Monad Transformers and Modular Interpreters*

Sheng Liang   Paul Hudak   Mark Jones†

Yale University
Department of Computer Science
New Haven, CT 06520-8285
{liang,hudak,jones-mark}@cs.yale.edu

Abstract

We show how a set of *building blocks* can be used to construct programming language interpreters, and present implementations of such building blocks capable of supporting many commonly known features, including simple expressions, three different function call mechanisms (call-by-name, call-by-value and lazy evaluation), references and assignment, nondeterminism, first-class continuations, and program tracing.

The underlying mechanism of our system is *monad transformers*, a simple form of abstraction for introducing a wide range of computational behaviors, such as state, I/O, continuations, and exceptions.

Our work is significant in the following respects. First, we have succeeded in designing a fully modular interpreter based on monad transformers that includes features missing from Steele's, Espinosa's, and Wadler's earlier efforts. Second, we have found new ways to lift monad operations through monad transformers, in particular difficult cases not achieved in Moggi's original work. Third, we have demonstrated that interactions between features are reflected in liftings and that semantics can be changed by reordering monad transformers. Finally, we have implemented our interpreter in Gofer, whose constructor classes provide just the added power over Haskell's type classes to allow precise and convenient expression of our ideas. This implementation includes a method for constructing extensible unions and a form of subtyping that is interesting in its own right.

1 Introduction and Related Work

This paper discusses how to construct programming language interpreters out of modular components. We will show how an interpreter for a language with many features can be composed from building blocks, each implementing a specific feature. The interpreter writer is able to specify the set of incorporated features at a very high level.

The motivation for building modular interpreters is to isolate the semantics of individual programming language features for the purpose of better understanding, simplifying, and implementing the features and their interactions. The lack of separability of traditional denotational semantics [19] has long been recognized. Algebraic approaches such as Mosses' action semantics [16], and related efforts by Lee [13], Wand [23], Appel & Jim [1], Kelsey & Hudak [11], and others, attempt to solve parts of this problem, but fall short in several crucial ways.[1]

A ground-breaking attempt to better solve the overall problem began with Moggi's [15] proposal to use *monads* to structure denotational semantics. Wadler [21] popularized Moggi's ideas in the functional programming community by showing that many type constructors (such as *List*) were monads and how monads could be used in a variety of settings, many with an "imperative" feel (such as in Peyton Jones & Wadler [17]). Wadler's interpreter design, however, treats the interpreter monad as a monolithic structure which has to be reconstructed every time a new feature is added. More recently, Steele [18] proposed *pseudomonads* as a way to compose monads and thus build up an interpreter from smaller parts, but he failed to properly incorporate important features such as an environment and store, and struggled with restrictions in the Haskell [7] type system when trying to implement his ideas. In fact, pseudomonads are really just a special kind of *monad transformer*, first suggested by Moggi [15] as a potential way to leave a "hole" in a monad for further extension.

Returning to Moggi's original ideas, Espinosa [4] nicely formulated in Scheme a system called *Semantic Lego* — the first modular interpreter based on monad transformers — and laid out the issues in lifting. Espinosa's work reminded the programming language community (including us) — 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.

We begin by realizing the limitations of Moggi's framework and Espinosa's implementation, in particular the difficulty in dealing with complicated operations such as *callcc*, and investigate how common programming language fea-

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

†Current address: Department of Computer Science, University of Nottingham, University Park, Nottingham NG7 2RD, England. Email: mpj@cs.nott.ac.uk.

To appear in *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA*, January 1995.

[1] Very recently, Cartwright and Felleisen [3] have independently proposed a modular semantics emphasizing a *direct* semantics approach, which seems somewhat more complex than ours; the precise relationship between the approaches is, however, not yet clear.

```
type Term    =   OR TermA   -- arithmetic        type InterpM  =   StateT Store    -- memory cells
               ( OR TermF   -- functions                         ( EnvT Env        -- environment
               ( OR TermR   -- assignment                        ( ContT Answer    -- continuations
               ( OR TermL   -- lazy evaluation                   ( StateT String   -- trace output
               ( OR TermT   -- tracing                           ( ErrorT          -- error reporting
               ( OR TermC   -- callcc                              List            -- multiple results
                    TermN   -- nondeterminism                      ))))
                 )))))

type Value   =   OR Int (OR Fun ())
```

Figure 1: A modular interpreter

tures interact with each other. In so doing we are able to express more modularity and more language features than in previous work, solving several open problems that arose not only in Moggi's work, but in Steele's and Espinosa's as well. Our work also shares results with Jones and Duponcheel's [10] work on composing monads.

Independently, Espinosa [5] has continued working on monad transformers, and has also recognized the limitations of earlier approaches and proposed a solution quite different from ours. His new approach relies on a notion of "higher-order" monads (called *situated monads*) to relate different layers of monad transformers, and he has investigated the semantic implications of the order of monad transformer composition. It is not yet clear how his new approach relates to ours.

We use *Gofer* [8] syntax, which is very similar to Haskell's, throughout the paper. We choose Gofer over Haskell because of its extended type system, and we choose a functional language over mathematical syntax for three reasons: (1) it is just about as concise as mathematical syntax,[2] (2) it emphasizes the fact that our ideas are implementable (and thus have been debugged!), and (3) it shows how the relatively new idea of *constructor classes* [9] can be used to represent some rather complex typing relationships. Of course, monads can be expressed in a variety of other (higher-order) programming languages, in particular SML [14], whose type system is equally capable of expressing some of our ideas. The system could also be expressed in Scheme, but of course we would then lose the benefits of strong static type-checking. Our Gofer source code is available via anonymous ftp from *nebula.cs.yale.edu* in the directory *pub/yale-fp/modular-interpreter*.

To appreciate the extent of our results, Figure 1 gives the high-level definition of an interpreter, which is constructed in a modular way, and supports arithmetic, three different kinds of functions (call-by-name, call-by-value, and lazy), references and assignment, nondeterminism, first-class continuations, and tracing. The rest of the paper will provide the details of how the type declarations expand into a full interpreter and how each component is built. For now just note that *OR* is equivalent to the domain sum operator, and *Term*, *Value* and *InterpM* denote the source-level terms, run-time values, and supporting features (which can be regarded as the run-time system), respectively. *Int* and *Fun* are the semantic domains for integers and functions. *TermA*, *TermF*, etc. are the abstract syntax for arithmetic terms, function expressions, etc. Type constructors such as *StateT* and *ContT* are *monad transformers*; they add features, and are used to transform the monad *List* into the monad *InterpM* used by the interpreter.

To see how *Term*, *Value*, and *InterpM* constitute to modular interpreters, in the next section we will walk through some simple examples.

[2] Although (for lack of space) we do not include any proofs, all constructs (monads, monad transformers and liftings) expressed as Gofer code have been verified to satisfy the necessary properties stated in this paper.

2 An Example

A conventional interpreter maps, say, a term, environment, and store, to an answer. In contrast, a monadic interpreter such as ours maps terms to *computations*, where the details of the environment, store, etc. are "hidden". Specifically:

```
interp :: Term -> InterpM Value
```

where "*InterpM Value*" is the interpreter monad of final answers.

What makes our interpreter modular is that all three components above — the term type, the value type, and the monad — are configurable. To illustrate, if we initially wish to have an interpreter for a small arithmetic language, we can fill in the definitions as follows:

```
type Value    =  OR Int ()
type Term     =  TermA
type InterpM  =  ErrorT Id
```

The first line declares the answer domain to be the union of integers and the unit type (used as the base type). The second line defines terms as *TermA*, the abstract syntax for arithmetic operations. The final line defines the interpreter monad as a transformation of the identify monad *Id*. The monad transformer *ErrorT* accounts for the possibility of errors; in this case, arithmetic exceptions.

At this point the interpreter behaves like a calculator: [3]

```
> ((1 + 4) * 8)
40
> (3/0)
ERROR: divide by 0
```

Now if we wish to add function calls, we can extend the value domain with function types, add the abstract syntax for function calls to the term type, and apply the monad transformer *EnvT* to introduce an environment *Env*.

```
type Value    =  OR Int (OR Fun ())
type Term     =  OR TermF TermA
type InterpM  =  EnvT Env (ErrorT Id)
```

[3] For lack of space, we omit the details of parsing and printing.

Here is a test run:

$> ((\backslash x.(x+4))\ 7)$
11
$> (x+4)$
ERROR: unbound variable: x

By adding other features, we can arrive at (and go beyond) the interpreter in Figure 1. In the process of adding new source-level terms, whenever a new value domain (such as Boolean) is needed, we extend the *Value* type, and to add a new semantic feature (such as a store or continuation), we apply the corresponding monad transformer.

Why monads? In a sense, monads are nothing more than a good example of data abstraction. But they just happen to be a particularly *good* abstraction, and by using them in a disciplined (and appropriate) way, we generally obtain well-structured, modular programs. In our application, they are surprisingly useful for individually capturing the essence of a wide range of programming language features, while abstracting away from low-level details. Then with monad *transformers* we can put the individual features together, piece-by-piece in different orders, to create full-featured interpreters.

3 The Constructor Class System

For readers not familiar with the Gofer type system (in particular, constructor classes [9]), this section provides a motivating example.

Constructor classes 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:

$$map :: (a \to b) \to [a] \to [b]$$

Meanwhile, we can define similar functions for a wide range of other datatypes. 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* are both 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)
```

In building modular interpreters, we will find constructor classes extremely useful for dealing with multiple instances of monads and monad transformers (which are all type constructors).

4 Extensible Union Types

We begin with a discussion of a key idea in our framework: 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 captured by the datatype *OR*.

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

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; in particular, an extensible union may be arbitrarily nested, and we would like a *single* pair of injection and projection functions to work on all such constructions.

To achieve this, we define a type class to capture 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* datatype is used because the projection function may fail. We can now express the relationships that we desire:

```
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
```

Now we can see, for example, how the *Value* domain used in the interpreter example given earlier is actually constructed:

```
type Value  =  OR Int (OR Fun ())
type Fun    =  InterpM Value -> InterpM Value
```

With these definitions the Gofer type system will infer that *Int* and *Function* are both "subtypes" of *Value*, and the coercion functions *inj* and *prj* will be generated automatically.[4] (Note that the representation of a function is quite general — it maps *computations* to *computations*. As will be seen, this generality allows us to model both call-by-name and call-by-value semantics.)

[4] We should point out here that most of the typing problems Steele encountered disappear with the use of our extensible union types; in particular, there is no need for Steele's "towers" of datatypes.

5 The Interpreter Building Blocks

As in the example of Section 2, 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:

$$\begin{array}{l} \textbf{class}\ \mathit{InterpC}\ t\ \textbf{where} \\ \quad \mathit{interp}\ ::\ t \rightarrow \mathit{InterpM\ Value} \end{array}$$

The behavior of *interp* on unions of terms is given in the obvious way:

$$\begin{array}{l} \textbf{instance}\ (\mathit{InterpC}\ t_1,\ \mathit{InterpC}\ t_2) \Rightarrow \\ \qquad\qquad\qquad\qquad \mathit{InterpC}\ (\mathit{OR}\ t_1\ t_2)\ \textbf{where} \\ \quad \mathit{interp}\ (L\ t) \ = \ \mathit{interp}\ t \\ \quad \mathit{interp}\ (R\ t) \ = \ \mathit{interp}\ t \end{array}$$

The *interp* function mentioned in the opening example is just the method associated with the top-level type *Term*.

In the remainder of this section we define several representative *interpreter building blocks*, each an instance of class *InterpC* and written in a monadic style. We will more formally define monads later, but for now we note that the interpreter monad *InterpM* comes equipped with two basic operations:

$$\begin{array}{lll} \mathit{unit} & :: & a \rightarrow \mathit{InterpM}\ a \\ \mathit{bind} & :: & \mathit{InterpM}\ a \rightarrow (a \rightarrow \mathit{InterpM}\ b) \rightarrow \mathit{InterpM}\ b \end{array}$$

Intuitively, *InterpM a* denotes a computation returning a result of type *a*. "*Unit x*" is a null computation that just returns *x* as result, whereas "*m 'bind' k*" runs *m* and passes the result to the rest of the computation *k*. As will be seen, besides *unit* and *bind*, each interpreter building block has several other operations that are specific to its purpose.

5.1 The Arithmetic Building Block

Our (very tiny) arithmetic sublanguage is given by:

$$\begin{array}{lcl} \textbf{data}\ \mathit{TermA} & = & \mathit{Num\ Int} \\ & | & \mathit{Add\ Term\ Term} \end{array}$$

whose monadic interpretation is given by:

$$\begin{array}{l} \textbf{instance}\ \mathit{InterpC\ TermA}\ \textbf{where} \\ \quad \begin{array}{lcl} \mathit{interp}\ (\mathit{Num}\ x) & = & \mathit{unitInj}\ x \\ \mathit{interp}\ (\mathit{Add}\ x\ y) & = & \mathit{interp}\ x\ \text{`}\mathit{bindPrj}\text{`}\ \backslash i \rightarrow \\ & & \mathit{interp}\ y\ \text{`}\mathit{bindPrj}\text{`}\ \backslash j \rightarrow \\ & & \mathit{unitInj}\ ((i + j) :: \mathit{Int}) \end{array} \end{array}$$

$$\begin{array}{l} \mathit{unitInj} \qquad = \ \mathit{unit} \cdot \mathit{inj} \\ m\ \text{`}\mathit{bindPrj}\text{`}\ k \ = \\ \qquad m\ \text{`}\mathit{bind}\text{`}\ \backslash a \rightarrow \\ \qquad \textbf{case}\ (\mathit{prj}\ a)\ \textbf{of} \\ \qquad\quad \begin{array}{lcl} \mathit{Just}\ x & \rightarrow & k\ x \\ \mathit{Nothing} & \rightarrow & \mathit{err}\ \text{"run-time type error"} \end{array} \end{array}$$

$$\mathit{err}\ ::\ \mathit{String} \rightarrow \mathit{InterpM}\ a \qquad \text{-- defined later}$$

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 (*unitInj* and *bindPrj* make this a tad easier, and will be used later as well). *Err* is an operation for reporting errors to be defined later.

5.2 The Function Building Block

Our "function" sublanguage is given by:

$$\begin{array}{lcll} \textbf{data}\ \mathit{TermF} & = & \mathit{Var\ Name} & \\ & | & \mathit{LambdaN\ Name\ Term} & \text{-- cbn} \\ & | & \mathit{LambdaV\ Name\ Term} & \text{-- cbv} \\ & | & \mathit{App\ Term\ Term} & \end{array}$$

which supports two kinds of abstractions, one for call-by-name, the other for call-by-value.

We assume a type *Env* of environments that associates variable names with computations (corresponding to the "closure" mode of evaluation [2]), and that has two operations:

$$\begin{array}{lcl} \mathit{lookupEnv} & :: & \mathit{Name} \rightarrow \mathit{Env} \rightarrow \mathit{Maybe}\ (\mathit{InterpM\ Value}) \\ \mathit{extendEnv} & :: & (\mathit{Name},\ \mathit{InterpM\ Value}) \rightarrow \mathit{Env} \rightarrow \mathit{Env} \\ \textbf{type}\ \mathit{Name} & = & \mathit{String} \end{array}$$

In addition, we will define later two monadic operations, *rdEnv* and *inEnv*, that return the current environment and perform a computation in a given environment, respectively:

$$\begin{array}{lcl} \mathit{rdEnv} & :: & \mathit{InterpM\ Env} \\ \mathit{inEnv} & :: & \mathit{Env} \rightarrow \mathit{InterpM}\ a \rightarrow \mathit{InterpM}\ a \end{array}$$

The interpretation of the applicative sublanguage is then given in Figure 2.

The difference between call-by-value and call-by-name is clear: the former reduces the argument before evaluating the function body, whereas the latter does not. In a function application, the function itself is evaluated first, and *bindPrj* checks if it is indeed a function. The computation of e_2 is packaged up with the current environment to form a closure, which is then passed to f. We could just as easily realize dynamic scoping by passing not the closure, but the computation of e_2 alone.

When applying a call-by-value function, we build a computation which gets evaluated immediately upon entering the function body. Although semantically correct, this does not correspond to an efficient implementation. In practice, however, we expect that the presence of some kind of type information or a special syntax for call-by-value application will enable us to optimize away this overhead.

We note that Steele felt it unsatisfactory that his interpreter always had an environment argument, even though it was only used in the function building block. By abstracting environment-related operations as two functions (*inEnv* and *rdEnv*), we achieve exactly what Steele wished for.

5.3 The References and Assignment Building Block

A sublanguage of references and assignment is given by:

$$\begin{array}{lcl} \textbf{data}\ \mathit{TermR} & = & \mathit{Ref\ Term} \\ & | & \mathit{Deref\ Term} \\ & | & \mathit{Assign\ Term\ Term} \end{array}$$

Given a heap of memory cells and three functions for managing it:

$$\begin{array}{lcl} \mathit{allocLoc} & :: & \mathit{InterpM\ Loc} \\ \mathit{lookupLoc} & :: & \mathit{Loc} \rightarrow \mathit{InterpM\ Value} \\ \mathit{updateLoc} & :: & (\mathit{Loc},\ \mathit{InterpM\ Value}) \rightarrow \mathit{InterpM}\ () \\ \textbf{type}\ \mathit{Loc} & = & \mathit{Int} \end{array}$$

we can then give an appropriate interpretation to the new language features:

```
instance InterpC TermF where
    interp (Var v)         =  rdEnv `bind` \env ->
                              case lookupEnv v env of
                                Just val  ->  val
                                Nothing   ->  err ("unbound variable: " ++ v)
    interp (LambdaN s t)   =  rdEnv `bind` \env ->
                              unitInj (\arg -> inEnv (extendEnv (s, arg) env) (interp t))
    interp (LambdaV s t)   =  rdEnv `bind` \env ->
                              unitInj (\arg ->  arg `bind` \v ->
                                                inEnv (extendEnv (s, unit v) env) (interp t))
    interp (App e1 e2)     =  interp e1 `bindPrj` \f ->
                              rdEnv `bind` \env ->
                              f (inEnv env (interp e2))
```

Figure 2: The function building block

```
instance InterpC TermR where
    interp (Ref x) =
                interp x `bind` \val ->
                allocLoc `bind` \loc ->
                updateLoc (loc, unit val) `bind` \_ ->
                unitInj loc
    interp (Deref x) =
                interp x `bindPrj` \loc ->
                lookupLoc loc

    interp (Assign lhs rhs) =
                interp lhs `bindPrj` \loc ->
                interp rhs `bind` \val ->
                updateLoc (loc, unit val) `bind` \_ ->
                unit val
```

5.4 A Lazy Evaluation Building Block

Using this same heap of memory cells for references, we can implement "lazy" abstractions:

```
data TermL   =   LambdaL Name Term
```

whose operational semantics implies "caching" of results.

```
instance InterpC TermL where
    interp (LambdaL s t) =
          rdEnv `bind` \env ->
          unitInj (\arg ->
             allocLoc `bind` \loc ->
             let thunk = arg `bind` \v ->
                            updateLoc (loc, unit v) `bind` \_ ->
                            unit v
             in
             updateLoc (loc, thunk) `bind` \_ ->
             inEnv  (extendEnv (s, lookupLoc loc) env)
                    (interp t))
```

Upon entering a lazy function, the interpreter first allocates a memory cell and stores a thunk (updatable closure) in it. When the argument is first evaluated in the function body, the interpreter evaluates the thunk and stores the result back into the memory cell, overwriting the thunk itself.

5.5 A Program Tracing Building Block

Given a function:

```
write   ::   String -> InterpM ()
```

which writes a string output and continues the computation, we can define a "tracing" sublanguage, which attaches labels to expressions which cause a "trace record" to be invoked whenever that expression is evaluated:

```
data TermT = Trace String Term
```

```
instance InterpC TermT where
    interp (Trace l t) =
        write ("enter " ++ l) `bind` \_ ->
        interp t `bind` \v ->
        write ("leave " ++ l ++ " with:" ++ show v) `bind` \_ ->
        unit v
```

Here we see that some of the features in Kishon et al.'s system [12] are easily incorporated into our interpreter.

5.6 The Continuation Building Block

First-class continuations can be included in our language with:

```
data TermC   =   CallCC
```

Using the *callcc* semantic function (to be defined later):

```
callcc   ::   ((a -> InterpM b) -> InterpM a) -> InterpM a
```

we can give an interpretation for *CallCC*:

```
instance InterpC TermC where
    interp CallCC = unitInj (\f ->
        f `bindPrj` \f' ->
        callcc (\k -> (f' (unitInj (\a -> a `bind` k)))))
```

CallCC is interpreted as a (strict) builtin function. *Interp* in this case does nothing more than inject and project values to the right domains.

Feature	Function
Error handling	err :: $String \to InterpM\ a$
Nondeterminism	$merge$ :: $[InterpM\ a] \to InterpM$ a
Environment	$rdEnv$ :: $InterpM\ Env$ $inEnv$ :: $Env \to InterpM\ a \to InterpM$ a
Store	$allocLoc$ :: $InterpM\ Int$ $lookupLoc$ :: $Int \to InterpM\ Value$ $updateLoc$ :: $(Int, InterpM\ Value) \to InterpM\ Int$
String output	$write$ :: $String \to InterpM\ ()$
Continuations	$callcc$ :: $((a \to InterpM\ b) \to InterpM\ a) \to InterpM\ a$

Table 1: Monad operations used by the interpreter

5.7 The Nondeterminism Building Block

Our nondeterministic sublanguage is given by:

$$\textbf{data}\ TermN \quad = \quad Amb\ [Term]$$

Given a function:

$$merge \quad :: \quad [InterpM\ a] \to InterpM\ a$$

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

$$\begin{array}{l}\textbf{instance}\ InterpC\ TermN\ \textbf{where} \\ \quad interp\ (Amb\ t) = merge\ (map\ interp\ t)\end{array}$$

6 Monads With Operations

As mentioned earlier, particular monads have other operations besides *unit* and *bind*. Indeed, from the last section, it is clear that operations listed in Table 1 must be supported.

If we were building an interpreter in the traditional way, now is the time to set up the domains and implement the functions listed in the table. The major drawback of this 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 the functions in the table will have to be updated.

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

To simplify the set of operations somewhat, we note that both the store and output (used by the tracer) have to do with some notion of *state*. Thus we define *allocLoc*, *lookupLoc*, *updateLoc*, and *write* in terms of just one function:

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

for some suitably chosen s. We can read the state by passing *update* the identity function, and change the state by passing it a state transformer. For example:

$$\begin{array}{lcl} write\ msg & = & update\ (\backslash\ sofar \to sofar\ {+}{+}\ msg) \\ & & \text{`}bind\text{`}\ \backslash\_ \to unit\ () \end{array}$$

7 Monad Transformers

To get an intuitive feel for monad transformers, consider the merging of a state monad with an arbitrary monad, an example adapted from Jones's constructor class paper [9]:

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

Note that the type variable m above stands for a type constructor, a fact automatically determined by the Gofer kind inference system. It turns out that if m is a monad, so is "*StateT s m*".[5] "*StateT s*" is thus a monad transformer.

For example, if we substitute the identity monad:

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

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

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

which is the standard state monad found, for example, in Wadler's work [21].

The power of monad transformers is two-fold. First, they add operations (i.e. introduce new features) to a monad. The *StateT* monad transformer above, for example, adds state s to the monad it is applied to, and the resulting monad accepts *update* as a legitimate operation on it.

Second, monad transformers compose easily. For example, applying both "*StateT s*" and "*StateT t*" to the identity monad, we get:

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

which is the expected type signature for transforming both states s and t. 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 will be addressed in detail later. But first we define monads and monad transformers more formally, and then describe monad transformers covering the features listed in Section 5.

We can formally define monads as follows:

[5] In fact "*StateT s m*" is only legal in the current version of Gofer if *StateT* is a datatype rather than a type synonym. This does not limit our results, but does introduce superfluous data constructors that slightly complicate the presentation, so we will use *type* declarations as if they worked as *data* declarations.

$$
\begin{array}{l}
\textbf{class } \mathit{Monad}\ m\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{unit} & :: & a \to m\ a \\
\mathit{bind} & :: & m\ a \to (a \to m\ b) \to m\ b \\[4pt]
\mathit{map} & :: & (a \to b) \to m\ a \to m\ b \\
\mathit{join} & :: & m\ (m\ a) \to m\ a
\end{array} \\[4pt]
\quad \begin{array}{lll}
\mathit{map}\ f\ m & = & m\ \text{`}\mathit{bind}\text{'}\ \backslash a \to \mathit{unit}\ (f\ a) \\
\mathit{join}\ z & = & z\ \text{`}\mathit{bind}\text{'}\ \mathit{id}
\end{array}
\end{array}
$$

The two functions *map* and *join*, together with *unit* provide an equivalent definition of monads, but are easily defined (as default methods) in terms of *bind* and *unit*.

To be a monad, *bind* and *unit* must satisfy the well-known *Monad Laws* [21]:

Left unit:

$$(\mathit{unit}\ a)\ \text{`}\mathit{bind}\text{'}\ k = k\ a$$

Right unit:

$$m\ \text{`}\mathit{bind}\text{'}\ \mathit{unit} = m$$

Associativity:

$$m\ \text{`}\mathit{bind}\text{'}\ \backslash a \to (k\ a\ \text{`}\mathit{bind}\text{'}\ h) = (m\ \text{`}\mathit{bind}\text{'}\ k)\ \text{`}\mathit{bind}\text{'}\ h$$

We define a monad transformer as any type constructor t such that if m is a monad (based on the above laws), so is "$t\ m$". We can express this (other than the verification of the laws, which is generally undecidable) using the two-parameter constructor class *MonadT*:

$$
\begin{array}{l}
\textbf{class } (\mathit{Monad}\ m,\ \mathit{Monad}\ (t\ m)) \Rightarrow \mathit{MonadT}\ t\ m\ \textbf{where} \\
\quad \mathit{lift}\ \ ::\ \ m\ a \to t\ m\ a
\end{array}
$$

The member function *lift* embeds a computation in monad m into monad "$t\ m$". Furthermore, we expect a monad transformer to *add* features, without changing the nature of an existing computation. We introduce *Monad Transformer Laws* to capture the properties of *lift*:

$$
\begin{array}{rcl}
\mathit{lift} \cdot \mathit{unit}_m & = & \mathit{unit}_{tm} \\
\mathit{lift}\ (m\ \text{`}\mathit{bind}_m\text{'}\ k) & = & \mathit{lift}\ m\ \text{`}\mathit{bind}_{tm}\text{'}\ (\mathit{lift} \cdot k)
\end{array}
$$

The above laws say that lifting a null computation results in a null computation, and that lifting a sequence of computations is equivalent to first lifting them individually, and then combining them in the lifted monad.

Specific monad transformers are described in the remainder of this section. Some of these (*StateT*, *ContT*, and *ErrorT*) appear in an abstract form in Moggi's note [15]. The *environment* monad is similar to the *state reader* by Wadler [22]. The *state* and *environment* monad transformers are related to ideas found in Jones and Duponcheel's [9] [10] work.

7.1 State Monad Transformer

Recall the definition of state monad transformer *StateT*:

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

Using instance declarations, we now wish to declare both that "*StateT s m*" is a monad (given *m* is a monad), and that "*StateT s*" is a monad transformer (for each of the monad transformers defined in subsequent subsections, we will do exactly the same thing).

First, we establish the monad definition for "*StateT s m*", involving methods for *unit* and *bind*:

$$
\begin{array}{l}
\textbf{instance } \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathit{StateT}\ s\ m)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{unit}\ x & = & \backslash s \to \mathit{unit}\ (s, x) \\
m\ \text{`}\mathit{bind}\text{'}\ k & = & \backslash s_0 \to\ \ m\ s_0\ \text{`}\mathit{bind}\text{'}\ \backslash (s_1, a) \to \\
 & & \qquad k\ a\ s_1
\end{array}
\end{array}
$$

Note that these definitions are not recursive; the constructor class system automatically infers that the *bind* and *unit* appearing on the right are for monad *m*.

Next, we define "*StateT s*" as a monad transformer:

$$
\begin{array}{l}
\textbf{instance } (\mathit{Monad}\ m,\ \mathit{Monad}\ (\mathit{StateT}\ s\ m)) \Rightarrow \\
\qquad\qquad\qquad \mathit{MonadT}\ (\mathit{StateT}\ s)\ m\ \textbf{where} \\
\quad \mathit{lift}\ m\ \ =\ \ \backslash s \to m\ \text{`}\mathit{bind}\text{'}\ \backslash x \to \mathit{unit}\ (s, x)
\end{array}
$$

Note that *lift* simply runs m in the new context, while preserving the state.

Finally, as explained earlier, a state monad must support the operation *update*. To keep things modular, we define a class of state monads:

$$
\begin{array}{l}
\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{StateMonad}\ s\ m\ \textbf{where} \\
\quad \mathit{update}\ \ ::\ \ (s \to s) \to m\ s
\end{array}
$$

In particular, "*StateT s*" transforms any monad into a state monad, where "*update f*" applies *f* to the state, and returns the old state:

$$
\begin{array}{l}
\textbf{instance } \mathit{Monad}\ m \Rightarrow \mathit{StateMonad}\ s\ (\mathit{StateT}\ s\ m)\ \textbf{where} \\
\quad \mathit{update}\ f\ \ =\ \ \backslash s \to \mathit{unit}\ (f\ s, s)
\end{array}
$$

7.2 Environment Monad Transformer

"*EnvT r*" transforms any monad into an environment monad. The definition of *bind* tells us that two subsequent computation steps run under the same environment *r*. (Compare this with the state monad, where the second computation is run in the state returned by the first computation.) *Lift* just performs a computation — which cannot depend on the environment — and ignores the environment. *InEnv* ignores the environment carried inside the monad, and performs the computation in a given environment.

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

$$
\begin{array}{l}
\textbf{instance } \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathit{EnvT}\ r\ m)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{unit}\ a & = & \backslash r \to \mathit{unit}\ a \\
m\ \text{`}\mathit{bind}\text{'}\ k & = & \backslash r \to m\ r\ \text{`}\mathit{bind}\text{'}\ \backslash a \to k\ a\ r
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } (\mathit{Monad}\ m,\ \mathit{Monad}\ (\mathit{EnvT}\ r\ m)) \Rightarrow \\
\qquad\qquad\qquad \mathit{MonadT}\ (\mathit{EnvT}\ r)\ m\ \textbf{where} \\
\quad \mathit{lift}\ m\ \ =\ \ \backslash r \to m
\end{array}
$$

$$
\begin{array}{l}
\textbf{class } \mathit{Monad}\ m \Rightarrow \mathit{EnvMonad}\ \mathit{env}\ m\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{inEnv} & :: & \mathit{env} \to m\ a \to m\ a \\
\mathit{rdEnv} & :: & m\ \mathit{env}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Monad}\ m \Rightarrow \mathit{EnvMonad}\ r\ (\mathit{EnvT}\ r\ m)\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{inEnv}\ r\ m & = & \backslash\_ \to m\ r \\
\mathit{rdEnv} & = & \backslash r \to \mathit{unit}\ r
\end{array}
\end{array}
$$

7.3 Error Monad Transformer

Monad *Error* completes a series of computations if all succeed, or aborts as soon as an error occurs. The monad transformer *ErrorT* transforms a monad into an error monad.

$$
\begin{array}{lll}
\textbf{data } \mathit{Error}\ a & = & \mathit{Ok}\ a \mid \mathit{Error}\ \mathit{String} \\
\textbf{type } \mathit{ErrorT}\ m\ a & = & m\ (\mathit{Error}\ a)
\end{array}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathit{ErrorT}\ m)\ \textbf{where}\\
&\quad \mathit{unit} \quad = \quad \mathit{unit} \cdot \mathit{Ok}\\
&\quad m\ \text{`}\mathit{bind}\text{'}\ k \quad =\\
&\qquad m\ \text{`}\mathit{bind}\text{'}\ \backslash a \rightarrow\\
&\qquad \textbf{case}\ a\ \textbf{of}\\
&\qquad\quad (\mathit{Ok}\ x) \quad \rightarrow \quad k\ x\\
&\qquad\quad (\mathit{Error}\ \mathit{msg}) \quad \rightarrow \quad \mathit{unit}\ (\mathit{Error}\ \mathit{msg})
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Monad}\ m,\ \mathit{Monad}\ (\mathit{ErrorT}\ m)) \Rightarrow\\
&\qquad\qquad\qquad \mathit{MonadT}\ \mathit{ErrorT}\ m\ \textbf{where}\\
&\quad \mathit{lift} \quad = \quad \mathit{map}\ \mathit{unit}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{ErrMonad}\ m\ \textbf{where}\\
&\quad \mathit{err} \quad :: \quad \mathit{String} \rightarrow m\ a
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{ErrMonad}\ (\mathit{ErrorT}\ m)\ \textbf{where}\\
&\quad \mathit{err} \quad = \quad \mathit{unit} \cdot \mathit{Error}
\end{aligned}
$$

7.4 Continuation Monad Transformer

We define the continuation monad transformer as:

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

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathit{ContT}\ \mathit{ans}\ m)\ \textbf{where}\\
&\quad \mathit{unit}\ x \quad = \quad \backslash k \rightarrow k\ x\\
&\quad m\ \text{`}\mathit{bind}\text{'}\ f \quad = \quad \backslash k \rightarrow m\ (\backslash a \rightarrow f\ a\ k)
\end{aligned}
$$

ContT introduces an additional continuation argument (of type "$a \rightarrow m\ ans$"), and by the above definitions of *unit* and *bind*, all computations in monad "*ContT ans m*" are carried out in a continuation passing style.

Lift for "*Cont ans m*" turns out to be the same as *bind* for *m*. (It is easy to see this from the type signature.) "*Callcc f*" invokes the computation in *f*, passing it a continuation that once applied, throws away the current continuation (denoted as "_") and invokes the captured continuation *k*.

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Monad}\ m,\ \mathit{Monad}\ (\mathit{ContT}\ \mathit{ans}\ m)) \Rightarrow\\
&\qquad\qquad\qquad \mathit{MonadT}\ (\mathit{ContT}\ \mathit{ans})\ m\ \textbf{where}\\
&\quad \mathit{lift} \quad = \quad \mathit{bind}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{ContMonad}\ m\ \textbf{where}\\
&\quad \mathit{callcc} \quad :: \quad ((a \rightarrow m\ b) \rightarrow m\ a) \rightarrow m\ a
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{ContMonad}\ (\mathit{ContT}\ \mathit{ans}\ m)\ \textbf{where}\\
&\quad \mathit{callcc}\ f \quad = \quad \backslash k \rightarrow f\ (\backslash a \rightarrow \backslash\_ \rightarrow k\ a)\ k
\end{aligned}
$$

7.5 The List Monad

Jones and Duponcheel [10] have shown that lists compose with special kinds of monads called *commutative monads*. 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 takes arbitrary monads. We thus use lists as the *base* monad, upon which other transformers can be applied.

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ \mathit{List}\ \textbf{where}\\
&\quad \mathit{unit}\ x \quad = \quad [x]\\
&\quad [\,]\ \text{`}\mathit{bind}\text{'}\ k \quad = \quad [\,]\\
&\quad (x : \mathit{xs})\ \text{`}\mathit{bind}\text{'}\ k \quad = \quad k\ x +\!\!+ (\mathit{xs}\ \text{`}\mathit{bind}\text{'}\ k)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{ListMonad}\ m\ \textbf{where}\\
&\quad \mathit{merge} \quad :: \quad [m\ a] \rightarrow m\ a
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{ListMonad}\ \mathit{List}\ \textbf{where}\\
&\quad \mathit{merge} \quad = \quad \mathit{concat}
\end{aligned}
$$

8 Lifting Operations

We have introduced monad transformers that add useful operations to a given monad, but have not addressed how these operations can be carried through other layers of monad transformers, or equivalently, how a monad transformer lifts existing operations within a monad.

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 "$inEnv :: r \rightarrow m\ a \rightarrow m\ a$" through *t* results in an operation with type "$r \rightarrow t\ m\ a \rightarrow t\ m\ a$."
Given the types of operations in monad *m*:

$$
\begin{array}{rcll}
\tau & ::= & A & \text{(type constants)}\\
& | & a & \text{(type variables)}\\
& | & \tau \rightarrow \tau & \text{(function types)}\\
& | & (\tau, \tau) & \text{(product types)}\\
& | & m\ \tau & \text{(monad types)}
\end{array}
$$

$\lceil\rceil_t$ is the mapping of types across the monad transformer *t*:

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

Moggi [15] 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.

We individually consider how to lift these difficult cases. This allows us to make use of their definitions (rather than just the types), and find ways to lift them through all 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, and the more involved cases require a deeper analysis of monad structures in order 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*). What we introduced so far are able to model almost all commonly known features of sequential languages. Even so, not all of them are strictly necessary. The environment, for example, can be simulated using a state monad:

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Monad}\ m, \mathit{StateMonad}\ r\ m) \Rightarrow\\
&\qquad\qquad\qquad \mathit{EnvMonad}\ r\ m\ \textbf{where}\\
&\quad \mathit{inEnv}\ r\ m \quad = \quad \mathit{update}\ (\backslash\_ \rightarrow r)\ \text{`}\mathit{bind}\text{'}\ \backslash o \rightarrow\\
&\qquad\qquad\qquad m\ \text{`}\mathit{bind}\text{'}\ \backslash v \rightarrow\\
&\qquad\qquad\qquad \mathit{update}\ (\backslash\_ \rightarrow o)\ \text{`}\mathit{bind}\text{'}\ \backslash\_ \rightarrow\\
&\qquad\qquad\qquad \mathit{unit}\ v\\
&\quad \mathit{rdEnv} \quad = \quad \mathit{update}\ \mathit{id}
\end{aligned}
$$

Also, as is well known, error reporting can be implemented using *callcc*.

8.1 Correctness Criteria

The basic requirement of lifting is that any program which does not use the added features should behave in the same way after a monad transformer is applied. The monad transformer laws introduced in Section 7 are meant to guarantee such property for lifting a single computation. Most monad operations, however, have more general types. To deal with operations on arbitrary types, we extend Moggi's corresponding categorical approach, and define $\mathcal{L}_\tau$ as the *natural lifting* of operations of type τ along the monad transformer *t*:

$$\begin{array}{lcll}
\mathcal{L}_\tau & :: & \tau \to \lceil\tau\rceil_t & \\
 & & & \\
\mathcal{L}_A & = & id & (1)\\
\mathcal{L}_a & = & id & (2)\\
\mathcal{L}_{\tau_1\to\tau_2} & = & \backslash f \to f' \text{ such that} & \\
 & & \quad f'\cdot\mathcal{L}_{\tau_1} = \mathcal{L}_{\tau_2}\cdot f & (3)\\
\mathcal{L}_{(\tau_1,\tau_2)} & = & \backslash(a,b) \to (\mathcal{L}_{\tau_1}\ a, \mathcal{L}_{\tau_2}\ b) & (4)\\
\mathcal{L}_{m\ \tau} & = & \mathit{lift}\cdot(\mathit{map}\ \mathcal{L}_\tau) & (5)
\end{array}$$

Constant types (such as *Integer*) and type variables 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 the 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:

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

The lifting of tuples is straightforward. Finally, the *lift* operator come with the monad transformer lifts computations expressed in monad types. Note that $\mathcal{L}_\tau$ is mapped to the result of the computation, which may involve other computations.

Note that the above does *not* provide a Gofer definition for an overloaded lifting function $\mathcal{L}$. The "such that" clause in the third equation specifies a constraint, rather than a definition of f'. In practice, we first find out by hand how to lift an operation through a certain (or a class of) monad transformer, and then use the above equations to verify that such a lifting is indeed natural. Generally we require operations to be lifted naturally — although as will be seen, certain unnatural liftings change the semantics in interesting ways.

8.2 Easy Cases

Err and *update* are handled by *lift*, whereas *merge* benefits from *List* being the base monad.

$$\begin{array}{l}
\textbf{instance}\ (\mathit{ErrMonad}\ m,\ \mathit{MonadT}\ t\ m) \Rightarrow \\
\qquad\qquad\qquad \mathit{ErrMonad}\ (t\ m)\ \textbf{where} \\
\quad \mathit{err}\ \ =\ \ \mathit{lift}\cdot\mathit{err}
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ (\mathit{StateMonad}\ m,\ \mathit{MonadT}\ t\ m) \Rightarrow \\
\qquad\qquad\qquad \mathit{StateMonad}\ (t\ m)\ \textbf{where} \\
\quad \mathit{update}\ \ =\ \ \mathit{lift}\cdot\mathit{update}
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{MonadT}\ t\ \mathit{List} \Rightarrow \mathit{ListMonad}\ (t\ \mathit{List})\ \textbf{where} \\
\quad \mathit{merge}\ \ =\ \ \mathit{join}\cdot\mathit{lift}
\end{array}$$

8.3 Lifting *Callcc*

The following lifting of *callcc* through *EnvT* discards the *current* environment r' upon invoking the captured continuation *k*. The execution will continue in the environment *r* captured when *callcc* was first invoked.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{MonadT}\ (\mathit{EnvT}\ r)\ m,\ \mathit{ContMonad}\ m) \Rightarrow} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \mathit{ContMonad}\ (\mathit{EnvT}\ r\ m)\ \textbf{where}} \\
\text{-\,-}\ \mathit{callcc} & :: & ((a \to r \to m\ b) \to r \to m\ a) \to r \to m\ a \\
\quad \mathit{callcc}\ f & = & \backslash r \to \mathit{callcc}\ (\backslash k \to f\ (\backslash a \to \backslash r' \to k\ a)\ r)
\end{array}$$

The Appendix shows that if we flip the order of monad transformers and apply *ContT* to "*EnvT env m*" — in which case no lifting of *callcc* will be necessary — the current environment will be passed to the continuation. (We will see how to fix this by carefully recovering the environment when we lift *inEnv* in a moment.)

In general we can swap the order 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 Filinski's observations [6], and, in practice, provides us an opportunity to fine tune the resulting semantics.

In lifting *callcc* through "*StateT s*", 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 [20].

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{MonadT}\ (\mathit{StateT}\ s)\ m,\ \mathit{ContMonad}\ m) \Rightarrow} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \mathit{ContMonad}\ (\mathit{StateT}\ s\ m)\ \textbf{where}} \\
\text{-\,-}\ \mathit{callcc} & :: & ((a \to s \to m\ (s,b)) \to s \to m\ (s,a)) \\
 & & \quad \to s \to m\ (s,a) \\
\quad \mathit{callcc}\ f & = & \backslash s_0 \to \mathit{callcc}\ (\backslash k \to \\
 & & \quad f\ (\backslash a \to \backslash s_1 \to k\ (s_1,a))\ s_0)
\end{array}$$

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

The lifting of *inEnv* through *ErrorT* can be found in the Appendix.

8.4 Lifting *InEnv*

We only consider lifting *inEnv* through *ContT* here; the Appendix shows how to lift *inEnv* through other monad transformers.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{MonadT}\ (\mathit{ContT}\ \mathit{ans})\ m,\ \mathit{EnvMonad}\ r\ m) \Rightarrow} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \mathit{EnvMonad}\ r\ (\mathit{ContT}\ \mathit{ans}\ m)\ \textbf{where}} \\
\quad \mathit{inEnv}\ r\ c & = & \backslash k \to\ \mathit{rdEnv}\ \text{`}\mathit{bind}\text{'}\ \backslash o \to \\
 & & \qquad \mathit{inEnv}\ r\ (c\ (\mathit{inEnv}\ o\cdot k)) \\
\quad \mathit{rdEnv} & = & \mathit{lift}\ \mathit{rdEnv}
\end{array}$$

We restore the environment before invoking the continuation, sort of like popping arguments off the stack. On the other hand, an interesting (but not natural) way to lift *inEnv* is:

$$\begin{array}{l}
\textbf{instance}\ (MonadT\ (ContT\ ans)\ m,\ EnvMonad\ r\ m) \Rightarrow \\
\qquad\qquad\qquad\qquad EnvMonad\ r\ (ContT\ ans\ m)\ \textbf{where} \\
\quad inEnv\ r\ c \quad = \quad \backslash k \rightarrow inEnv\ r\ (c\ k) \\
\quad rdEnv \quad\ \ = \quad lift\ rdEnv
\end{array}$$

Here the environment is not restored when c invokes k, and thus reflects the history of dynamic execution.

9 Conclusions

We have shown how a modular monadic interpreter can be designed using two key ideas: *extensible union types* and *monad transformers*, and implemented using *constructor classes*. A key technical problem that we had to overcome was the lifting of operations through monads. Our approach also helps to clarify the interactions between various programming language features.

This paper realized Moggi's idea of a modular presentation of denotational semantics for complicated languages, and is much cleaner than the traditional approach [19]. On the practical side, our results provide new insights into designing and implementing programming languages, in particular, extensible languages, which allow the programmer to specify new features on top of existing ones.

Acknowledgements

We thank Dan Rabin, Zhong Shao, Rajiv Mirani and anonymous referees for helpful suggestions.

References

[1] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *ACM Symposium on Principles of Programming Languages*, pages 193–302, January 1989.

[2] Adrienne Bloss, Paul Hudak, and Jonathan Young. Code optimization for lazy evaluation. *Lisp and Symbolic Computation*, 1(1):147–164, 1988.

[3] Robert Cartwright and Matthias Felleisen. Extensible denotational semantics. In *Proceedings of Symposium on Theoretical Aspects of Computer Software*, pages 244–272, 1994.

[4] David Espinosa. Modular denotational semantics. Unpublished manuscript, December 1993.

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

[6] Andrzej Filinski. Representing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 446–457, New York, January 1994. ACM Press.

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

[8] Mark P. Jones. Introduction to gofer 2.20. Ftp from nebula.cs.yale.edu in the directory pub/haskell/gofer, September 1991.

[9] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, June 1993. ACM Press.

[10] Mark P. Jones and Luc Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University Department of Computer Science, New Haven, Connecticut, December 1993.

[11] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *ACM Symposium on Principles of Programming Languages*, pages 181–192, January 1989.

[12] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: A formal framework for specifying, implementing and reasoning about execution monitors. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 338–352, June 1991.

[13] Peter Lee. *Realistic Compiler Generation*. Foundations of Computing. MIT Press, 1989.

[14] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

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

[16] Peter D. Mosses. A basic abstract semantic algebra. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types: International Symposium, Sophia-Antipolis, France*, pages 87–107. Springer-Verlag, June 1984. Lecture Notes in Computer Science 173.

[17] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages*, pages 71–84. ACM, January 1993.

[18] Guy L. Steele Jr. Building interpreters by composing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 472–492, New York, January 1994. ACM Press.

[19] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[20] Andrew P. Tolmach and Andrew W. Appel. Debugging standard ML without reverse engineering. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[21] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 1–14, January 1992.

[22] Philip L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, 1990.

[23] Mitchell Wand. A semantic prototyping system. *SIGPLAN Notices, ACM Symposium on Compiler Construction*, 19(6):213–221, 1984.

A The Ordering of *ContT* and *EnvT*

It is interesting to compare the following two *callcc* functions on monad *M* and *N*, both composed from "*ContT ans*" and "*EnvT m*", but in different order.

Case 1:

$$\begin{array}{lcl}\textbf{type}\ M\ a & = & \mathit{ContT\ ans}\ (\mathit{EnvT}\ r\ m)\ a \\ & = & (a \to r \to m\ \mathit{ans}) \to r \to m\ \mathit{ans}\end{array}$$

$$\begin{array}{lcl}\mathit{callcc}\ f & = & \backslash k \to f\ (\backslash a \to \backslash\_ \to k\ a)\ k \\ & & \text{(eta convert } \backslash r \text{ and } \backslash r'\text{)} \\ & = & \backslash k \to \backslash r \to f\ (\backslash a \to \backslash\_ \to \backslash r' \to k\ a\ r')\ k\ r\end{array}$$

Case 2:

$$\begin{array}{lcl}\textbf{type}\ M\ a & = & \mathit{EnvT}\ r\ (\mathit{ContT}\ ans\ m)\ a \\ & = & r \to (a \to m\ \mathit{ans}) \to m\ \mathit{ans}\end{array}$$

$$\begin{array}{lcl}\mathit{callcc}\ f & = & \backslash r \to \mathit{callcc}\ (\backslash k \to f\ (\backslash a \to \backslash r' \to k\ a)\ r) \\ & = & \backslash r \to \backslash k \to (\backslash k' \to f\ (\backslash a \to \backslash r' \to k'\ a)\ r) \\ & & \qquad\qquad (\backslash a \to \backslash\_ \to k\ a)\ k \\ & = & \backslash r \to \backslash k \to f\ (\backslash a \to \backslash r' \to \backslash\_ \to k\ a)\ r\ k\end{array}$$

From the expansion of type *M* in case 1, we can see that both result and environment are passed to the continuation. When *callcc* invokes a continuation, it passes the *current*, rather than the captured continuation. The *callcc* function in case 2 works in the opposite way.

B Lifting *Callcc* through *ErrorT*

$$\begin{array}{l}\textbf{instance}\ (\mathit{MonadT}\ \mathit{ErrorT}\ m,\ \mathit{ContMonad}\ m) \Rightarrow \\ \qquad\qquad\qquad\qquad \mathit{ContMonad}\ (\mathit{ErrorT}\ m)\ \textbf{where} \\ \text{-- }\ \mathit{callcc} \quad :: \quad ((a \to m\ (\mathit{Error}\ a)) \to m\ (\mathit{Error}\ a)) \\ \text{-- }\qquad\qquad\qquad\quad \to m\ (\mathit{Error}\ a) \\ \quad \mathit{callcc}\ f \ = \ \mathit{callcc}\ (\backslash k \to f\ (\backslash a \to k\ (\mathit{Ok}\ a)))\end{array}$$

C Lifting *InEnv* through *EnvT*, *StateT* and *ErrorT*

$$\begin{array}{l}\textbf{instance}\ (\mathit{MonadT}\ (\mathit{EnvT}\ r')\ m,\ \mathit{EnvMonad}\ r\ m) \Rightarrow \\ \qquad\qquad\qquad\qquad \mathit{EnvMonad}\ r\ (\mathit{EnvT}\ r'\ m)\ \textbf{where} \\ \quad \mathit{inEnv}\ r\ m \ = \ \backslash r' \to \mathit{inEnv}\ r\ (m\ r') \\ \quad \mathit{rdEnv} \qquad = \ \mathit{lift}\ \mathit{rdEnv}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{MonadT}\ (\mathit{StateT}\ s)\ m,\ \mathit{EnvMonad}\ r\ m) \Rightarrow \\ \qquad\qquad\qquad\qquad \mathit{EnvMonad}\ r\ (\mathit{StateT}\ s\ m)\ \textbf{where} \\ \quad \mathit{inEnv}\ r\ m \ = \ \backslash s \to \mathit{inEnv}\ r\ (m\ s) \\ \quad \mathit{rdEnv} \qquad = \ \mathit{lift}\ \mathit{rdEnv}\end{array}$$

A function of type "$m\ a \to m\ a$" maps "*m (Error a)*" to "*m (Error a)*", thus *inEnv* stays the same after being lifted through *ErrorT*.