in Monadic Style

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. For example, Wand (Wand, 1990) pointed out that he could change to a continuation-based semantics, and prove the theorem, but only by modifying the proofs accordingly.

Modular monadic semantics, on the other hand, offers highly modularized proofs and more general results. This is particularly applicable to real programming languages, which usually carry a large set of features and undergo evolving designs.

5.3 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 stores and continuations.

5.3.1 The Target Language Monad

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

Monadic operations	ML constructs
$\mathit{return}\, x$	x
$\{x \leftarrow c_1; c_2\}$	**let val** $x = c_1$ **in** c_2 **end**
update	**ref**, **!**, **:=**, print
callcc	callcc
err	**raise Err**

Note that the imperative features in ML (e.g., **:=** and print) supports a single-threaded store, whereas the monadic *update* operation more generally supports recoverable store. 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):

$$\boxed{\begin{array}{ll}\textbf{let val } v_2 = & \textbf{let val } v_1 = c_1 \\ & \textbf{in } c_2 \textbf{ end} \\ \textbf{in } c_3 \textbf{ end} & \end{array}} = \boxed{\begin{array}{l}\textbf{let val } v_1 = c_1 \\ \textbf{in } \ \textbf{let val } v_2 = c_2 \\ \qquad \textbf{in } c_3 \textbf{ end end}\end{array}}$$

5.3.2 Utilizing Target Language Features

We now investigate how to utilize the features directly supported by the target language monad. Because of a technical limitation related to nondeterminism, we tentatively drop it from our source language. (We will discuss the support for nondeterminism later.) The underlying monad M becomes:

$$\textbf{type } M\ a = \mathit{EnvT}\ \mathit{Env}\ (\mathit{ContT}\ \mathit{Answer}\ (\mathit{StateT}\ \mathit{Store}\ (\mathit{StateT}\ \mathit{IO}\ (\mathit{ErrT}\ \mathit{Id}))))\ a$$

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

$$\textbf{type } M_1\ a = \mathit{EnvT}\ \mathit{Env}\ (\mathit{ContT}\ \mathit{Answer}\ (\mathit{StateT}\ \mathit{Store}\ (\mathit{StateT}\ \mathit{IO}\ (\mathit{ErrT}\ M_{ML}))))\ a$$

Note that M_1 supports two sets of kernel-level operations for continuation, store, I/O, and error reporting. The monadic code can choose to use the ML built-in operations instead of those implemented as higher-order functions. In addition, if we have used the natural semantics to transform away all environment accesses, then the *EnvT* monad transformer is no longer useful. Because the natural lifting condition guarantees that adding or deleting an unused monad transformer does not effect the result of the computation, it suffices to run the target program on M_2:

$$\textbf{type } M_2\ a = M_{ML}\ a$$

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

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 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.

We do not need to transform away all monad transformers. For example, the following monad is also capable of supporting the source language:

type $M_3\ a = \mathit{ContT}\ \mathit{Answer}\ M_{ML}\ a$

Because M_3 supports two *callcc* operations, the monadic code can either use the ML built-in *callcc* function, or use the *callcc* supported by the continuation monad transformer.

5.3.3 Limitations of This Approach

It is important to recognize the limitations of the transformation process:

1. Unlike other features, nondeterminism must be directly supported by the target language, since the nondeterminism monad (*List*) must be the base monad. This is why we put aside nondeterminism in the preceding discussion.
2. We have shown that the ordering of monad transformers (in particular, the cases involving *ContT*) has an impact on the resulting semantics. In practice, we need to make sure when we use one monad transformer instead of another, that the resulting change of ordering does not have unwanted effects on semantics. For example, if we had left one of the state monad transformers unreduced:

   type $M_4\ a = \mathit{StateT}\ \mathit{Store}\ M_{ML}\ a$

   we have effectively swapped the order of *StateT* and *ContT*. (The latter is now supported in M_{ML}.)

5.3.4 Implications for a Common Back-end

To overcome the above limitations, a common back-end must support a rich set of features needed by a wide range of source languages, thereby guaranteeing that we can always transform away the monad transformers.

The ordering of monad transformers only effects the semantics of *callcc*. To deal with situations where the order of monad transformers matters, the back-end can provide multiple variations of the a monadic operation, with each version implementing a variation of the semantics. For example, a back-end can support a special version of *callcc* that captures the current state for the purpose of debugging.

6 Related Work, Future Work and Conclusion

In this paper, we have demonstrated how monads and monad transformers can be used to provide more modular specification of programming language features than traditional denotational semantics. In addition, we have shown how the modularity offered in our framework can provide better support for equational reasoning, program transformation, interpreter construction, and semantics-directed compilation. More specifically, the contributions of the work presented in this paper are as follows:

- We have constructed modular semantic building blocks that support a wide variety of source language features, including arithmetic expressions, call-by-value, call-by-name, lazy evaluation, references and assignment, tracing, first-class continuation, and nondeterminism. Although each of these features has been modeled using monads before, it is the first time all of them fit into a single modular framework.
- We have solved a number of open problems in how to lift operations through monad transformers. We have extended Moggi's (Moggi, 1990) natural lifting condition to higher-order types, making it possible to reason about the relatively complex operations related to environment and continuation. In addition, we have shown how liftings capture the interactions between various programming language features.

- We have implementationed modular semantic building blocks and monad transformers in Gofer (Jones, 1991). This is the first implementation of a full-featured modular monadic interpreter using a strongly-typed language.
- We have investigated high-level monadic properties of programming language features (for example, the environment axioms). We have applied these properties to construct modular proofs and to perform semantics-directed compilation.

6.1 Related Work

Our work is built on a number of previous attempts to better organize modular semantics, to more effectively reason about programming languages, and to more efficiently compile higher-order programs.

6.1.1 Modular Semantics

The lack of modularity of traditional denotational semantics (Stoy, 1977) has long been recognized (Mosses, 1984) (Lee, 1989).

Moggi first suggested the use of *monads* and *monad transformers* to structure denotational semantics. Wadler popularized Moggi's ideas in the functional programming community by showing how monads could be used in a variety of settings, including incorporating imperative features (Peyton Jones & Wadler, 1993) and building modular interpreters (Wadler, 1992). Wadler (King & Wadler, 1993) also discussed the issues in combining monads. *Pseudomonads* (Steele Jr., 1994) were proposed as a way to compose monads and thus build up an interpreter from smaller parts. However, implementing pseudomonads in the Haskell (Hudak *et al.*, 1992) type system turned out to be problematic.

Returning to Moggi's original ideas, Espinosa formulated a system called *Semantic Lego* (Espinosa, 1993) (Espinosa, 1995). Espinosa's Scheme-based system was the first modular interpreter that incorporated monad transformers. Among his contributions, Espinosa pointed out that pseudomonads were really just a special kind of *monad transformer*, first suggested by Moggi as a way to leave a "hole" in a monad for further extension. Espinosa's work reminded the programming language community—who had become distracted by the use of monads—that Moggi himself, responsible in many ways for the interest in monadic programming, had actually focussed more on the importance of monad transformers.

Related approaches to enhance modularity include composing monads (Jones & Duponcheel, 1993) and stratified monads (Espinosa, 1994).

This paper was motivated by the above line of work, which led to the solution (Liang *et al.*, 1995) of a number of open issues in how to lift operations through monad transformers, as well as how to implement modular interpreters in a strongly-typed language.

6.1.2 Reasoning with Monads

In his original note (Moggi, 1990), Moggi raised the issue of reasoning in the monadic framework. The monadic framework has been used to specify state monad laws (Wadler, 1990), and to reason about exceptions (Spivey, 1990). A related, but more general, framework to reason about states is mutable abstract data types (MADTs) (Chen & Hudak, 1997).

This paper extends previous work by presenting the environment axioms (Liang & Hudak, 1996). In addition, we demonstrate how these axioms, together with monad laws, can be used to reason about programs in a modular way.

6.1.3 Semantics-directed Compilation

Early efforts (Wand, 1984) (Paulson, 1982) were based on traditional denotational semantics. The resulting compilers were inefficient.

Action Semantics (Mosses, 1992) allows modular specification of programming language semantics. Action semantics and a related approach (Lee, 1989) have been successfully used to generate efficient compilers. While action semantics is easy to construct, extend, understand and implement, we note the following comments ((Mosses, 1992), 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."

Our work 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 approach.

A related approach (Meijer, 1995) is to combine the standard initial algebra semantics approach with aspects of Action Semantics to derive compilers from denotational semantics.

One application of partial evaluation (Jones *et al.*, 1989) is to generate compilers from interpreters. A partial evaluator has been successfully applied to an action interpreter (Bondorf & Palsberg, 1993), and similar results can be achieved with monadic interpreters (Danvy *et al.*, 1991) as well.

Staging transformations (Jørring & Scherlis, 1986) are a class of general program transformation techniques for separating a given computation into stages. Monad transformers make computational stages somewhat more explicit by separating compile-time features, such as the environment, from run-time features.

There have been several successful efforts (including (Kelsey & Hudak, 1989), (Appel & Jim, 1989), and others) to build 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 & Felleisen, 1992) and (Hatcliff & Danvy, 1994)).

6.2 Future Work

6.2.1 Theory of Programming Language Features

We have used monads and monad transformers to study programming language features and their interactions. Plenty of work remains on extending the theory to handle other useful features we have not covered. As a result, we may be able to better understand and implement these features.

6.2.2 Monadic Program Transformation

We have demonstrated that monadic code is particularly suitable for program transformation. Because monadic semantics is no more than an abstraction of traditional denotational semantics, all equational reasoning techniques apply. Monadic semantics can thus be used to facilitate various program transformation techniques such as partial evaluation.

6.2.3 A Common Back-end for Modern Languages

The experience of building a retargeted Haskell compiler suggests the feasibility of a common back-end for modern languages. An efficient, well-thought-out system such as SML/NJ is a strong candidate to serve as a common back-end for a variety of modern languages.

We can further develop our monadic semantics based compilation method into a compiler construction tool for a common back-end.

6.2.4 Concurrency

Concurrency is an important feature in many modern languages such as Java$^{\mathrm{TM}}$ (Gosling *et al.*, 1996). The monadic framework covers the properties of *callcc*. Since *callcc* captures the activities occur during a thread context switch, we expect the results related to *callcc* will be useful in reasoning about multi-threaded concurrent systems.

6.3 Conclusion

We have demonstrated the power of modular monadic semantics in two ways. First, it is a powerful technique to specify and reason about programming language features. Second, it can be used in practice to construct modular interpreters and perform semantics-directed compilation.

The key benefit of our approach is modularity. The underlying mechanism is monad-based abstraction. Modular monadic semantics helps to bridge the gap between programming language theory and the complex practical languages.

References

Appel, Andrew W. (1992). *Compiling with continuations.* Cambridge University Press.

Appel, Andrew W., & Jim, Trevor. 1989 (Jan.). Continuation-passing, closure-passing style. *Pages 193–302 of: Acm symposium on principles of programming languages.*

Bondorf, Anders, & Palsberg, Jens. (1993). Compiling actions by partial evaluation. *Pages 308–317 of: FPCA '93: Conference on functional programming languages and computer architecture, Copenhagen, Denmark.* New York: ACM Press.

Chen, Chih-Ping, & Hudak, Paul. (1997). Rolling your own mutable ADT—a connection between linear types and monads. *Pages 54–66 of: Proceedings of 24th acm symposium on principles of programming languages.* New York: ACM Press.

Clément, D., Despeyroux, Joëlle, Despeyroux, Thierry, & Kahn, Gilles. (1986). A simple applicative language: Mini-ML. *Pages 13–27 of: Proceedings of the 1986 ACM symposium on Lisp and functional programming.*

Clinger, William, & Rees, Jonathan. 1991 (November). *Revised^4 report on the algorithmic language Scheme.* Available via World Wide Web as http://www-swiss.ai.mit.edu/ftpdir/scheme-reports/r4rs.ps.

Danvy, Olivier, Koslowski, Jürgen, & Malmkjær, Karoline. 1991 (Dec.). *Compiling monads.* Technical Report CIS-92-3. Kansas State University.

Espinosa, David. 1993 (December). *Modular denotational semantics.* Unpublished manuscript.

Espinosa, David. 1994 (June). *Building interpreters by transforming stratified monads.* Unpublished manuscript, ftp from altdorf.ai.mit.edu:pub/dae.

Espinosa, David. (1995). *Semantic lego.* Ph.D. thesis, Columbia University.

Felleisen, Matthias, & Hieb, Robert. (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, **103**, 235–271.

Felleisen, Matthias, Friedman, Daniel P., Kohlbecker, Eugene, & Duba, Bruce. 1986 (June). Reasoning with continuations. *Pages 131–141 of: Proceedings of the symposium on logic in computer science*. IEEE, Cambridge, Massachusetts.

Gosling, James, Joy, Bill, & Steele, Guy. (1996). *The* Java$^{\text{tm}}$ *programming language*. The Java Series. Addison-Wesley.

Hatcliff, John, & Danvy, Olivier. (1994). A generic account of continuation-passing styles. *Pages 458–471 of: 21st ACM symposium on principles of programming languages (POPL '94), Portland, Oregon*. New York: ACM Press.

Hudak, Paul, & Bloss, Adriene. 1985 (January). The aggregate update problem in functional programming languages. *Pages 300–314 of: Conference record of the twelfth ACM symposium on principles of programming languages*. Association for Computing Machinery.

Hudak, Paul, Peyton Jones, Simon, & Wadler, Philip. 1992 (March). *Report on the programming language Haskell: a non-strict, purely functional language, version 1.2*. Tech. rept. YALEU/DCS/RR-777. Yale University Department of Computer Science. Also in ACM SIGPLAN Notices, Vol. 27(5), May 1992.

Jones, Mark P. 1991 (September). *Introduction to gofer 2.20*. Ftp from nebula.cs.yale.edu in the directory pub/haskell/gofer.

Jones, Mark P. (1993). A system of constructor classes: Overloading and implicit higher-order polymorphism. *Pages 52–61 of: FPCA '93: Conference on functional programming languages and computer architecture, Copenhagen, Denmark*. New York: ACM Press.

Jones, Mark P., & Duponcheel, Luc. 1993 (December). *Composing monads*. Research Report YALEU/DCS/RR-1004. Yale University Department of Computer Science, New Haven, Connecticut.

Jones, Neil D., Sestoft, Peter, & Søndergaard, Harald. (1989). Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and symbolic computation*, **2**, 9–50.

Jørring, Ulrik, & Scherlis, William. (1986). Compilers and staging transformations. *Pages 86–96 of: Proceedings thirteenth ACM symposium on principles of programming languages, St. Petersburg, Florida*.

Kelsey, Richard, & Hudak, Paul. 1989 (Jan.). Realistic compilation by program transformation. *Pages 181–192 of: Acm symposium on principles of programming languages*.

King, David J., & Wadler, Philip. (1993). Combining monads. *Pages 134–143 of:* Launchbury, John, & Sansom, Patrick (eds), *Functional programming, Glasgow 1992*. New York: Springer-Verlag.

Kishon, Amir, Hudak, Paul, & Consel, Charles. 1991 (June). Monitoring semantics: A formal framework for specifying, implementing and reasoning about execution monitors. *Pages 338–352 of: Proceedings of the ACM SIGPLAN '91 conference on programming language design and implementation*.

Kranz, David, Kelsey, Richard, Rees, Jonathan, Hudak, Paul, Philbin, James, & Adams, Norman. 1986 (July). ORBIT: An optimizing compiler for Scheme. *Pages 219–233 of: Proceedings SIGPLAN '86 symposium on compiler construction*. SIGPLAN Notices Volume 21, Number 7.

Lee, Peter. (1989). *Realistic compiler generation*. Foundations of Computing. MIT Press.

Liang, Sheng, & Hudak, Paul. (1996). Modular denotational semantics for compiler construction. *Pages 219–234 of:* Nielson, Hanne Riis (ed), *ESOP '96: 6th European symposium on programming, Linkoping, sweden, proceedings*. New York: Springer-Verlag. Lecture Notes in Computer Science 1058.

Liang, Sheng, Hudak, Paul, & Jones, Mark. (1995). Monad transformers and modular interpreters. *22nd ACM symposium on principles of programming languages (POPL '95), San Francisco, California*. New York: ACM Press.

Mac Lane, Saunders. (1971). *Categories for the working mathematician*. Graduate Texts in Mathematics. Springer-Verlag.

Meijer, Erik. (1995). *More advice on proving a compiler correct: Improving a correct compiler*. Submitted to Journal of Functional Programming.

Milner, Robin, Tofte, Mads, & Harper, Robert. (1990). *The definition of standard ML*. MIT Press.

Moggi, Eugenio. (1990). *An abstract view of programming languages*. Technical Report ECS-LFCS-90-113. Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland.

Mosses, Peter D. (1984). A basic abstract semantic algebra. *Pages 87–107 of:* Kahn, Gilles, MacQueen, David B., & Plotkin, Gordon D. (eds), *Semantics of data types: International symposium, Sophia-Antipolis, france*. Springer-Verlag. Lecture Notes in Computer Science 173.

Mosses, Peter D. (1992). *Action semantics*. Cambridge Tracts in Theoretical Computer Science, vol. 26. Cambridge University Press.

Paterson, Ross A. (1995). private communication.

Paulson, Laurence C. (1982). A semantics-directed compiler generator. *Pages 224–233 of: Proceedings of the ninth ACM symposium on principles of programming languages, albuquerque, new mexico*.

Peterson, John, & Hammond, Kevin. 1996 (May). *Report on the programming language Haskell: a non-strict, purely functional language, version 1.3*. Tech. rept. YALEU/DCS/RR-1106. Yale University Department of Computer Science.

Peyton Jones, Simon, & Wadler, Philip. 1993 (Jan.). Imperative functional programming. *Pages 71–84 of: Proceedings 20th symposium on principles of programming languages*. ACM.

Peyton Jones, S.L. (1992). Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of functional programming*, **2**(2), 127–202.

Sabry, Amr, & Felleisen, Matthias. (1992). Reasoning about programs in continuation-passing style. *Pages 288–298 of: Proceedings of the 1992 ACM conference on LISP and functional programming*. ACM Press.

Spivey, Michael. (1990). A functional theory of exceptions. *Science of computer programming*, **14**(1), 25–42.

Steele Jr., Guy L. (1994). Building interpreters by composing monads. *Pages 472–492 of: Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT symposium on principles of programming languages, Portland, Oregon*. New York: ACM Press.

Stoy, Joseph. (1977). *Denotational semantics: The scott-strachey approach to programming language theory*. MIT Press.

Tolmach, Andrew P., & Appel, Andrew W. 1990 (June). Debugging standard ML without reverse engineering. *Proceedings of the 1990 ACM conference on Lisp and functional programming*.

Wadler, Philip. 1992 (January). The essence of functional programming. *Pages 1–14 of: Conference record of the nineteenth annual ACM symposium on principles of programming languages, Albuquerque, New Mexico*.

Wadler, Philip L. 1989 (September). Theorems for free! *Fourth symposium on functional programming languages and computer architecture*. ACM. London.

Wadler, Philip L. (1990). Comprehending monads. *Proceedings of the 1990 ACM confer-*

ence on Lisp and functional programming.

Wand, Mitchell. (1984). A semantic prototyping system. *Sigplan notices, acm symposium on compiler construction*, **19**(6), 213–221.

Wand, Mitchell. (1990). A short proof of the lexical addressing algorithm. *Information processing letters*, **35**(June), 1–5.

A Proofs

This appendix contains detailed proofs for many of the results given in the body of this paper. For convenience, we repeat the statement of each result at the beginning of the corresponding proof.

Proposition 3.2 *EnvT r*, *StateT s*, *ErrT*, and *ContT c* are monad transformers.

Proof: We need to show that the corresponding *lift* functions are monad morphisms.

Case *EnvT r*:

$$\begin{array}{lcll}
\mathit{return}_{\mathit{EnvT}\ r\ m}\, e & = & \lambda\rho.\, \mathit{return}_m\, e & (\mathit{return}_{\mathit{EnvT}\ r\ m}) \\
 & = & \mathit{lift}_{\mathit{EnvT}\ r}\ (\mathit{return}_m\, e) & (\mathit{lift}_{\mathit{EnvT}\ r})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{bind}_{\mathit{EnvT}\ r\ m}\ (\mathit{lift}_{\mathit{EnvT}\ r} m)\ (\lambda a.(\mathit{lift}_{\mathit{EnvT}\ r}(ka)))} \\
\quad = & \mathit{bind}_{\mathit{EnvT}\ r\ m}\ (\lambda\rho'.m)\ (\lambda a.(\lambda\rho'.ka)) & (\mathit{lift}_{\mathit{EnvT}\ r}) \\
\quad = & \lambda\rho.\{a \leftarrow (\lambda\rho'.m)\ \rho; (\lambda\rho'.ka)\rho\}_m & (\mathit{bind}_{\mathit{EnvT}\ r\ m}) \\
\quad = & \lambda\rho.\mathit{bind}_m\ e\ k & (\beta) \\
\quad = & \mathit{lift}_{\mathit{EnvT}\ r}\ (\mathit{bind}_m\ e\ k) & (\mathit{lift}_{\mathit{EnvT}\ r})
\end{array}$$

Case *StateT s*:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{return}_{\mathit{StateT}\ s\ m}\, e} \\
\quad = & \lambda s.\, \mathit{return}_m(s, e) & (\mathit{return}_{\mathit{StateT}\ s\ m}) \\
\quad = & \lambda s.\{a \leftarrow \mathit{return}_m\, e; \mathit{return}_m(s, a)\}_m & \text{(left unit)} \\
\quad = & \mathit{lift}_{\mathit{StateT}\ s}\ (\mathit{return}_m\, e) & (\mathit{lift}_{\mathit{StateT}\ s})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{bind}_{\mathit{StateT}\ s\ m}\ (\mathit{lift}_{\mathit{StateT}\ s} e)\ (\lambda a.\mathit{lift}_{\mathit{StateT}\ s}(ka))} \\
\quad = & \lambda s.\{(s', a) \leftarrow \mathit{lift}_{\mathit{StateT}\ s}\ e\ s; \mathit{lift}_{\mathit{StateT}\ s}\ (ka)\ s'\}_m & (\mathit{bind}_{\mathit{StateT}\ s\ m}) \\
\quad = & \lambda s.\{(s', a') \leftarrow \{a \leftarrow e; \mathit{return}_m(s, a)\}_m; & \\
 & \qquad b \leftarrow ka'; \mathit{return}_m(s', b)\}_m & (\mathit{lift}_{\mathit{StateT}\ s}) \\
\quad = & \lambda s.\{a \leftarrow e; (s', a') \leftarrow \mathit{return}_m(s, a); & \\
 & \qquad b \leftarrow ka'; \mathit{return}_m(s', b)\}_m & \text{(associativity)} \\
\quad = & \lambda s.\{a \leftarrow e; b \leftarrow ka; \mathit{return}_m(s, b)\}_m & \text{(left unit)} \\
\quad = & \mathit{lift}_{\mathit{StateT}\ s}\ (\mathit{bind}_m\ e\ k) & (\mathit{lift}_{\mathit{StateT}\ s})
\end{array}$$

Case *ErrT*:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{return}_{\mathit{ErrT}\ m}\, e} \\
\quad = & \mathit{return}_m(\mathit{Ok}\ e) & (\mathit{return}_{\mathit{ErrT}\ m}) \\
\quad = & \{a \leftarrow \mathit{return}_m\, e; \mathit{return}_m(\mathit{Ok}\ a)\}_m & \text{(left unit)} \\
\quad = & \mathit{map}_m\ \mathit{Ok}\ (\mathit{return}_m\, e) & (\mathit{map}_m) \\
\quad = & \mathit{lift}_{\mathit{ErrT}}\ (\mathit{return}_m\, e) & (\mathit{lift}_{\mathit{ErrT}})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{bind}_{\mathit{ErrT}\ m}\ (\mathit{lift}_{\mathit{ErrT}}e)\ (\lambda a.\mathit{lift}_{\mathit{ErrT}}(ka))} \\
\quad = & \{a' \leftarrow \mathit{lift}_{\mathit{ErrT}}e; & \\
 & \quad \textbf{case}\ a'\ \textbf{of}\ \mathit{Ok}\ a \rightarrow \mathit{lift}_{\mathit{ErrT}}(ka)\ldots\}_m & (\mathit{bind}_{\mathit{ErrT}\ m}) \\
\quad = & \{a' \leftarrow \{a \leftarrow e; \mathit{return}_m(\mathit{Ok}\ a)\}_m; & \\
 & \quad \textbf{case}\ a'\ \textbf{of}\ \mathit{Ok}\ a \rightarrow \{b \leftarrow ka; \mathit{return}_m(\mathit{Ok}\ b)\}_m\ldots\}_m & (\mathit{lift}_{\mathit{ErrT}},\ \mathit{map}_m) \\
\quad = & \{a \leftarrow e; a' \leftarrow \mathit{return}_m(\mathit{Ok}\ a); & \\
 & \quad \textbf{case}\ a'\ \textbf{of}\ \mathit{Ok}\ a \rightarrow \{b \leftarrow ka; \mathit{return}_m(\mathit{Ok}\ b)\}_m\ldots\}_m & (\text{associativity}) \\
\quad = & \{a \leftarrow e; b \leftarrow ka; \mathit{return}_m(\mathit{Ok}\ b)\}_m & (\text{left unit}) \\
\quad = & \mathit{lift}_{\mathit{ErrT}}\ (\mathit{bind}_m\ e\ k) & (\mathit{lift}_{\mathit{ErrT}},\ \mathit{map}_m)
\end{array}$$

Case *ContT*:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{return}_{\mathit{ContT}\ c\ m}\ e} \\
\quad = & \lambda k.ke & (\mathit{return}_{\mathit{ContT}\ c\ m}) \\
\quad = & \mathit{bind}_m\ (\mathit{return}_m\ e) & (\text{left unit}) \\
\quad = & \mathit{lift}_{\mathit{ContT}\ c}\ (\mathit{return}_m\ e) & (\mathit{lift}_{\mathit{ContT}\ c})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{bind}_{\mathit{ContT}\ c\ m}\ (\mathit{lift}_{\mathit{ContT}\ c}e)\ (\lambda a.\mathit{lift}_{\mathit{ContT}\ c}(fa))} \\
\quad = & \lambda k.(\mathit{lift}_{\mathit{ContT}\ c}e)\ (\lambda a.\mathit{lift}_{\mathit{ContT}\ c}\ (fa)\ k) & (\mathit{bind}_{\mathit{ContT}\ c\ m}) \\
\quad = & \lambda k.\mathit{bind}_m\ e\ (\lambda a.\mathit{bind}_m\ (fa)\ k) & (\mathit{lift}_{\mathit{ContT}\ c}) \\
\quad = & \lambda k.\mathit{bind}_m\ (\mathit{bind}_m\ e\ f)\ k & (\text{associativity}) \\
\quad = & \mathit{lift}_{\mathit{ContT}\ c}\ (\mathit{bind}_m\ e\ f) & (\mathit{lift}_{\mathit{ContT}\ c})
\end{array}$$

Proposition 3.4 The environment operations, *rdEnv* and *inEnv* satisfy the following axioms:

$$\begin{array}{rcll}
(\mathit{inEnv}\ \rho) \cdot \mathit{return} & = & \mathit{return} & (\text{unit}) \\
\mathit{inEnv}\ \rho\ \{v \leftarrow e_1; e_2\} & = & \{v \leftarrow \mathit{inEnv}\ \rho\ e_1; \mathit{inEnv}\ \rho\ e_2\} & (\text{distribution}) \\
\mathit{inEnv}\ \rho\ \mathit{rdEnv} & = & \mathit{return}\,\rho & (\text{cancellation}) \\
\mathit{inEnv}\ \rho'\ (\mathit{inEnv}\ \rho\ e) & = & \mathit{inEnv}\ \rho\ e & (\text{overriding})
\end{array}$$

Proof: We verify that: 1) *inEnv* and *rdEnv* satisfy the axioms after being introduced by *EnvT*, and that: 2) the axioms are preserved through *EnvT*, *StateT*, and *ErrT*. (There is no lifting of *inEnv* through *ContT*.)
Base case:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho\ (\mathit{return}_{\mathit{EnvT}\ r\ m}\ x)} \\
\quad = & \lambda\rho'.(\mathit{return}_{\mathit{EnvT}\ r\ m}\ x)\rho & (\mathit{inEnv}_{\mathit{EnvT}\ r\ m}) \\
\quad = & \lambda\rho'.(\lambda\rho''.\ \mathit{return}_m\ x)\rho & (\mathit{return}_{\mathit{EnvT}\ r\ m}) \\
\quad = & \lambda\rho'.\ \mathit{return}_m\ x & (\beta) \\
\quad = & \mathit{return}_{\mathit{EnvT}\ r\ m}\ x & (\mathit{return}_{\mathit{EnvT}\ r\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho\ \{v \leftarrow e_1; e_2\}_{\mathit{EnvT}\ r\ m}} \\
\quad = & \lambda\rho'.\{v \leftarrow e_1; e_2\}_{\mathit{EnvT}\ r\ m}\rho & (\mathit{inEnv}_{\mathit{EnvT}\ r\ m}) \\
\quad = & \lambda\rho'.\{v \leftarrow e_1\ \rho; e_2\ \rho\}_m & (\mathit{bind}_{\mathit{EnvT}\ r\ m}) \\
\quad = & \lambda\rho'.\{v \leftarrow (\lambda\rho''.e_1\ \rho)\rho'; (\lambda\rho''.e_2\ \rho)\rho'\}_m & \\
\quad = & \lambda\rho'.\{v \leftarrow \mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho\ e_1\ \rho'; \mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho\ e_2\ \rho'\}_m & (\mathit{inEnv}_{\mathit{EnvT}\ r\ m}) \\
\quad = & \{v \leftarrow \mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho\ e_1; \mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho\ e_2\}_{\mathit{EnvT}\ r\ m} & (\mathit{bind}_{\mathit{EnvT}\ r\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho\ \mathit{rdEnv}_{\mathit{EnvT}\ r\ m}} \\
\quad = & \lambda\rho'.\mathit{rdEnv}_{\mathit{EnvT}\ r\ m}\ \rho & (\mathit{inEnv}_{\mathit{EnvT}\ r\ m}) \\
\quad = & \lambda\rho'.(\lambda\rho.\ \mathit{return}_m\ \rho)\rho & (\mathit{rdEnv}_{\mathit{EnvT}\ r\ m}) \\
\quad = & \lambda\rho'.\ \mathit{return}_m\ \rho & \\
\quad = & \mathit{return}_{\mathit{EnvT}\ r\ m}\ \rho & (\mathit{return}_{\mathit{EnvT}\ r\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho'\ (\mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho\ e)} \\
 = & \lambda\rho''.\mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho\ e\ \rho' & (\mathit{inEnv}_{\mathit{EnvT}\ r\ m}) \\
 = & \lambda\rho''.(\lambda\rho'.e\ \rho)\ \rho' & (\mathit{inEnv}_{\mathit{EnvT}\ r\ m}) \\
 = & \lambda\rho''.e\ \rho & (\beta) \\
 = & \mathit{inEnv}_{\mathit{EnvT}\ r\ m}\ \rho\ e & (\mathit{inEnv}_{\mathit{EnvT}\ r\ m})
\end{array}$$

Case *EnvT* r':

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho\ (\mathit{return}_{\mathit{EnvT}\ r'\ m}\ x)} \\
 = & \lambda\rho'.\mathit{inEnv}_m\ \rho\ (\mathit{return}_{\mathit{EnvT}\ r'\ m}\ x\rho') & (\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}) \\
 = & \lambda\rho'.\mathit{inEnv}_m\ \rho\ ((\lambda\rho''.\,\mathit{return}_m\ x)\rho') & (\mathit{return}_{\mathit{EnvT}\ r'\ m}) \\
 = & \lambda\rho'.\mathit{inEnv}_m\ \rho\ (\mathit{return}_m\ x) & (\beta) \\
 = & \lambda\rho'.\,\mathit{return}_m\ x & (\text{ind. hypo.}) \\
 = & \mathit{return}_{\mathit{EnvT}\ r'\ m}\ x & (\mathit{return}_{\mathit{EnvT}\ r'\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho\ \{v \leftarrow e_1; e_2\}_{\mathit{EnvT}\ r'\ m}} \\
 = & \lambda\rho'.\mathit{inEnv}_m\ \rho\ (\{v \leftarrow e_1; e_2\}_{\mathit{EnvT}\ r'\ m}\rho') & (\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}) \\
 = & \lambda\rho'.\mathit{inEnv}_m\ \rho\ \{v \leftarrow e_1\ \rho'; e_2\ \rho'\}_m & (\mathit{bind}_{\mathit{EnvT}\ r'\ m}) \\
 = & \lambda\rho'.\{v \leftarrow \mathit{inEnv}_m\ \rho\ (e_1\ \rho'); \mathit{inEnv}_m\ \rho\ (e_2\ \rho')\}_m & (\text{ind. hypo.}) \\
 = & \lambda\rho'.\{v \leftarrow \mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho\ e_1\ \rho'; \mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho\ e_2\ \rho'\}_m & (\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}) \\
 = & \{v \leftarrow \mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho\ e_1; \mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho\ e_2\}_{\mathit{EnvT}\ r'\ m} & (\mathit{bind}_{\mathit{EnvT}\ r'\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho\ \mathit{rdEnv}_{\mathit{EnvT}\ r'\ m}} \\
 = & \lambda\rho'.\mathit{inEnv}_m\ \rho\ (\mathit{rdEnv}_{\mathit{EnvT}\ r'\ m}\ \rho') & (\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}) \\
 = & \lambda\rho'.\mathit{inEnv}_m\ \rho\ (\lambda\rho''.\mathit{rdEnv}_m\ \rho') & (\mathit{rdEnv}_{\mathit{EnvT}\ r'\ m}) \\
 = & \lambda\rho'.\mathit{inEnv}_m\ \rho\ \mathit{rdEnv}_m & (\beta) \\
 = & \lambda\rho'.\,\mathit{return}_m\ \rho & (\text{ind. hypo.}) \\
 = & \mathit{return}_{\mathit{EnvT}\ r'\ m}\ \rho & (\mathit{return}_{\mathit{EnvT}\ r'\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho'\ (\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho\ e)} \\
 = & \lambda\rho''.\mathit{inEnv}_m\ \rho'\ (\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho\ e\ \rho'') & (\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}) \\
 = & \lambda\rho''.\mathit{inEnv}_m\ \rho'\ ((\lambda\rho'.\mathit{inEnv}_m\ \rho\ (e\rho'))\ \rho'') & (\mathit{inEnv}_{\mathit{EnvT}\ r'\ m}) \\
 = & \lambda\rho''.\mathit{inEnv}_m\ \rho'\ (\mathit{inEnv}_m\ \rho\ (e\rho'')) & (\beta) \\
 = & \lambda\rho''.\mathit{inEnv}_m\ \rho\ (e\rho'') & (\text{ind. hypo.}) \\
 = & \mathit{inEnv}_{\mathit{EnvT}\ r'\ m}\ \rho\ e & (\mathit{inEnv}_{\mathit{EnvT}\ r'\ m})
\end{array}$$

Case *StateT* s:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho\ (\mathit{return}_{\mathit{StateT}\ s\ m}\ x)} \\
 = & \lambda s.\mathit{inEnv}_m\ \rho\ (\mathit{return}_{\mathit{StateT}\ s\ m}\ xs) & (\mathit{inEnv}_{\mathit{StateT}\ s\ m}) \\
 = & \lambda s.\mathit{inEnv}_m\ \rho\ ((\lambda s.\,\mathit{return}_m(s,x))s) & (\mathit{return}_{\mathit{StateT}\ s\ m}) \\
 = & \lambda s.\mathit{inEnv}_m\ \rho\ (\mathit{return}_m(s,x)) & (\beta) \\
 = & \lambda s.\,\mathit{return}_m(s,x) & (\text{ind. hypo.}) \\
 = & \mathit{return}_{\mathit{StateT}\ s\ m}\ x & (\mathit{return}_{\mathit{StateT}\ s\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho\ \{v \leftarrow e_1; e_2\}_{\mathit{StateT}\ s\ m}} \\
 = & \lambda s.\mathit{inEnv}_m\ \rho\ (\{v \leftarrow e_1; e_2\}_{\mathit{StateT}\ s\ m}s) & (\mathit{inEnv}_{\mathit{StateT}\ s\ m}) \\
 = & \lambda s.\mathit{inEnv}_m\ \rho\ \{(s',v) \leftarrow e_1\ s; e_2 s'\}_m & (\mathit{bind}_{\mathit{StateT}\ s\ m}) \\
 = & \lambda s.\{(s',v) \leftarrow \mathit{inEnv}_m\ \rho\ (e_1\ s); \mathit{inEnv}_m\ \rho\ (e_2 s')\}_m & (\text{ind. hypo.}) \\
 = & \lambda s.\{(s',v) \leftarrow \mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho\ e_1\ s; \mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho\ e_2\ s'\}_m & (\mathit{inEnv}_{\mathit{StateT}\ s\ m}) \\
 = & \{v \leftarrow \mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho\ e_1; \mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho\ e_2\}_{\mathit{StateT}\ s\ m} & (\mathit{bind}_{\mathit{StateT}\ s\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho\ \mathit{rdEnv}_{\mathit{StateT}\ s\ m}} & \\
= & \lambda s.\mathit{inEnv}_m\ \rho\ (\mathit{rdEnv}_{\mathit{StateT}\ s\ m}\ s) & (\mathit{inEnv}_{\mathit{StateT}\ s\ m}) \\
= & \lambda s.\mathit{inEnv}_m\ \rho\ \{\rho' \leftarrow \mathit{rdEnv}_m; \mathit{return}_m(s,\rho')\}_m & (\mathit{rdEnv}_{\mathit{StateT}\ s\ m}) \\
= & \lambda s.\{\rho' \leftarrow \mathit{inEnv}_m\ \rho\ \mathit{rdEnv}_m; \mathit{inEnv}_m\ \rho\ \ \mathit{return}_m(s,\rho')\}_m & (\text{ind. hypo.}) \\
= & \lambda s.\{\rho' \leftarrow \mathit{return}_m\ \rho; \mathit{return}_m(s,\rho')\}_m & (\text{ind. hypo.}) \\
= & \lambda s.\ \mathit{return}_m(s,\rho) & (\text{left unit}) \\
= & \mathit{return}_{\mathit{StateT}\ s\ m}\ \rho & (\mathit{return}_{\mathit{StateT}\ s\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho'\ (\mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho\ e)} & \\
= & \lambda s.\mathit{inEnv}_m\ \rho'\ (\mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho\ e\ s) & (\mathit{inEnv}_{\mathit{StateT}\ s\ m}) \\
= & \lambda s.\mathit{inEnv}_m\ \rho'\ ((\lambda s'.\mathit{inEnv}_m\ \rho\ (es'))\ s) & (\mathit{inEnv}_{\mathit{StateT}\ s\ m}) \\
= & \lambda s.\mathit{inEnv}_m\ \rho'\ (\mathit{inEnv}_m\ \rho\ (es)) & (\beta) \\
= & \lambda s.\mathit{inEnv}_m\ \rho\ (es) & (\text{ind. hypo.}) \\
= & \mathit{inEnv}_{\mathit{StateT}\ s\ m}\ \rho\ e & (\mathit{inEnv}_{\mathit{StateT}\ s\ m})
\end{array}$$

Case *ErrT*:

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}_{\mathit{ErrT}\ m}\ \rho\ (\mathit{return}_{\mathit{ErrT}\ m}\ x)} & \\
= & \mathit{inEnv}_m\ \rho\ (\mathit{return}_{\mathit{ErrT}\ m}\ x) & (\mathit{inEnv}_{\mathit{ErrT}\ m}) \\
= & \mathit{inEnv}_m\ \rho\ (\mathit{return}_m(\mathit{Ok}\ x)) & (\mathit{return}_{\mathit{ErrT}\ m}) \\
= & \mathit{return}_m(\mathit{Ok}\ x) & (\text{ind. hypo.}) \\
= & \mathit{return}_{\mathit{ErrT}\ m}\ x & (\mathit{return}_{\mathit{ErrT}\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}_{\mathit{ErrT}\ m}\ \rho\ \{v \leftarrow e_1; e_2\}_{\mathit{ErrT}\ m}} & \\
= & \mathit{inEnv}_m\ \rho\ (\{v \leftarrow e_1; e_2\}_{\mathit{ErrT}\ m} s) & (\mathit{inEnv}_{\mathit{ErrT}\ m}) \\
= & \mathit{inEnv}_m\ \rho\ \{\ a \leftarrow e_1; & (\mathit{bind}_{\mathit{ErrT}\ m}) \\
 & \qquad\qquad \textbf{case}\ a\ \textbf{of} & \\
 & \qquad\qquad\quad \mathit{Ok}\ v \rightarrow e_2 & \\
 & \qquad\qquad\quad \mathit{Err}\ s \rightarrow \mathit{return}_m(\mathit{Err}\ s)\}_m & \\
= & \{\ a \leftarrow \mathit{inEnv}_m \rho e_1; & (\text{ind. hypo.}) \\
 & \quad \textbf{case}\ a\ \textbf{of} & \\
 & \qquad \mathit{Ok}\ v \rightarrow \mathit{inEnv}_m \rho e_2 & \\
 & \qquad \mathit{Err}\ s \rightarrow \mathit{inEnv}_m \rho\ \mathit{return}_m(\mathit{Err}\ s)\}_m & \\
= & \{\ a \leftarrow \mathit{inEnv}_m \rho e_1; & (\text{ind. hypo.}) \\
 & \quad \textbf{case}\ a\ \textbf{of} & \\
 & \qquad \mathit{Ok}\ v \rightarrow \mathit{inEnv}_m \rho e_2 & \\
 & \qquad \mathit{Err}\ s \rightarrow \mathit{return}_m(\mathit{Err}\ s)\}_m & \\
= & \{v \leftarrow \mathit{inEnv}_{\mathit{ErrT}\ m}\ \rho\ e_1; \mathit{inEnv}_{\mathit{ErrT}\ m}\ \rho\ e_2\}_{\mathit{ErrT}\ m} & (\mathit{bind}_{\mathit{ErrT}\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}_{\mathit{ErrT}\ m}\ \rho\ \mathit{rdEnv}_{\mathit{ErrT}\ m}} & \\
= & \mathit{inEnv}_m\ \rho\ \mathit{rdEnv}_{\mathit{ErrT}\ m} & (\mathit{inEnv}_{\mathit{ErrT}\ m}) \\
= & \mathit{inEnv}_m\ \rho\ \{\rho' \leftarrow \mathit{rdEnv}_m; \mathit{return}_m(\mathit{Ok}\ \rho')\}_m & (\mathit{rdEnv}_{\mathit{ErrT}\ m}) \\
= & \{\rho' \leftarrow \mathit{inEnv}_m\ \rho\ \mathit{rdEnv}_m; \mathit{inEnv}_m\ \rho\ \ \mathit{return}_m(\mathit{Ok}\ \rho')\}_m & (\text{ind. hypo.}) \\
= & \{\rho' \leftarrow \mathit{return}_m\ \rho; \mathit{return}_m(\mathit{Ok}\ \rho')\}_m & (\text{ind. hypo.}) \\
= & \mathit{return}_m(\mathit{Ok}\ \rho) & (\text{left unit}) \\
= & \mathit{return}_{\mathit{ErrT}\ m}\ \rho & (\mathit{return}_{\mathit{ErrT}\ m})
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}_{\mathit{ErrT}\ m}\ \rho'\ (\mathit{inEnv}_{\mathit{ErrT}\ m}\ \rho\ e)} & \\
= & \mathit{inEnv}_m\ \rho'\ (\mathit{inEnv}_m\ \rho\ e) & (\mathit{inEnv}_{\mathit{ErrT}\ m}) \\
= & \mathit{inEnv}_m\ \rho\ e & (\text{ind. hypo.}) \\
= & \mathit{inEnv}_{\mathit{ErrT}\ m}\ \rho\ e & (\mathit{inEnv}_{\mathit{ErrT}\ m})
\end{array}$$

Lemma 3.4

$$
\begin{array}{l}
\forall g, h, f, f', s_0 .\\
\quad (\forall k, f'(\lambda x.\lambda s.\mathit{map}\ (\lambda x.h(s,x))\ (kx))s_0 = \mathit{map}\ g\ (fk) \Rightarrow\\
\qquad \mathit{callcc}\ (\lambda k.f'(\lambda x.\lambda s.k(gx))s_0) = \mathit{map}\ g\ (\mathit{callcc}\ f)
\end{array}
$$

Proof: We establish the lemma by covering the cases when *callcc* was first introduced by *ContT* and lifted through *EnvT*, *StateT*, and *ErrT*. (There is no lifting of *callcc* through *ContT*.)

Base case:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{callcc}(\lambda k.f'(\lambda x.\lambda s.k(gx))s_0)k}\\
= & (\lambda k.f'(\lambda x.\lambda s.k(gx))s_0)(\lambda a.\lambda k'.ka)k\\
= & f'(\lambda x.\lambda s.\lambda k'.k(gx))s_0 k\\
= & f'(\lambda x.\lambda s.\lambda k'.(\lambda k''.k(gx))(\lambda x.k'(h(s,x))))s_0 k\\
= & f'(\lambda x.\lambda s.\mathit{map}_{\mathit{ContT}\ c\ m}(\lambda x.h(s,x))\ (\lambda k''.k(gx)))s_0 k\\
= & \mathit{map}_{\mathit{ContT}\ c\ m}\ g\ (f(\lambda x.\lambda k''.k(gx)))\ k \qquad\qquad \text{(pre-condition)}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{map}_{\mathit{ContT}\ c\ m}\ g\ (\mathit{callcc} f)\ k & = & \mathit{callcc}\ f\ (\lambda x.k(gx))\\
& = & (\lambda k.f(\lambda a.\lambda k'.ka)k)(\lambda x.k(gx))\\
& = & f(\lambda a.\lambda k'.k(ga))(\lambda x.k(gx))\\
& = & \mathit{map}_{\mathit{ContT}\ c\ m}\ g\ (f(\lambda a.\lambda k'.k(ga)))k
\end{array}
$$

Case "$t = \mathit{EnvT}\ r$:"
Let:

$$
\begin{array}{lcl}
\underline{f}'\ k\ s_0 & = & f'(\lambda x.\lambda s.\lambda \rho'.kxs)s_0\rho\\
\underline{f}\ k & = & f(\lambda a.\lambda \rho'.ka)\rho
\end{array}
$$

We first verify that:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\underline{f}'(\lambda x.\lambda s.\mathit{map}_m(\lambda x.h(s,x))(kx))s_0}\\
= & f'(\lambda x.\lambda s.\lambda \rho'.\mathit{map}_m(\lambda x.h(s,x))(kx))s_0\rho\\
= & f'(\lambda x.\lambda s.\lambda \rho'.\mathit{bind}_m\ (kx)\ (\lambda x.\mathit{return}_m(h(s,x))))s_0\rho\\
= & f'(\lambda x.\lambda s.\lambda \rho'.\mathit{bind}_m\ ((\lambda \rho''.kx)\rho')\\
& \qquad (\lambda x.(\lambda \rho''.\mathit{return}_m(h(s,x)))\rho'))s_0\rho\\
= & f'(\lambda x.\lambda s.\mathit{bind}_{tm}\ (\lambda \rho''.kx)\ (\lambda x.\mathit{return}_{tm}(h(s,x))))s_0\rho\\
= & f'(\lambda x.\lambda s.\mathit{map}_{tm}\ (\lambda x.h(s,x))\ (\lambda \rho''.kx))s_0\rho\\
& \text{(condition)}\\
= & \mathit{map}_{tm}\ g\ (f(\lambda x.\lambda \rho''.kx))\rho\\
= & \mathit{bind}_{tm}\ (f(\lambda x.\lambda \rho''.kx))\ (\lambda x.\mathit{return}_{tm}(gx))\rho\\
= & \mathit{bind}_m\ (f(\lambda x.\lambda \rho''.kx)\rho)\ (\lambda x.\mathit{return}_m(gx))\\
= & \mathit{bind}_m\ (\underline{f}k)\ (\lambda x.\mathit{return}_m(gx))\\
= & \mathit{map}_m\ g\ (\underline{f}k)
\end{array}
$$

We now set out to prove:

$$
\mathit{callcc}_{tm}(\lambda k.f'(\lambda x.\lambda s.k(gx))s_0)\rho \quad = \quad \mathit{map}_{tm}\ g\ (\mathit{callcc}_{tm} f)\rho
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{callcc}_{tm}(\lambda k.f'(\lambda x.\lambda s.k(gx))s_0)\rho}\\
= & \mathit{callcc}_m(\lambda k.(\lambda k.f'(\lambda x.\lambda s.k(gx))s_0)(\lambda a.\lambda \rho'.ka)\rho)\\
= & \mathit{callcc}_m(\lambda k.f'(\lambda x.\lambda s.\lambda \rho'.k(gx))s_0\rho)
\end{array}
$$

$$
\begin{aligned}
&\mathit{map}_{tm}\ g\ (\mathit{callcc}_{tm} f)\rho \\
&\quad= \mathit{bind}_{tm}(\mathit{callcc}_{tm} f)(\lambda x.\, \mathit{return}_{tm}(gx))\rho \\
&\quad= \mathit{bind}_{m}(\mathit{callcc}_{tm} f\ \rho)(\lambda x.\, \mathit{return}_{m}(gx)) \\
&\quad= \mathit{map}_{m}\ g\ (\mathit{callcc}_{m}(\lambda k.f(\lambda a.\lambda\rho'.ka)\rho)) \\
&\quad= \mathit{map}_{m}\ g\ (\mathit{callcc}_{m}\underline{f}) \\
&\qquad \text{(induction hypo.)} \\
&\quad= \mathit{callcc}_{m}(\lambda k.\underline{f}'(\lambda x.\lambda s.k(gx))s_0) \\
&\quad= \mathit{callcc}_{m}(\lambda k.f'(\lambda x.\lambda s.\lambda\rho'.k(gx))s_0\rho)
\end{aligned}
$$

Case "$t = \mathit{StateT}\ s$:" (States of type s are underlined.)
Let:

$$
\begin{aligned}
\underline{f}'\ k\ s_0 &= f'(\lambda x.\lambda s.\lambda\underline{s_1}.k(\underline{s_0},x)s)s_0\underline{s_0} \\
\underline{f}\ k &= f(\lambda a.\lambda\underline{s_1}.k(\underline{s_0},a))\underline{s_0} \\
\underline{g} &= \lambda(\underline{s_1},x).(\underline{s_1},gx) \\
\underline{h} &= \lambda(s,(\underline{s_2},x)).h(\underline{s_2},(s,x))
\end{aligned}
$$

We first verify that:

$$
\begin{aligned}
&\underline{f}'(\lambda x.\lambda s.\mathit{map}_m(\lambda x.\underline{h}(s,x))(kx))s_0 \\
&\quad= f'(\lambda x.\lambda s.\lambda\underline{s_1}.\mathit{map}_m(\lambda x.\underline{h}(s,x))(k(\underline{s_0},x)))s_0\underline{s_0} \\
&\quad= f'(\lambda x.\lambda s.\lambda\underline{s_1}.\mathit{bind}_m\ (k(\underline{s_0},x))\ (\lambda x.\, \mathit{return}_m(\underline{h}(s,x))))s_0\underline{s_0} \\
&\quad= f'(\lambda x.\lambda s.\lambda\underline{s_1}.\mathit{bind}_m\ ((\lambda\underline{s_2}.k(\underline{s_0},x))\underline{s_1}) \\
&\qquad\qquad\qquad (\lambda(\underline{s_2},x).(\lambda\underline{s_3}.\, \mathit{return}_m(\underline{h}(s,(\underline{s_2},x))))\underline{s_2}))s_0\underline{s_0} \\
&\quad= f'(\lambda x.\lambda s.\mathit{bind}_{tm}\ (\lambda\underline{s_2}.k(\underline{s_0},x))\ (\lambda x.\, \mathit{return}_{tm}(h(s,x))))s_0\underline{s_0} \\
&\quad= f'(\lambda x.\lambda s.\mathit{map}_{tm}\ (\lambda x.h(s,x))\ (\lambda\underline{s_2}.k(\underline{s_0},x)))s_0\underline{s_0} \\
&\qquad \text{(condition)} \\
&\quad= \mathit{map}_{tm}\ g\ (f(\lambda x.\lambda\underline{s_2}.k(\underline{s_0},x)))\underline{s_0} \\
&\quad= \mathit{bind}_{tm}\ (f(\lambda x.\lambda\underline{s_2}.k(\underline{s_2},x)))\ (\lambda x.\, \mathit{return}_{tm}(gx))\underline{s_0} \\
&\quad= \mathit{bind}_m\ (f(\lambda x.\lambda\underline{s_2}.k(\underline{s_0},x))\underline{s_0})\ (\lambda(\underline{s_2},x).\, \mathit{return}_m(\underline{s_2},gx)) \\
&\quad= \mathit{bind}_m\ (\underline{f}k)\ (\lambda(\underline{s_2},x).\, \mathit{return}_m(\underline{s_2},gx)) \\
&\quad= \mathit{map}_m\ \underline{g}\ (\underline{f}k)
\end{aligned}
$$

We now set out to prove:

$$
\mathit{callcc}_{tm}(\lambda k.f'(\lambda x.\lambda s.k(gx))s_0)\underline{s_0} \quad = \quad \mathit{map}_{tm}\ g\ (\mathit{callcc}_{tm} f)\underline{s_0}
$$

$$
\begin{aligned}
&\mathit{callcc}_{tm}(\lambda k.f'(\lambda x.\lambda s.k(gx))s_0)\underline{s_0} \\
&\quad= \mathit{callcc}_m(\lambda k.(\lambda k.f'(\lambda x.\lambda s.k(gx))s_0)(\lambda a.\lambda\underline{s_1}.k(\underline{s_0},a))\underline{s_0}) \\
&\quad= \mathit{callcc}_m(\lambda k.f'(\lambda x.\lambda s.\lambda\underline{s_1}.k(\underline{s_0},gx))s_0\underline{s_0})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{map}_{tm}\ g\ (\mathit{callcc}_{tm} f)\underline{s_0} \\
&\quad= \mathit{bind}_{tm}(\mathit{callcc}_{tm} f)(\lambda x.\, \mathit{return}_{tm}(gx))\underline{s_0} \\
&\quad= \mathit{bind}_m(\mathit{callcc}_{tm} f\ \underline{s_0})(\lambda(\underline{s_1},x).\, \mathit{return}_m(\underline{s_1},gx)) \\
&\quad= \mathit{map}_m\ (\lambda(\underline{s_1},x).(\underline{s_1},gx))\ (\mathit{callcc}_m(\lambda k.f(\lambda a.\lambda\underline{s_1}.k(\underline{s_0},a))\underline{s_0})) \\
&\quad= \mathit{map}_m\ \underline{g}\ (\mathit{callcc}_m\underline{f}) \\
&\qquad \text{(induction hypo.)} \\
&\quad= \mathit{callcc}_m(\lambda k.\underline{f}'(\lambda x.\lambda s.k(\underline{g}x))s_0) \\
&\quad= \mathit{callcc}_m(\lambda k.f'(\lambda x.\lambda s.\lambda\underline{s_1}.k(\underline{g}(\underline{s_0},x)))s_0\underline{s_0}) \\
&\quad= \mathit{callcc}_m(\lambda k.f'(\lambda x.\lambda s.\lambda\underline{s_1}.k(\underline{s_0},gx))s_0\underline{s_0})
\end{aligned}
$$

Case "$t = \mathit{ErrT}$:"
Let:

$$\begin{array}{lll}
\underline{f}'\ k\ s_0 & = & f'(\lambda x.\lambda s.k(\mathit{Ok}\ x)s)s_0 \\
\underline{f}\ k & = & f(\lambda a.k(\mathit{Ok}\ a)) \\
\underline{g} & = & \lambda a.\textbf{case}\ a\ \textbf{of}\ \ \mathit{Ok}\ x \rightarrow \mathit{Ok}\ (gx) \\
 & & \qquad\qquad\qquad \mathit{Err}\ s \rightarrow \mathit{Err}\ s \\
\underline{h} & = & \lambda(s,a).\textbf{case}\ a\ \textbf{of}\ \ \mathit{Ok}\ x \rightarrow \mathit{Ok}\ (h(s,x)) \\
 & & \qquad\qquad\qquad\qquad \mathit{Err}\ s \rightarrow \mathit{Err}\ s
\end{array}$$

We first verify that:

$$\begin{array}{ll}
\multicolumn{2}{l}{\underline{f}'(\lambda x.\lambda s.\mathit{map}_m(\lambda x.\underline{h}(s,x))(kx))s_0} \\
= & f'(\lambda x.\lambda s.\mathit{map}_m(\lambda x.\underline{h}(s,x))(k(\mathit{Ok}\ x)))s_0 \\
= & f'(\lambda x.\lambda s.\mathit{bind}_m\ (k(\mathit{Ok}\ x))\ (\lambda x.\ \mathit{return}_m(\underline{h}(s,x))))s_0 \\
= & f'(\lambda x.\lambda s.\mathit{bind}_m\ (k(\mathit{Ok}\ x)) \\
 & \qquad\qquad (\lambda x.\ \mathit{return}_m(\textbf{case}\ x\ \textbf{of}\ \ \mathit{Ok}\ y \rightarrow \mathit{Ok}\ (h(s,y)) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Err}\ s \rightarrow \mathit{Err}\ s\ )))s_0 \\
= & f'(\lambda x.\lambda s.\mathit{bind}_m\ (k(\mathit{Ok}\ x)) \\
 & \qquad\qquad (\lambda x.\textbf{case}\ a\ \textbf{of}\ \ \mathit{Ok}\ y \rightarrow \mathit{return}_m(\mathit{Ok}\ (h(s,y))) \\
 & \qquad\qquad\qquad\qquad\qquad \mathit{Err}\ s \rightarrow \mathit{return}_m(\mathit{Err}\ s)\ ))s_0 \\
= & f'(\lambda x.\lambda s.\mathit{bind}_{tm}\ (k(\mathit{Ok}\ x))\ (\lambda x.\ \mathit{return}_{tm}(h(s,x))))s_0 \\
= & f'(\lambda x.\lambda s.\mathit{map}_{tm}\ (\lambda x.h(s,x))\ (k(\mathit{Ok}\ x)))s_0 \\
 & \text{(condition)} \\
= & \mathit{map}_{tm}\ g\ (f(\lambda x.k(\mathit{Ok}\ x))) \\
= & \mathit{bind}_{tm}\ (f(\lambda x.k(\mathit{Ok}\ x)))\ (\lambda x.\ \mathit{return}_{tm}(gx)) \\
= & \mathit{bind}_m\ (\underline{f}k)\ (\lambda a.\textbf{case}\ a\ \textbf{of}\ \ \mathit{Ok}\ x \rightarrow \mathit{return}_m(\mathit{Ok}\ (gx)) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \mathit{Err}\ s \rightarrow \mathit{return}_m(\mathit{Err}\ s)) \\
= & \mathit{map}_m\ \underline{g}\ (\underline{f}k)
\end{array}$$

We now set out to prove:

$$\mathit{callcc}_{tm}(\lambda k.f'(\lambda x.\lambda s.k(gx))s_0) \quad = \quad \mathit{map}_{tm}\ g\ (\mathit{callcc}_{tm} f)$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{callcc}_{tm}(\lambda k.f'(\lambda x.\lambda s.k(gx))s_0)} \\
= & \mathit{callcc}_m(\lambda k.(\lambda k.f'(\lambda x.\lambda s.k(gx))s_0)(\lambda a.k(\mathit{Ok}\ a))) \\
= & \mathit{callcc}_m(\lambda k.f'(\lambda x.\lambda s.k(\mathit{Ok}\ (gx)))s_0)
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{map}_{tm}\ g\ (\mathit{callcc}_{tm} f)} \\
= & \mathit{bind}_{tm}(\mathit{callcc}_{tm} f)(\lambda x.\ \mathit{return}_{tm}(gx)) \\
= & \mathit{bind}_m(\mathit{callcc}_{tm} f)(\lambda a.\textbf{case}\ a\ \textbf{of}\ \ \mathit{Ok}\ x \rightarrow \mathit{return}_m(\mathit{Ok}\ (gx)) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Err}\ s \rightarrow \mathit{return}_m(\mathit{Err}\ s)\ ) \\
= & \mathit{map}_m\ \underline{g}\ (\mathit{callcc}_m \underline{f}) \\
 & \text{(induction hypo.)} \\
= & \mathit{callcc}_m(\lambda k.\underline{f}'(\lambda x.\lambda s.k(\underline{g}x))s_0) \\
= & \mathit{callcc}_m(\lambda k.\underline{f}'(\lambda x.\lambda s.k(\underline{g}(\mathit{Ok}\ x)))s_0) \\
= & \mathit{callcc}_m(\lambda k.f'(\lambda x.\lambda s.k(\mathit{Ok}\ (gx)))s_0)
\end{array}$$

Theorem 5.1 For any source language program e, we have:

$$\mathit{inEnv}\ \overline{\rho}\ E[\![e]\!] = \mathit{inEnv}\ \overline{\rho}\ N[\![e]\!]$$

Proof:
We prove the theorem by induction over the structure of expressions.
Arithmetic expressions:

$$\begin{array}{llll}
\mathit{inEnv}\ \overline{\rho}\ E[\![n]\!] & = & \mathit{inEnv}\ \overline{\rho}\ (\mathit{return}\ \ n) & (E) \\
 & = & \mathit{inEnv}\ \overline{\rho}\ N[\![n]\!] & (N)
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}\ \overline{\rho}\ E[\![e_1+e_2]\!]} & \\
= & \mathit{inEnv}\ \overline{\rho}\ \{v_1 \leftarrow E[\![e_1]\!]; v_2 \leftarrow E[\![e_2]\!]; \mathit{return}\ (v_1+v_2)\} & (E)\\
= & \{v_1 \leftarrow \mathit{inEnv}\ \overline{\rho}\ E[\![e_1]\!]; v_2 \leftarrow \mathit{inEnv}\ \overline{\rho}\ E[\![e_2]\!]; & \\
 & \quad \mathit{inEnv}\ \overline{\rho}\ \ \mathit{return}\ (v_1+v_2)\} & \text{(distribution)}\\
= & \{v_1 \leftarrow \mathit{inEnv}\ \overline{\rho}\ N[\![e_1]\!]; v_2 \leftarrow \mathit{inEnv}\ \overline{\rho}\ N[\![e_2]\!]; & \\
 & \quad \mathit{inEnv}\ \overline{\rho}\ \ \mathit{return}\ (v_1+v_2)\} & \text{(ind. hypo.)}\\
= & \mathit{inEnv}\ \overline{\rho}\ \{v_1 \leftarrow N[\![e_1]\!]; v_2 \leftarrow N[\![e_2]\!]; \mathit{return}\ (v_1+v_2)\} & \text{(distribution)}\\
= & \mathit{inEnv}\ \overline{\rho}\ N[\![e_1+e_2]\!] & (N)
\end{array}$$

Functions:

$$\begin{array}{rcll}
\mathit{inEnv}\ \overline{\rho}\ E[\![v]\!] & = & \mathit{inEnv}\ \overline{\rho}\ \{\rho \leftarrow \mathit{rdEnv}; \rho[\![v]\!]\} & (E)\\
 & = & \{\rho \leftarrow \mathit{inEnv}\ \overline{\rho}\ \mathit{rdEnv}; \mathit{inEnv}\ \overline{\rho}\ (\rho[\![v]\!])\} & \text{(distribution)}\\
 & = & \{\rho \leftarrow \mathit{return}\ \ \overline{\rho}; \mathit{inEnv}\ \overline{\rho}\ (\rho[\![v]\!])\} & \text{(cancellation)}\\
 & = & \mathit{inEnv}\ \overline{\rho}\ (\overline{\rho}[\![v]\!]) & \text{(left unit)}\\
 & = & \mathit{inEnv}\ \overline{\rho}\ \overline{v} & (\overline{\rho})\\
 & = & \mathit{inEnv}\ \overline{\rho}\ N[\![v]\!] & (N)
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}\ \overline{\rho}\ E[\![\lambda v.e]\!]} & \\
= & \mathit{inEnv}\ \overline{\rho}\ \{\rho \leftarrow \mathit{rdEnv}; \mathit{return}(\lambda c.\mathit{inEnv}\ \rho[c/[\![v]\!]]\ E[\![e]\!])\} & (E)\\
= & \{\rho \leftarrow \mathit{inEnv}\ \overline{\rho}\ \mathit{rdEnv}; & \\
 & \quad \mathit{inEnv}\ \overline{\rho}\ (\mathit{return}(\lambda c.\mathit{inEnv}\ \rho[c/[\![v]\!]]\ E[\![e]\!]))\} & \text{(distribution)}\\
= & \{\rho \leftarrow \mathit{return}\ \ \overline{\rho}; \mathit{return}(\lambda c.\mathit{inEnv}\ \rho[c/[\![v]\!]]\ E[\![e]\!])\} & \text{(cancel., unit)}\\
= & \mathit{return}(\lambda c.\mathit{inEnv}\ \overline{\rho}[c/[\![v]\!]]\ E[\![e]\!]) & \text{(left unit)}\\
= & \mathit{return}(\lambda \overline{v}.\mathit{inEnv}\ \overline{\rho}[\overline{v}/[\![v]\!]]\ E[\![e]\!]) & (\alpha\ \text{renaming})\\
= & \mathit{return}(\lambda \overline{v}.\mathit{inEnv}\ \overline{\rho}\ E[\![e]\!]) & (\overline{\rho})\\
= & \mathit{return}(\lambda \overline{v}.\mathit{inEnv}\ \overline{\rho}\ N[\![e]\!]) & \text{(ind. hypo.)}\\
= & \mathit{inEnv}\ \overline{\rho}\ N[\![\lambda v.e]\!] & (N)
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}\ \overline{\rho}\ E[\![(e_1\ e_2)_n]\!]} & \\
= & \mathit{inEnv}\ \overline{\rho}\ \{f \leftarrow E[\![e_1]\!]; \rho \leftarrow \mathit{rdEnv}; f(\mathit{inEnv}\ \rho\ E[\![e_2]\!])\} & (E)\\
= & \{f \leftarrow \mathit{inEnv}\ \overline{\rho}\ E[\![e_1]\!]; \rho \leftarrow \mathit{inEnv}\ \overline{\rho}\ \mathit{rdEnv}; & \\
 & \quad \mathit{inEnv}\ \overline{\rho}\ (f(\mathit{inEnv}\ \rho\ E[\![e_2]\!]))\} & \text{(distribution)}\\
= & \{f \leftarrow \mathit{inEnv}\ \overline{\rho}\ E[\![e_1]\!]; \rho \leftarrow \mathit{return}\ \ \overline{\rho}; & \\
 & \quad \mathit{inEnv}\ \overline{\rho}\ (f(\mathit{inEnv}\ \rho\ E[\![e_2]\!]))\} & \text{(cancellation)}\\
= & \{f \leftarrow \mathit{inEnv}\ \overline{\rho}\ E[\![e_1]\!]; \mathit{inEnv}\ \overline{\rho}\ (f(\mathit{inEnv}\ \overline{\rho}\ E[\![e_2]\!]))\} & \text{(left unit)}\\
= & \{f \leftarrow \mathit{inEnv}\ \overline{\rho}\ N[\![e_1]\!]; \mathit{inEnv}\ \overline{\rho}\ (f(\mathit{inEnv}\ \overline{\rho}\ N[\![e_2]\!]))\} & \text{(ind. hypo.)}\\
= & \mathit{inEnv}\ \overline{\rho}\ \{f \leftarrow N[\![e_1]\!]; f(\mathit{inEnv}\ \overline{\rho}\ N[\![e_2]\!])\} & \text{(distribution)}\\
= & \mathit{inEnv}\ \overline{\rho}\ N[\![(e_1\ e_2)_n]\!] & (N)
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}\ \overline{\rho}\ E[\![(e_1\ e_2)_v]\!]} & \\
= & \mathit{inEnv}\ \overline{\rho}\ \{f \leftarrow E[\![e_1]\!]; v \leftarrow E[\![e_2]\!]; f(\mathit{return}\ \ v)\} & (E)\\
= & \{f \leftarrow \mathit{inEnv}\ \overline{\rho}\ E[\![e_1]\!]; v \leftarrow \mathit{inEnv}\ \overline{\rho}\ E[\![e_2]\!]; & \\
 & \quad \mathit{inEnv}\ \overline{\rho}\ (f(\mathit{return}\ \ v))\} & \text{(distribution)}\\
= & \{f \leftarrow \mathit{inEnv}\ \overline{\rho}\ N[\![e_1]\!]; v \leftarrow \mathit{inEnv}\ \overline{\rho}\ N[\![e_2]\!]; & \\
 & \quad \mathit{inEnv}\ \overline{\rho}\ (f(\mathit{return}\ \ v))\} & \text{(ind. hypo.)}\\
= & \mathit{inEnv}\ \overline{\rho}\ \{f \leftarrow N[\![e_1]\!]; v \leftarrow N[\![e_2]\!]; f(\mathit{return}\ \ v)\} & \text{(distribution)}\\
= & \mathit{inEnv}\ \overline{\rho}\ N[\![(e_1\ e_2)_v]\!] & (N)
\end{array}$$

References and assignment:
We can prove:

$$\begin{array}{rcl}
\mathit{inEnv}\ \overline{\rho}\ E[\![\mathbf{ref}\ e]\!] & = & \mathit{inEnv}\ \overline{\rho}\ N[\![\mathbf{ref}\ e]\!]\\
\mathit{inEnv}\ \overline{\rho}\ E[\![\mathbf{deref}\ e]\!] & = & \mathit{inEnv}\ \overline{\rho}\ N[\![\mathbf{deref}\ e]\!]\\
\mathit{inEnv}\ \overline{\rho}\ E[\![e_1 := e_2]\!] & = & \mathit{inEnv}\ \overline{\rho}\ N[\![e_1 := e_2]\!]
\end{array}$$

the same way we established the case for $[\![e_1 + e_2]\!]$.

Lazy evaluation:

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}\ \overline{\rho}\ E[\![(e_1\ e_2)_l]\!]} & \\
= & \mathit{inEnv}\ \overline{\rho}\ \{f \leftarrow E[\![e_1]\!]; l \leftarrow \mathit{alloc}; \rho \leftarrow \mathit{rdEnv}; & \\
 & \qquad \textbf{let}\ \mathit{thunk}\ =\ \{v \leftarrow \mathit{inEnv}\ \rho\ E[\![e_2]\!]; \ldots\}\ \textbf{in}\ \ldots\} & (E) \\
= & \{f \leftarrow \mathit{inEnv}\ \overline{\rho}\ E[\![e_1]\!]; l \leftarrow \mathit{inEnv}\ \overline{\rho}\ \mathit{alloc}; & \\
 & \ \rho \leftarrow \mathit{inEnv}\ \overline{\rho}\ \mathit{rdEnv}; & \\
 & \ \mathit{inEnv}\ \overline{\rho}\ (\textbf{let}\ \mathit{thunk}\ =\ \{v \leftarrow \mathit{inEnv}\ \rho\ E[\![e_2]\!]; \ldots\}\ \textbf{in}\ \ldots)\} & \text{(distribution)} \\
= & \{f \leftarrow \mathit{inEnv}\ \overline{\rho}\ E[\![e_1]\!]; l \leftarrow \mathit{inEnv}\ \overline{\rho}\ \mathit{alloc}; & \\
 & \ \mathit{inEnv}\ \overline{\rho}\ (\textbf{let}\ \mathit{thunk}\ =\ \{v \leftarrow \mathit{inEnv}\ \overline{\rho}\ E[\![e_2]\!]; \ldots\}\ \textbf{in}\ \ldots)\} & \text{(can., l. unit)} \\
= & \{f \leftarrow \mathit{inEnv}\ \overline{\rho}\ N[\![e_1]\!]; l \leftarrow \mathit{inEnv}\ \overline{\rho}\ \mathit{alloc}; & \\
 & \ \mathit{inEnv}\ \overline{\rho}\ (\textbf{let}\ \mathit{thunk}\ =\ \{v \leftarrow \mathit{inEnv}\ \overline{\rho}\ N[\![e_2]\!]; \ldots\}\ \textbf{in}\ \ldots)\} & \text{(ind. hypo.)} \\
= & \mathit{inEnv}\ \overline{\rho}\ \{f \leftarrow N[\![e_1]\!]; l \leftarrow \mathit{alloc}; & \\
 & \qquad \textbf{let}\ \mathit{thunk}\ =\ \{v \leftarrow \mathit{inEnv}\ \overline{\rho}\ N[\![e_2]\!]; \ldots\}\ \textbf{in}\ \ldots\} & \text{(distribution)} \\
= & \mathit{inEnv}\ \overline{\rho}\ N[\![(e_1\ e_2)_l]\!] & (N)
\end{array}$$

Tracing:
Again, we can prove:

$$\mathit{inEnv}\ \overline{\rho}\ E[\![l\ @\ e]\!] \quad = \quad \mathit{inEnv}\ \overline{\rho}\ N[\![l\ @\ e]\!]$$

the same way we established the case for $[\![e_1 + e_2]\!]$.
First-class continuations:
We can prove:

$$\mathit{inEnv}\ \overline{\rho}\ E[\![\text{callcc}]\!] \quad = \quad \mathit{inEnv}\ \overline{\rho}\ N[\![\text{callcc}]\!]$$

the same way we established the case for $[\![n]\!]$.
Nondeterminism:
First we establish a lemma:

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}\ \rho\ (\mathit{merge}\ (\mathit{map}\ (\lambda x.\mathit{inEnv}\ \rho\ x)\ e))} & \\
= & \mathit{inEnv}\ \rho\ (\mathit{join}\ (\mathit{lift}\ (\mathit{map}\ (\lambda x.\mathit{inEnv}\ \rho\ x)\ e))) & (\mathit{merge}) \\
= & \mathit{inEnv}\ \rho\ (\mathit{join}\ (\mathit{lift}\ \{x \leftarrow e; \mathit{return}\ (\mathit{inEnv}\ \rho\ x)\})) & (\mathit{map}) \\
= & \mathit{inEnv}\ \rho\ (\mathit{join}\ \{x \leftarrow \mathit{lift}\ e; \mathit{lift}\ (\mathit{return}\ (\mathit{inEnv}\ \rho\ x))\}) & \text{(monad morphism)} \\
= & \mathit{inEnv}\ \rho\ (\mathit{join}\ \{x \leftarrow \mathit{lift}\ e; \mathit{return}\ (\mathit{inEnv}\ \rho\ x)\}) & \text{(monad morphism)} \\
= & \mathit{inEnv}\ \rho\ \{x \leftarrow \mathit{lift}\ e; a \leftarrow \mathit{return}\ (\mathit{inEnv}\ \rho\ x); a\} & (\mathit{join}) \\
= & \mathit{inEnv}\ \rho\ \{x \leftarrow \mathit{lift}\ e; \mathit{inEnv}\ \rho\ x\} & \text{(left unit)} \\
= & \{x \leftarrow \mathit{inEnv}\ \rho\ (\mathit{lift}\ e); \mathit{inEnv}\ \rho\ (\mathit{inEnv}\ \rho\ x)\} & \text{(distribution)} \\
= & \{x \leftarrow \mathit{inEnv}\ \rho\ (\mathit{lift}\ e); \mathit{inEnv}\ \rho\ x\} & \text{(overriding)} \\
= & \mathit{inEnv}\ \rho\ \{x \leftarrow \mathit{lift}\ e; x\} & \text{(distribution)} \\
= & \mathit{inEnv}\ \rho\ (\mathit{join}\ (\mathit{lift}\ e)) & (\mathit{join}) \\
= & \mathit{inEnv}\ \rho\ (\mathit{merge}\ e)) & (\mathit{merge})
\end{array}$$

Now we can prove:

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathit{inEnv}\ \overline{\rho}\ E[\![\{e_0, e_1, \ldots\}]\!]} & \\
= & \mathit{inEnv}\ \overline{\rho}\ \mathit{merge}\ [E[\![e_0]\!], E[\![e_1]\!], \ldots] & (E) \\
= & \mathit{inEnv}\ \overline{\rho}\ \mathit{merge}\ [\mathit{inEnv}\ \overline{\rho}\ E[\![e_0]\!], \mathit{inEnv}\ \overline{\rho}\ E[\![e_1]\!], \ldots] & \text{(lemma)} \\
= & \mathit{inEnv}\ \overline{\rho}\ \mathit{merge}\ [\mathit{inEnv}\ \overline{\rho}\ N[\![e_0]\!], \mathit{inEnv}\ \overline{\rho}\ N[\![e_1]\!], \ldots] & \text{(ind. hypo.)} \\
= & \mathit{inEnv}\ \overline{\rho}\ \mathit{merge}\ [N[\![e_0]\!], N[\![e_1]\!], \ldots] & \text{(lemma)} \\
= & \mathit{inEnv}\ \overline{\rho}\ N[\![\{e_0, e_1, \ldots\}]\!] & (N)
\end{array}$$

B Gofer Code for Interpreter Building Blocks

Refer to Section 4.1 for the definitions of data types `OR`, `Value`, and `Fun`. Refer to Section 4.2 for the definitions of class `InterpC`, the arithmetic building block `TermA`, and convenience funcitons `bindPrj` and `returnInj`.

B.1 Variables and Functions

```
data TermF = Var String              -- variables
           | Abs String Term         -- lambda abstraction
           | AppN Term Term          -- call-by-name application
           | AppV Term Term          -- call-by-value application

-- The variable binding environment and its helping functions.
data Env = Env [(String, M Value)]

lookupEnv :: String -> Env -> M Value
lookupEnv v (Env ((s, t):xs)) = if s == v then t
                                else lookupEnv v (Env xs)
lookupEnv v (Env [])          = err ("unbound variable: " + v)

updateEnv :: (String, Value) -> Env -> Env
updateEnv n (Env e) = Env (n:e)

-- Modular semantics of function calls.
instance InterpC TermF where

    interp (Var v) = rdEnv `bind` \env ->
                     lookupEnv v env

    interp (Abs v e) =
        rdEnv `bind` \env ->
        returnInj (\c -> inEnv (updateEnv (v, c) env) (interp e))

    interp (AppN e1 e2) =
        interp e1 `bindPrj` \f ->
        rdEnv `bind` \env ->
        f (inEnv env (interp e2))

    interp (AppV e1 e2) =
        interp e1 `bindPrj` \f ->
        interp e2 `bind` \v ->
        f (return v)
```

B.2 References and Assginment

```
data TermA = Ref Term                    -- reference creation
           | Deref Term                  -- dereference
           | Assign Term Term            -- assignment

-- The Store type and its helper functions.

type Loc = Int
data Store = Store Int [(Loc, M Value)]

allocLoc :: M Loc
allocLoc = update (\(Store l s) -> Store (l+1) s) `bind` \_ ->
           return l
```

```
readLoc :: Loc -> M Value
readLoc l = update id `bind` \(Store b s) ->
            if (l < 0 || l >= b) then err "invalid loc"
            else lookup' l s where
                lookup' l [] = err "bad access"
                lookup' l ((l', v):rest) =
                    if l == l' then v else lookup' l rest

writeLoc :: (Loc, M Value) -> M ()
writeLoc (l, v) = update id `bind` \(Store b s) ->
                  if (l < 0 || l >= b) then err "invalid loc"
                  else update (\_ -> (Store b ((l, v):s)))

instance InterpC TermR where
    interp (Ref x) =
        interp x `bind` \v ->
        allocLoc `bind` \a ->
        writeLoc (a, return v) `bind` \_ ->
        returnInj a
    interp (Deref x) =
        interp x `bindPrj` \a ->
        readLoc a
    interp (Assign l r) =
        interp l `bindPrj` \a ->
        interp r `bind` \rv ->
        writeLoc (a, return rv) `bind` \_ ->
        return rv
```

B.3 Lazy Evaluation

```
data TermL= AppL Term Term

instance InterpC TermL where
    interp (AppL e1 e2) = interp e1 `bindPrj` \f ->
                          allocLoc `bind` \l ->
                          rdEnv `bind` \env ->
                          writeLoc (l, thunk) `bind` \_ ->
                          f (readLoc l)
        where thunk = inEnv env (interp e2) `bind` \v ->
                      writeLoc (l, return v) `bind` \_ ->
                      return v
```

B.4 Tracing

```
data TermT = Trace String Term

instance InterpC TermT where
    interp (Trace l e) =
                write ("enter " ++ l) `bind` \_ ->
                interp e `bind` \v ->
                write ("leave " ++ l) `bind` \_ ->
                return v
```

B.5 First-class Continuation

```
data TermC = CallCC

-- Helper function inj_f is functionally equivalent to
-- returnInj. It helps the Gofer type checker to correctly
-- infer various higher-order types.

instance InterpC TermC where
    interp CallCC =
        inj_f (\f -> f `bindPrj` \f' ->
                    callcc (\k ->
                        (f' (inj_f (\c -> c `bind` k)))))
        where
            inj_f :: Function -> InterpM Value
            inj_f = returnInj
```

B.6 Nondeterminism

```
data TermN= Amb [Term]

instance InterpC TermN where
    interp (Amb vs) = merge (map interp vs)
```

C Gofer Code for Monad Transformers

This section lists the Gofer implementation for three monad transformers (environment, continuation and error reporting) and their associated liftings. Section 4.5 lists the Gofer implementation for the state monad transformer.

C.1 The Environment Monad Transformer

```
data EnvT r m a = EnvM (r -> m a)
unEnvM (EnvM x) = x

instance Monad m => Monad (EnvT r m) where
  return a          = EnvM (\r -> return a)
  (EnvM m) `bind` k = EnvM (\r -> m r `bind` \a ->
                                  unEnvM (k a) r)

instance MonadT (EnvT r) where
 -- lift :: m a -> EnvT r m a
    lift m = EnvM (\r -> m)

class Monad m => EnvMonad r m where
    inEnv  :: r -> m a -> m a
    rdEnv  :: m r

instance Monad m => EnvMonad r (EnvT r m) where
    inEnv r (EnvM m) = EnvM (\_ -> m r)
    rdEnv            = EnvM (\r -> return r)
```

```
-- lift EnvMonad through EnvT
instance (MonadT (EnvT r'), EnvMonad r m) =>
                        EnvMonad r (EnvT r' m) where
    inEnv r (EnvM m) = EnvM (\r' -> inEnv r (m r'))
    rdEnv            = lift rdEnv

-- lift EnvMonad through StateT
instance (MonadT (StateT s), EnvMonad r m) =>
                        EnvMonad r (StateT s m) where
    inEnv r (StateM m) = StateM (\s -> inEnv r (m s))
    rdEnv              = lift rdEnv

-- lift EnvMonad through ErrT
instance (MonadT ErrT, EnvMonad r m) =>
                EnvMonad r (ErrT m) where
    inEnv r (ErrM m) = ErrM (inEnv r m)
    rdEnv            = lift rdEnv
```

C.2 The Error Monad Transformer

```
data Err a = Ok a | Err String
data ErrT m a = ErrM (m (Err a))
unErrM (ErrM x) = x

instance Monad m => Monad (ErrT m) where
  return            = ErrM . return . Ok
  (ErrM m) `bind` k = ErrM (m `bind` \a ->
                           case a of
                             Ok x    -> unErrM (k x)
                             Err msg -> return (Err msg))

instance MonadT ErrT where
 -- lift :: m a -> ErrT m a
    lift c = ErrM (map Ok c)

class Monad m => ErrMonad m where
    err :: String -> m a

instance Monad m => ErrMonad (ErrT m) where
    err = ErrM . return . Err

instance (ErrMonad m, MonadT t) => ErrMonad (t m) where
    err = lift . err
```

C.3 The Continuation Monad Transformer

```
data ContT ans m a = ContM ((a -> m ans) -> m ans)
unContM (ContM x) = x

instance Monad m => Monad (ContT ans m) where
    return x           = ContM (\k -> k x)
    (ContM m) `bind` f =
                 ContM (\k -> m (\a -> unContM (f a) k))

instance MonadT (ContT ans) where
 -- lift  :: m a -> ContT ans m a
    lift m = ContM (\f -> m `bind` f)

class Monad m => ContMonad m where
    callcc :: ((a -> m b) -> m a) -> m a

instance Monad m => ContMonad (ContT ans m) where
    callcc f =
      ContM (\k -> unContM (f (\a -> ContM (\_ -> k a))) k)

-- lift callcc through EnvT
instance (MonadT (EnvT r), ContMonad m) =>
                       ContMonad (EnvT r m) where
    callcc f = EnvM (\r -> callcc (\k ->
                 unEnvM (f (\a -> EnvM (\r -> k a))) r))

-- lift callcc through StateT
instance (MonadT (StateT s), ContMonad m) =>
                       ContMonad (StateT s m) where
    callcc f = StateM (\s -> callcc (\k -> unStateM
                 (f (\a -> StateM (\s1 -> k (s, a)))) s))

-- lift callcc through ErrT
instance (MonadT ErrT, ContMonad m) =>
                       ContMonad (ErrT m) where
    callcc f = ErrM (callcc (\k ->
                 unErrM (f (\a -> ErrM (k (Ok a))))))
```