The essence of functional programming

Philip Wadler, University of Glasgow*

Abstract

This paper explores the use monads to structure functional programs. No prior knowledge of monads or category theory is required.

Monads increase the ease with which programs may be modified. They can mimic the effect of impure features such as exceptions, state, and continuations; and also provide effects not easily achieved with such features. The types of a program reflect which effects occur.

The first section is an extended example of the use of monads. A simple interpreter is modified to support various extra features: error messages, state, output, and non-deterministic choice. The second section describes the relation between monads and continuation-passing style. The third section sketches how monads are used in a compiler for Haskell that is written in Haskell.

1 Introduction

Shall I be pure or impure?

Pure functional languages, such as Haskell or Miranda, offer the power of lazy evaluation and the simplicity of equational reasoning. Impure functional languages, such as Standard ML or Scheme, offer a tempting spread of features such as state, exception handling, or continuations.

One factor that should influence my choice is the ease with which a program can be modified. Pure languages ease change by making manifest the data upon which each operation depends. But, sometimes, a seemingly small change may require a program in a pure language to be extensively restructured, when judicious use of an impure feature may obtain the same effect by altering a mere handful of lines.

Say I write an interpreter in a pure functional language.

To add error handling to it, I need to modify the result type to include error values, and at each recursive call to check for and handle errors appropriately. Had I used an impure language with exceptions, no such restructuring would be needed.

---

*Author's address: Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: wadler@dcs.glasgow.ac.uk.

Presented as an invited talk at *19'th Annual Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992. This version differs slightly from the conference proceedings.

To add an execution count to it, I need to modify the the result type to include such a count, and modify each recursive call to pass around such counts appropriately. Had I used an impure language with a global variable that could be incremented, no such restructuring would be needed.

To add an output instruction to it, I need to modify the result type to include an output list, and to modify each recursive call to pass around this list appropriately. Had I used an impure language that performed output as a side effect, no such restructuring would be needed.

Or I could use a *monad*.

This paper shows how to use monads to structure an interpreter so that the changes mentioned above are simple to make. In each case, all that is required is to redefine the monad and to make a few local changes. This programming style regains some of the flexibility provided by various features of impure languages. It also may apply when there is no corresponding impure feature.

The technique applies not just to interpreters, but to a wide range of functional programs. The GRASP team at Glasgow is constructing a compiler for the functional language Haskell. The compiler is itself written in Haskell, and uses monads to good effect. Though this paper concentrates on the use of monads in a program tens of lines long, it also sketches our experience using them in a program three orders of magnitude larger.

Programming with monads strongly reminiscent of continuation-passing style (CPS), and this paper explores the relationship between the two. In a sense they are equivalent: CPS arises as a special case of a monad, and any monad may be embedded in CPS by changing the answer type. But the monadic approach provides additional insight and allows a finer degree of control.

The concept of a monad comes from category theory, but this paper assumes no prior knowledge of such arcana. Rather, it is intended as a gentle introduction, with an emphasis on why abstruse theory may be of interest to computing scientists.

The examples will be given in Haskell, but no knowledge of that is needed either. What the reader will require is a passing familiarity with the basics of pure and impure functional programming; for general background see [BW87, Pau91]. The languages refered to are Haskell [HPW91], Miranda[1] [Tur90], Standard ML [MTH90], and Scheme [RC86].

Some readers will recognise that the title of this paper is a homage to Reynolds [Rey81] and that the use of monads was inspired by Moggi [Mog89a, Mog89b]. Of these matters more will be said in the conclusion. For now, please note that the word "essence" is used in a technical sense: I wish to argue that the technique described in this paper is helpful, not that it is necessary.

The remainder of this paper is organised as follows. Section 2 illustrates the use of monads to structure programs by considering several variations of an interpreter. Section 3 explores the relation between monads and continuation-passing style. Section 4 sketches how these ideas have been applied in a compiler for Haskell that is itself written in Haskell. Section 5 concludes.

[1] Miranda is a trademark of Research Software Limited.

2 Interpreting monads

This section demonstrates the thesis that monads enhance modularity, by presenting several variations of a simple interpreter for lambda calculus.

The interpreter is shown in Figure 1. It is written in Haskell. The notation `(\name -> expr)` stands for a lambda expression, and `` `name` `` is an infix operator. The type constructor `M` and functions `unitM`, `bindM`, and `showM` have to do with monads, and are explained below.

The interpreter deals with values and terms. A value is either `Wrong`, a number, or a function. The value `Wrong` indicates an error, such as an unbound variable, an attempt to add non-numbers, or an attempt to apply a non-function.

A term is either a variable, a constant, a sum, a lambda expression, or an application. The following will serve as test data.

```
term0  =  (App (Lam "x" (Add (Var "x") (Var "x")))
               (Add (Con 10) (Con 11)))
```

In more conventional notation this would be written $((\lambda x.\, x + x)\,(10 + 11))$. For the standard interpreter, evaluating `test term0` yields the string `"42"`.

The interpreter has been kept small for ease of illustration. It can easily been extended to deal with additional values, such as booleans, pairs, and lists; and additional term forms, such as conditional and fixpoint.

2.1 What is a monad?

For our purposes, a *monad* is a triple `(M,unitM,bindM)` consisting of a type constructor `M` and a pair of polymorphic functions.

```
unitM  ::  a -> M a
bindM  ::  M a -> (a -> M b) -> M b
```

These functions must satisfy three laws, which are discussed in Section 2.10.

The basic idea in converting a program to monadic form is this: *a function of type* `a -> b` *is converted to one of type* `a -> M b`. Thus, in the definition of `Value`, functions have type `Value -> M Value` rather than `Value -> Value`, and `interp` has type `Term -> Environment -> M Value` rather than type `Term -> Environment -> Value`. Similarly for the auxiliary functions `lookup`, `add`, and `apply`.

The identity function has type `a -> a`. The corresponding function in monadic form is `unitM`, which has type `a -> M a`. It takes a value into its corresponding representation in the monad.

Consider the case for constants.

```
interp (Con i) e  =  unitM (Num i)
```

The expression `(Num i)` has type `Value`, so applying `unitM` to it yields the corresponding `M Value`, as required to match the type of `interp`.

Two functions `k :: a -> b` and `h :: b -> c` may be composed by writing

```
type  Name               =  String

data  Term               =  Var Name
                         |  Con Int
                         |  Add Term Term
                         |  Lam Name Term
                         |  App Term Term

data  Value              =  Wrong
                         |  Num Int
                         |  Fun (Value -> M Value)

type  Environment        =  [(Name, Value)]

showval                  :: Value -> String
showval Wrong            =  "<wrong>"
showval (Num i)          =  showint i
showval (Fun f)          =  "<function>"

interp                   :: Term -> Environment -> M Value
interp (Var x) e         =  lookup x e
interp (Con i) e         =  unitM (Num i)
interp (Add u v) e       =  interp u e `bindM` (\a ->
                            interp v e `bindM` (\b ->
                            add a b))
interp (Lam x v) e       =  unitM (Fun (\a -> interp v ((x,a):e)))
interp (App t u) e       =  interp t e `bindM` (\f ->
                            interp u e `bindM` (\a ->
                            apply f a))

lookup                   :: Name -> Environment -> M Value
lookup x []              =  unitM Wrong
lookup x ((y,b):e)       =  if  x==y  then  unitM b  else  lookup x e

add                      :: Value -> Value -> M Value
add (Num i) (Num j)      =  unitM (Num (i+j))
add a b                  =  unitM Wrong

apply                    :: Value -> Value -> M Value
apply (Fun k) a          =  k a
apply f a                =  unitM Wrong

test                     :: Term -> String
test t                   =  showM (interp t [])
```

Figure 1: Interpretation in a monad (call-by-value)

```
\a -> let  b = k a  in  h b
```

which has type `a -> c`. (Here `\name -> expr` is a lambda expression. By convention, `a` will double as a type variable and a value variable.) Similarly, two functions in monadic form `k :: a -> M b` and `h :: b -> M c` are composed by writing

```
(\a -> k a `bindM` (\b -> h b))
```

which has type `a -> M c`. (Here `` `name` `` is Haskell notation for an infix function. The expression ``a `name` b`` is equivalent to `name a b`.) Thus `bindM` serves a role similar to a `let` expression. The three monad laws alluded to above simply insure that this form of composition is associative, and has `unitM` as a left and right identity.

Consider the case for sums.

```
interp (Add u v) e  =  interp u e  `bindM` (\a ->
                       interp v e  `bindM` (\b ->
                       add a b))
```

This can be read as follows: evaluate `u`; bind `a` to the result; evaluate `v`; bind `b` to the result; add `a` to `b`. The types work out: the calls to `interp` and `add` yield results of type `M Value`, and variables `a` and `b` have type `Value`.

Application is handled similarly; in particular, both the function and its argument are evaluated, so this interpreter is using a call-by-value strategy. An interpreter with a call-by-name strategy is discussed in Section 2.

Just as the type `Value` represents a value, the type `M Value` can be thought of as representing a computation. The purpose of `unitM` is to coerce a value into a computation; the purpose of `bindM` is to evaluate a computation, yielding a value.

Informally, `unitM` gets us into a monad, and `bindM` gets us around the monad. How do we get out of the monad? In general, such operations require a more ad hoc design. For our purposes, it will suffice to provide the following.

```
showM  ::  M Value -> String
```

This is used in `test`.

By changing the definitions of `M`, `unitM`, `bindM`, and `showM`, and making other small changes, the interpreter can be made to exhibit a wide variety of behaviours, as will now be demonstrated.

2.2 Variation zero: Standard interpreter

To begin, define the trivial monad.

```
type  I a       =  a

unitI a         =  a
a `bindI` k     =  k a
showI a         =  showval a
```

This is called the *identity* monad: `I` is the identity function on types, `unitI` is the identity function, `bindI` is postfix application, and `showI` is equivalent to `showval`.

Substitute monad `I` for monad `M` in the interpreter (that is, substitute `I`, `unitI`, `bindI`, `showI` for each occurrence of `M`, `unitM`, `bindM`, `showM`). Simplifying yields the standard meta-circular interpreter for lambda calculus:

```
interp                  :: Term -> Environment -> Value
interp (Var x) e        =  lookup x e
interp (Con i) e        =  Num i
interp (Add u v) e      =  add (interp u e) (interp v e)
interp (Lam x v) e      =  Fun (\a -> interp v ((x,a):e))
interp (App t u) e      =  apply (interp t e) (interp u e)
```

The other functions in the interpreter simplify similarly.

For this variant of the interpreter, evaluating `test term0` returns the string `"42"`, as we would expect.

2.3 Variation one: Error messages

To add error messages to the interpreter, define the following monad.

```
data  E a               =  Success a | Error String

unitE a                 =  Success a
errorE s                =  Error s

(Success a) `bindE` k   =  k a
(Error s)   `bindE` k   =  Error s

showE (Success a)       =  "Success: " ++ showval a
showE (Error s)         =  "Error: " ++ s
```

Each function in the interpreter either returns normally by yielding a value of the form `Success a`, or indicates an error by yielding a value of the form `Error s` where `s` is an error message. If `m :: E a` and `k :: a -> E b` then ``m `bindE` k`` acts as strict postfix application: if `m` succeeds then `k` is applied to the successful result; if `m` fails then so does the application. The show function displays either the successful result or the error message.

To modify the interpreter, substitute monad `E` for monad `M`, and replace each occurrence of `unitE Wrong` by a suitable call to `errorE`. The only occurrences are in `lookup`, `add`, and `apply`.

```
lookup x []     =  errorE ("unbound variable: "   ++ x)
add a b         =  errorE ("should be numbers: "  ++ showval a
                                           ++ ","  ++ showval b)
apply f a       =  errorE ("should be function: " ++ showval f)
```

No other changes are required.

Evaluating `test term0` now returns `"Success: 42"`; and evaluating

```
test (App (Con 1) (Con 2))
```

returns `"Error: should be function: 1"`.

In an impure language, this modification could be made using exceptions or continuations to signal an error.

2.4 Variation two: Error messages with positions

Let `Position` be a type that indicates a place in the source text (say, a line number). Extend the `Term` datatype with a constructor that indicates a location:

```
data  Term  =  ... | At Position Term
```

The parser will produce such terms as suitable. For instance, `(At p (App t (At q u)))` indicates that `p` is the position of the term `(App t u)` and that `q` is the position of the subterm `u`.

Based on `E`, define a new monad `P` that accepts a position to use in reporting errors.

```
type  P a       =  Position -> E a

unitP a         =  \p -> unitE a
errorP s        =  \p -> errorE (showpos p ++ ": " ++ s)

m `bindP` k     =  \p -> m p `bindE` (\x -> k x p)

showP m         =  showE (m pos0)
```

Here `unitP` discards the current position, `errorP` adds it to the error message, `bindP` passes the position to the argument and function, and `showP` passes in an initial position `pos0`. In addition, there is a function to change position.

```
resetP          :: Position -> P x -> P x
resetP q m      =  \p -> m q
```

This discards the position `p` that is passed in, replacing it with the given position `q`.

To modify the interpreter of the previous section, substitute monad `P` for monad `E` and add a case to deal with `At` terms.

```
interp (At p t) e  =  resetP p (interp t e)
```

This resets the position as indicated. No other change is required.

Without monads, or a similar technique, this modification would be far more tedious. Each clause of the interpreter would need to be rewritten to accept the current position as an additional parameter, and to pass it on as appropriate at each recursive call.

In an impure language, this modification is not quite so easy. One method is to use a state variable that contains a stack of positions. Care must be taken to maintain the state properly: push a position onto the stack on entering the `At` construct and pop a position off the stack when leaving it.

2.5 Variation three: State

To illustrate the manipulation of state, the interpreter is modified to keep count of the number of reductions that occur in computing the answer. The same technique could be used to deal with other state-dependent constructs, such as extending the interpreted language with reference values and operations that side-effect a heap.

The monad of state transformers is defined as follows.

```
type  S a        =  State -> (a, State)

unitS a          =  \s0 -> (a, s0)
m `bindS` k      =  \s0 -> let  (a,s1) = m s0
                                (b,s2) = k a s1
                           in   (b,s2)
```

A state transformer takes an initial state and returns a value paired with the new state. The unit function returns the given value and propagates the state unchanged. The bind function takes a state transformer `m :: S a` and a function `k :: a -> S b`. It passes the initial state to the transformer `m`; this yields a value paired with an intermediate state; function `k` is applied to the value, yielding a state transformer (`k a :: S b`), which is passed the intermediate state; this yields the result paired with the final state.

To model execution counts, take the state to be an integer.

```
type  State      =  Int
```

The show function is passed the initial state `0` and prints the final state as a count.

```
showS m          =  let  (a,s1) = m 0
                    in   "Value: " ++ showval a ++ "; " ++
                         "Count: " ++ showint s1
```

The current count is incremented by the following.

```
tickS            :: S ()
tickS            =  \s -> ((), s+1)
```

The value returned is the empty tuple `()` whose type is also written `()`. The typing of `tickS` makes clear that the value returned is not of interest. It is analogous to the use in an impure language of a function with result type `()`, indicating that the purpose of the function lies in a side effect.

The interpreter is modified by substituting monad `S` for monad `M`, and changing the first lines of `apply` and `add`.

```
apply (Fun k) a         =  tickS `bindS` (\() -> k a)
add (Num i) (Num j)     =  tickS `bindS` (\() -> unitS (Num (i+j)))
```

This counts one tick for each application and addition. No other changes are required.

Evaluating `test term0` now returns `"Value: 42; Count: 3"`.

A further modification extends the language to allow access to the current execution count. First, add a further operation to the monad.

```
fetchS          ::  S State
fetchS          =   \s -> (s, s)
```

This returns the current count. Second, extend the term data type, and add a new clause to the interpreter.

```
data  Term      =  ... | Count

interp Count e  =  fetchS `bindS` (\i -> unitS (Num i))
```

Evaluating `Count` fetches the number of execution steps performed so far, and returns it as the value of the term.

For example, applying `test` to

```
(Add (Add (Con 1) (Con 2)) Count)
```

returns `"Value: 4; Count: 2"`, since one addition occurs before `Count` is evaluated.

In an impure language, these modifications could be made using state to contain the count.

2.6 Variation four: Output

Next we modify the interpreter to perform output. The state monad seems a natural choice, but it's a poor one: accumulating the output into the final state means no output will be printed until the computation finishes. The following design displays output as it occurs; it depends on lazy evaluation.

The output monad is defined as follows.

```
type  O a       =  (String, a)

unitO a         =  ("", a)
m `bindO` k     =  let  (r,a) = m; (s,b) = k a  in  (r++s, b)
showO (s,a)     =  "Output: " ++ s ++ " Value: " ++ showval a
```

Each value is paired with the output produced while computing that value. The `unitO` function returns the given value and produces no output. The `bindO` function performs an application and concatenates the output produced by the argument to the output produced by the application. The `showO` function prints the output followed by the value.

The above functions propagate output but do not generate it; that is the job of the following.

```
outO            :: Value -> O ()
outO a          =  (showval a ++ "; ", ())
```

This outputs the given value followed by a semicolon.

The language is extended with an output operation. Substitute monad `O` for monad `M`, and add an a new term and corresponding clause.

```
data  Term        =  ... | Out Term

interp (Out u) e  =  interp u e `bindO` (\a  ->
                     outO a     `bindO` (\() ->
                     unitO a))
```

Evaluating `(Out u)` causes the value of `u` to be sent to the output, and returned as the value of the term.

For example, applying `test` to

```
(Add (Out (Con 41)) (Out (Con 1)))
```

returns `"Output: 41; 1; Value: 42"`.

In an impure language, this modification could be made using output as a side effect.

2.7 Variation five: Non-deterministic choice

We now modify the interpreter to deal with a non-deterministic language that returns a list of possible answers.

The monad of lists is defined as follows.

```
type  L a       =  [a]

unitL a         =  [a]
m `bindL` k     =  [ b | a <- m, b <- k a ]

zeroL           =  []
l `plusL` m     =  l ++ m

showL m         =  showlist [ showval a | a <- m ]
```

This is expressed with the usual list comprehension notation. The function `showlist` takes a list of strings into a string, with appropriate punctuation.

The interpreted language is extended with two new constructs. Substitute monad `L` for monad `M`, and add two new terms and appropriate clauses.

```
data Term               =  ... | Fail | Amb Term Term

interp Fail e           =  zeroL
interp (Amb u v) e      =  interp u e `plusL` interp v e
```

Evaluating `Fail` returns no value, and evaluating `(Amb u v)` returns all values returned by `u` or `v`.

For example, applying `test` to

```
(App (Lam "x" (Add (Var "x") (Var "x"))) (Amb (Con 1) (Con 2)))
```

returns `"[2,4]"`.

It is more difficult to see how to make this change in an impure language. Perhaps one might create some form of coroutine facility.

2.8 Variation six: Backwards state

Return now to the state example of Section 2.5. Lazy evaluation makes possible a strange variation: the state may be propogated *backward*.

All that is required is to change the definition of `bindS`.

```
m `bindS` k     =  \s2 -> let  (a,s0) = m s1
                               (b,s1) = k a s2
                          in   (b,s0)
```

This takes the *final* state as input, and returns the *initial* state as output. As before, the value `a` is generated by `m` and passed to `k`. But now the initial state is passed to `k`, the intermediate state goes from `k` to `m`, and the final state is returned by `m`. The two clauses in the `let` expression are mutually recursive, so this works only in a lazy language.

The `Count` term defined in Section 2.5 now returns the number of steps to be performed between its evaluation and the *end* of execution. As before, applying `test` to

```
(Add (Add (Con 1) (Con 2)) Count)
```

returns `"Value: 4; Count: 2"`, but for a different reason: one addition occurs *after* the point at which `Count` is evaluated. An unresolvable mutual dependence, known as a *black hole*, would arise in the unfortunate situation where the number of steps yet to be performed depends on the value returned by `Count`. In such a case the interpreter would fail to terminate or terminate abnormally.

This example may seem contrived, but this monad arises in practice. John Hughes and I discovered it by analysing a text processing algorithm that passes information both from left to right and right to left.

To make this change in an impure language is left as an exercise for masochistic readers.

2.9 Call-by-name interpreter

The interpreter of Figure 1 is call-by-value. This can be seen immediately from the types. Functions are represented by the type `Value -> M Value`, so the argument to a function is a value, though the result of applying a function is a computation.

The corresponding call-by-name interpreter is shown in Figure 2. Only the types and functions that differ from Figure 1 are shown. The type used to represent functions is now `M Value -> M Value`, so the argument to a function is now a computation. Similarly, the environment is changed to contain computations rather than values. The code for interpreting constants and addition is unchanged. The code for variables and lambda

```
data  Value             =  Wrong
                        |  Num Int
                        |  Fun (M Value -> M Value)

type  Environment       =  [(Name, M Value)]

interp                  :: Term -> Environment -> M Value
interp (Var x) e        =  lookup x e
interp (Con i) e        =  unitM (Num i)
interp (Add u v) e      =  interp u e `bindM` (\a ->
                           interp v e `bindM` (\b ->
                           add a b))
interp (Lam x v) e      =  unitM (Fun (\m -> interp v ((x,m):e)))
interp (App t u) e      =  interp t e `bindM` (\f ->
                           apply f (interp u e))

lookup                  :: Name -> Environment -> M Value
lookup x []             =  unitM Wrong
lookup x ((y,n):e)      =  if  x==y  then  n  else  lookup x e

apply                   :: Value -> M Value -> M Value
apply (Fun h) m         =  h m
apply f m               =  unitM Wrong
```

Figure 2: Interpretation in a monad (call-by-name)

abstraction looks the same but has changed subtly: previously variables were bound to values, now they are bound to computations. (Hence a small change in `lookup`: a call to `unitM` has vanished.) The code for application does change: now the function is evaluated but not the argument.

The new interpreter can be modified in the same way as the old one.

If modified for execution counts as in Section 2.5, the cost of an argument is counted each time it is evaluated. Hence evaluating `test term0` now returns the string `"Value: 42; Count: 4"`, because the cost of adding `10` to `11` is counted twice. (Compare this with a count of `3` for the call-by-value version.)

If modified for a non-deterministic language as in Section 2.7, then a term may return a different value each time it is evaluated. For example, applying `test` to

```
(App (Lam "x" (Add (Var "x") (Var "x"))) (Amb (Con 1) (Con 2)))
```

now returns `"[2,3,3,4]"`. (Compare this with `"[2,4]"` for the call-by-value version).

An advantage of the monadic style is that the types make clear where effects occur. Thus, one can distinguish call-by-value from call-by-name simply by examining the types. If one uses impure features in place of monads, the clues to behaviour are more obscure.

2.10 Monad laws

For (M,unitM,bindM) to qualify as a monad, the following laws must be satisfied.

```
Left unit:     (unitM a) `bindM` k  =  k a

Right unit:    m `bindM` unitM  =  m

Associative:      m `bindM` (\a -> (k a) `bindM` (\b -> h b))
               =  (m `bindM` (\a -> k a)) `bindM` (\b -> h b)
```

These laws guarantee that monadic composition, as discussed in Section 2.1, is associative and has a left and right unit. It is easy to verify that the monads described in this paper do satisfy these laws.

To demonstrate the utility of these laws, consider the task of proving that

(Add t (Add u v)) and (Add (Add t u) v)

always return the same value.

Simplify the left term:

```
   interp (Add t (Add u v)) e
=  interp t e            `bindM` (\a ->
   interp (Add u v) e    `bindM` (\y ->
   add a y))
=  interp t e            `bindM` (\a ->
   (interp u e           `bindM` (\b ->
    interp v e           `bindM` (\c ->
    add b c)))           `bindM` (\y ->
   add a y))
=  interp t e            `bindM` (\a ->
   interp u e            `bindM` (\b ->
   interp v e            `bindM` (\c ->
   add b c               `bindM` (\y ->
   add a y)))).
```

The first two steps are simple unfolding; the third step is justified by the associative law. Similarly, simplify the right term:

```
   interp (Add (Add t u) v) e
=  interp t e            `bindM` (\a ->
   interp u e            `bindM` (\b ->
   interp v e            `bindM` (\c ->
   add a b               `bindM` (\x ->
   add x c)))).
```

Again, this is two unfold steps and a use of the associative law. It remains to prove that

```
add a b `bindM` (\x -> add x c)  =  add b c `bindM` (\y -> add y a).
```

This is done by case analysis. If `a`, `b`, `c` have the forms `Num i`, `Num j`, `Num k` then the result is `unitM (i+j+k)`, as follows from two uses of the left unit law and the associativity of addition; otherwise the result is `Wrong`, also by the left unit law.

The above proof is trivial. Without the monad laws, it would be impossible.

As another example, note that for each monad we can define the following operations.

```
mapM            :: (a -> b) -> (M a -> M b)
mapM f m        =  m `bindM` (\a -> unitM (f a))

joinM           :: M (M a) -> M a
joinM z         =  z `bindM` (\m -> m)
```

For the list monad of Section 2.7, `mapM` is the familiar map function, and `joinM` concatenates a list of lists. Using `id` for the identity function (`id x = x`), and `(.)` for function composition (`(f.g) x = f (g x)`), one can then formulate a number of laws.

```
mapM id      =  id
mapM (f.g)   =  mapM f . mapM g

mapM f . unitM  =  unitM . f
mapM f . joinM  =  joinM . mapM (mapM f)

joinM . unitM       =  id
joinM . mapM unitM  =  id
joinM . mapM joinM  =  joinM . joinM

m `bindM` k  =  joinM (mapM k m)
```

The proof of each is a simple consequence of the three monad laws.

Often, monads are defined not in terms of `unitM` and `bindM`, but rather in terms of `unitM`, `jointM`, and `mapM` [Mac71, LS86, Mog89a, Wad90]. The three monad laws are replaced by the first seven of the eight laws above. If one defines `bindM` by the eighth law, then the three monad laws follow. Hence the two definitions are equivalent.

As described in [Wad90], the list comprehension notation generalises to an arbitrary monad. That paper gives the following translations:

```
[ t ]                    =  unitM t
[ t | x <- u ]           =  mapM (\x -> t) u
[ t | x <- u, y <- v]    =  joinM (mapM (\x -> mapM (\y -> t) v) u)
```

For the list monad, this yields the usual notion of list comprehension. In the notation of this paper, the translation may be expressed as follows.

```
[ t ]                    =  unitM t
[ t | x <- u ]           =  u `bindM` (\x -> unitM t)
[ t | x <- u, y <- v ]   =  u `bindM` (\x -> v `bindM` (\y -> unitM t))
```

The notation on the right, if not a comprehension, is at least comprehensible. The equivalence of the two translations follows from the monad laws.

3 Continuing monads

The purpose of this section is to compare the monadic style advocated in Section 2 with continuation-passing style (CPS).

Continuation-passing style was first developed for use with denotational semantics [Rey72, Plo75]. It provides fine control over the execution order of a program, and has become popular as an intermediate language for compilers [SS76, AJ89]. This paper stresses the modularity afforded by CPS, and in this sense has similar goals to the work of Danvy and Filinski [DF90].

3.1 CPS interpreter

The monad of continuations is defined as follows.

```
type  K a        =  (a -> Answer) -> Answer

unitK a          =  \c -> c a
m `bindK` k      =  \c -> m (\a -> k a c)
```

In CPS, a value `a` (of type `a`) is represented by a function that takes a continuation `c` (of type `a -> Answer`) and applies the continuation to the value, yielding the final result `c a` (of type `Answer`). Thus, `unitK a` yields the CPS representation of `a`. If `m :: K a` and `k :: a -> K b`, then ``m `bindK` k`` acts as follows: bind `c` to the current continuation, evaluate `m`, bind the result to `a`, and apply `k` to `a` with continuation `c`.

Substituting monad `K` for monad `M` in the interpreter and simplifying yields an interpreter written in CPS.

```
interp  :: Term -> Environment -> (Value -> Answer) -> Answer
interp (Var x) e        =  \c -> lookup x e c
interp (Con i) e        =  \c -> c (Num i)
interp (Add u v) e      =  \c -> interp u e (\a ->
                                 interp v e (\b ->
                                 add a b c))
interp (Lam x v) e      =  \c -> c (Fun (\a -> interp v ((x,a):e)))
interp (App t u) e      =  \c -> interp t e (\f ->
                                 interp u e (\a ->
                                 apply f a c))
```

The functions `lookup`, `add`, and `apply` now also take continuations. The line defining `Add` can be read: Let `c` be the current continuation, evaluate `u`, bind `a` to the result, evaluate `v`, bind `b` to the result, and add `a` to `b` with continuation `c`.

This reading is closely related to the monadic reading given in Section 2.1, and indeed the CPS and monadic versions are quite similar: the CPS version can be derived from the monadic one by simply eliding each occurrence of `` `bindM` ``, and adding bits to the front and end to capture and pass on the continuation `c`. The second argument to `` `bindM` `` has type `a -> (b -> Answer) -> Answer`; this is what `k` ranges over. A continuation has type `b -> Answer`; this is what `c` ranges over. Both `k` and `c` serve similar roles, acting as continuations at different levels.

The `Answer` type may be any type rich enough to represent the final result of a computation. One choice is to take an answer to be a value.

```
type  Answer    =  Value
```

This determines the definition of `showK`.

```
showK m         =  showval (m id)
```

Here `m :: K Value` is passed the identity function `id :: Value -> Value` as a continuation, and the resulting `Value` is converted to a string. Evaluating `test term0` returns `"42"`, as before.

Other choices for the `Answer` type will be considered in Section 3.3

3.2 Call with current continuation

Having converted our interpreter to CPS, it is now straightforward to add the call with current continuation (callcc) operation, found in Scheme [RC86] and Standard ML of New Jersey [DHM91].

The following operation captures the current continuation and passes it into the current expression.

```
callccK         :: ((a -> K b) -> K a) -> K a
callccK h       =  \c -> let  k a = \d -> c a  in  h k c
```

The argument to `callccK` is a function `h`, which is passed a function `k` of type `(a -> K b)`. If `k` is called with argument `a`, it ignores its continuation `d` and passes `a` to the captured continuation `c` instead.

To add callcc to the interpreted language, add an appropriate term and a new case to the interpreter.

```
data  Term                =  ... | Callcc Name Term

interp (Callcc x v) e    =  callccK (\k -> interp v ((x, Fun k):e))
```

This uses `callccK` to capture the current continuation `k`, and evaluates `v` with `x` bound to a function that acts as `k`.

For example, applying `test` to

```
(Add (Con 1) (Callcc "k" (Add (Con 2) (App (Var "k") (Con 4)))))
```

returns `"5"`.

3.3 Monads and CPS

We have seen that by choosing a suitable monad, the monad interpreter becomes a CPS interpreter. A converse property is also true: by choosing a suitable space of answers, a CPS interpreter can act as a monad interpreter.

The general trick is as follows. To achieve the effects of a monad `M` in CPS, redefine the answer type to include an application of `M`.

```
type  Answer    =  M Value
```

The definition of `showK` is modified accordingly.

```
showK n         =  showM (n unitM)
```

Here `n :: K Value` is passed `unitM :: Value -> M Value` as a continuation, and the resulting `M Value` is converted to a string by `showM`.

Just as `unitM` converts a value of type `a` into type `M a`, values of type `M a` can be converted into type `K a` as follows.

```
promoteK        ::  M a -> K a
promoteK m      =   \c -> m `bindM` c
```

Since `m :: M a` and `c :: a -> M Value`, the type of ``m `bindM` c`` is `M Value`, as required.

For example, to incorporate error messages, take `M` to be the monad `E` defined in Section 2.3. We then calculate as follows:

```
errorK    :: String -> (a -> E Value) -> E Value
errorK s  =  promoteK (errorE s)
          =  \c -> (errorE s) `bindE` c
          =  \c -> Error s `bindE` c
          =  \c -> Error s
```

The equalities follow by applying the definitions of `promoteK`, `errorE`, and `bindE`, respectively. We can take the last line as the definition of `errorK`. As we would expect, this simply ignores the continuation and returns the error as the final result.

The last section stressed that monads support modularity. For example, modifying the monadic interpreter to handle errors requires few changes: one only has to substitute monad `E` for monad `M` and introduce calls to `errorE` at appropriate places. CPS supports modularity in a similar way. For example, modifying the CPS interpreter to handle errors is equally simple: one only has to change the definitions of `Answer` and `test`, and introduce calls to `errorK` at appropriate places.

Execution counts (as in Section 2.5) and output (as in Section 2.6) may be incorporated into continuation-passing style similarly. For execution counts, take `Answer = S Value` and calculate a continuation version of `tickS`.

```
tickK    ::  (() -> S Value) -> S Value
tickK    =   promoteK tickS
         =   \c -> tickS `bindS` c
         =   \c -> (\s -> ((), s+1)) `bindS` c
         =   \c -> \s -> c () (s+1)
```

For output, take `Answer = O Value` and calculate a continuation version of `outO`.

```
outK     ::  Value -> (Value -> O Value) -> O Value
outK a   =   promoteK (outO a)
         =   \c -> (outO a) `bindO` c
         =   \c -> (showval a ++ "; ", ()) `bindO` c
         =   \c -> let (s,b) = c () in (showval a ++ "; " ++ s, b)
```

In both cases, the modifications to the CPS version of the interpreter are as simple as those to the monadic version.

3.4 Monads vs. CPS

Given the results of the previous section, one may wonder whether there is any real difference between monads and CPS. With monads, one writes

```
m `bindM` (\a -> k a)
```

and with CPS one writes

```
(\c -> m (\a -> k a c))
```

and the choice between these seems little more than a matter of taste.

There is a difference. Each of the monad types we have described may be turned into an abstract data type, and that provides somewhat finer control than CPS. For instance, we have seen that the CPS analogue of the monad type `S a` is the type

```
(a -> S Value) -> S Value.
```

This latter type contains values such as

```
\c -> \s -> (Wrong, c).
```

This provides an error escape: it ignores the current continuation and always returns `Wrong`. The state monad `S` provides no such escape facility. With monads, one can choose whether or not to provide an escape facility; CPS provides no such choice.

We can recover this facility for CPS by turning continuations into an abstract data type, and providing `unitK` and `bindK` as operations, but not providing `callccK`. So CPS can provide the same fine control as monads – if CPS is expressed as a monad!

Perhaps a more significant difference between monads and CPS is the change of viewpoint. Monads focus attention on the question of exactly what abstract operations are required, what laws they satisfy, and how one can combine the features represented by different monads.

4 Experiencing monads

Each phase of the Haskell compiler is associated with a monad.

The *type inference* phase uses a monad with an error component (similar to `E` in Section 2.3), a position component (similar to `P` in Section 2.4), and two state components (similar to `S` in Section 2.5). The state components are a name supply, used to generate unique new variable names, and a current substitution, used for unification.

The *simplification* phase uses a monad with a single single state component, which is again a name supply.

The *code generator* phase uses a monad with three state components: a list of the code generated so far, a table associating variable names with addressing modes, and a second table that caches what is known about the state of the stack at execution time.

In each case, the use of a monad greatly simplifies bookkeeping. The type inferencer would be extremely cluttered if it was necessary to mention explicitly at each step how the current substitution, name supply, and error information are propagated; for a hint of the problems, see [Han87]. The monads used have been altered several times without difficulty. The change to the interpreter described in Section 2.4 was based on a similar change made to the compiler.

The compiler has just begun to generate code, and a full assesment lies in the future. Our early experience supports the claim that monads enhance modularity.

5 Conclusion

5.1 The future

This work raises a number of questions for the future.

What are the limits of this technique? It would be desirable to characterise what sort of language features can be captured by monads, and what sort cannot. Call-by-value and call-by-name translations of lambda calculus into a monad are well known; it remains an open question whether there might be a call-by-need translation that evaluates each argument at most once.

Is syntactic support desirable? The technique given here, while workable, has a certain syntactic clumsiness. It may be better to provide an alternative syntax. One possibility is to provide

```
letM  a <- m  in  k a
```

as alternative syntax for `m 'bindM' (\a -> k a)`. Another possiblity arises from monad comprehensions [Wad90].

What about efficiency? The style advocated here makes heavy use of data abstraction and higher-order functions. It remains to be seen what impact this has on efficiency, and the GRASP team looks forward to examining the performance of our completed Haskell compiler. We are hopeful, since we have placed high priority on making the relevant features inexpensive.

How does one combine monads? The monads used in the Haskell compiler involve a combination of features; for instance, the type inferencer combines state and exceptions. There is no general technique for combining two arbitrary monads. However, Section 3.3 shows how to combine continuations with any other monad; and similar techniques are available for the state, exception, and output monads [Mog89a, Mog89b]. One might form a library of standard monads with standard ways of combining them. This would be aided by parameterised modules, which are present in Miranda and Standard ML, but absent in Haskell.

Should certain monads be provided as primitive? Monads may encapsulate impure effects in a pure way. For example, when the state is an array, the state monad can safely update the array by overwriting, as described in [Wad90]. Kevin Hammond and I have built an interface that allows Haskell programs to call C routines, using monads to sequence the calls and preserve referential transparency. The effect is similar to the "abstract continuations" used in Hope+C [Per87].

How do monads compare to other approaches to state? Several new approaches to state in pure functional languages have emerged recently, based on various type disciplines [GH90, SRI91, Wad91]. These need to be compared with each other and with the monad approach.

Can type inference help? By examining where monads appear in the types of a program, one determines in effect where impure features are used. In this sense, the use of monads is similar to the use of *effect systems* as advocated by Gifford, Jouvelot, and others, in which a type system infers where effects occur [GL86, JG91]. An intriguing question is whether a similar form of type inference could apply to a language based on monads.

5.2 The past

Finally, something should be said about the origin of these ideas.

The notion of monad comes from category theory [Mac71, LS86]. It first arose in the area of homological algebra, but later was recognised (due to the work of Kleisli and of Eilenberg and Moore) to have much wider applications. Its importance emerged slowly: in early days, it was not even given a proper name, but called simply a "standard construction" or a "triple". The formulation used here is due to Kleisli.

Eugenio Moggi proposed that monads provide a useful structuring tool for denotational semantics [Mog89a, Mog89b]. He showed how lambda calculus could be given call-by-value and call-by-name semantics in an arbitrary monad, and how monads could encapsulate a wide variety of programming language features such as state, exception handling, and continuations.

Independent of Moggi, but at about the same time, Michael Spivey proposed that monads provide a useful structuring tool for exception handling in pure functional languages, and demonstrated this thesis with an elegant program for term rewriting [Spi90]. He showed how monads could treat exceptions (as in Section 2.3) and non-deterministic choice (as in Section 2.7) in a common framework, thus capturing precisely a notion that

I had groped towards years earlier [Wad85].

Inspired by Moggi and Spivey, I proposed monads as a general technique for structuring functional programs. My early proposals were based on a special syntax for monads, that generalised list comprehensions [Wad90]. This was unfortunate, in that it led many to think a special syntax was needed. This new presentation is designed to convey that monads can be profitably applied to structure programs today with existing languages.

A key observation of Moggi's was that *values* and *computations* should be assigned different types: the value type `a` is distinct from the computation type `M a`. In a call-by-value language, functions take values into computations (as in `a -> M b`); in a call-by-name language, functions take computations into computations (as in `M a -> M b`).

John Reynolds made exactly the same point a decade ago [Rey81]. The essence of Algol, according to Reynolds, is a programming language that distinguishes data types from phrase types. In his work data types (such as `int`) play the roles of values, and phrase types (such as `int exp`) play the role of computations, and the same distinction between call-by-value and call-by-name appears. These ideas form the basis for the design of Forsythe [Rey89a]. But the vital `unitM` and `bindM` operations do not appear in Reynolds' work.

This is not the only time that John Reynolds has been a decade ahead of the rest of us. Among other things, he was an early promoter of continuation-passing style [Rey72] and the first to apply category theory to language design [Rey80, Rey81]. One intriguing aspect of his recent work is the use of intersection types [Rey89a, Rey89b, Rey91], so perhaps we should expect an upsurge of interest in that topic early in the next millenium.

This paper demonstrates that monads provide a helpful structuring technique for functional programs, and that the essence of impure features can be captured by the use of monads in a pure functional language.

In Reynolds' sense of the word, the essence of Standard ML is Haskell.

Acknowledgements. The work on the Haskell compiler reported here is a joint effort of the GRASP team, whose other members are Cordy Hall, Kevin Hammond, Will Partain, and Simon Peyton Jones. For helpful comments on this work, I'm grateful to Donald Brady, Geoffrey Burn, Stephen Eldridge, John Hughes, David King, John Launchbury, Muffy Thomas, and David Watt.

References

[AJ89] A. Appel and T. Jim, Continuation-passing, closure-passing style. In *16'th Symposium on Principles of Programming Languages*, Austin, Texas; ACM, January 1989.

[BW87] R. Bird and P. Wadler, *Introduction to Functional Programming.* Prentice Hall, 1987.

[DF90] O. Danvy and A. Filinski, Abstracting control. In *Conference on Lisp and Functional Programming*, Nice, France; ACM, June 1990.

[DHM91] B. Duba, R. Harper, and D. MacQueen, Typing first-class continuations in ML. In *18'th Symposium on Principles of Programming Languages*, Orlando, Florida; ACM, January 1991.

[GH90] J. Guzmán and P. Hudak, Single-threaded polymorphic lambda calculus. In *Symposium on Logic in Computer Science*, Philadelphia, Pennsylvania; IEEE, June 1990.

[GL86] D. K. Gifford and J. M. Lucassen, Integrating functional and imperative programming. In *Conference on Lisp and Functional Programming*, 28–39, Cambridge, Massachusetts; ACM, August 1986.

[Han87] P. Hancock, A type checker. Chapter 9 of Simon Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[HPW91] P. Hudak, S. Peyton Jones and P. Wadler, editors, *Report on the Programming Language Haskell: Version 1.1.* Technical report, Yale University and Glasgow University, August 1991.

[JG91] P. Jouvelot and D. Gifford, Algebraic reconstruction of types and effects. In *18'th ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 1991.

[LS86] J. Lambek and P. Scott, *Introduction to Higher Order Categorical Logic*, Cambridge University Press, 1986.

[Mac71] S. Mac Lane, *Categories for the Working Mathematician*, Springer-Verlag, 1971.

[Mog89a] E. Moggi, Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, Asilomar, California; IEEE, June 1989. (A longer version is available as a technical report from the University of Edinburgh.)

[Mog89b] E. Moggi, An abstract view of programming languges. Course notes, University of Edinburgh.

[MTH90] R. Milner, M. Tofte, and R. Harper, *The definition of Standard ML.* MIT Press, 1990.

[Pau91] L. C. Paulson, *ML for the Working Programmer.* Cambridge University Press, 1991.

[Per87] N. Perry, Hope+C, a continuation extension for Hope+. Imperial College, Department of Computing, Technical report IC/FPR/LANG/2.5.1/21, November 1987.

[Plo75] G. Plotkin, Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[RC86] J. Rees and W. Clinger (eds.), The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.

[Rey72] J. Reynolds, Definitional interpreters for higher-order programming languages. In *25'th ACM National Conference*, 717–740, 1972.

[Rey80] J. Reynolds, Using category theory to design implicit conversion and generic operators. In N. Jones, editor, *Semantics-Directed Compiler Generation*, 211–258, Berlin; LNCS 94, Springer-Verlag, 1980.

[Rey81] J. Reynolds, The essence of Algol. In de Bakker and van Vliet, editors, *Algorithmic Languages*, 345–372, North Holland, 1981.

[Rey89a] J. Reynolds, Preliminary design of the programming language Forsythe. Carnegie Mellon University technical report CMU-CS-88-159, June 1988.

[Rey89b] J. C. Reynolds, Syntactic control of interference, part II. In *International Colloquium on Automata, Languages, and Programming*, 1989.

[Rey91] J. Reynolds, The coherrence of languages with intersection types. In *International Conference on Theoretical Aspects of Computer Software*, Sendai, Japan, LNCS, Springer Verlag, September 1991.

[Spi90] M. Spivey, A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, June 1990.

[SRI91] V. Swarup, U. S. Reddy, and E. Ireland, Assignments for applicative languages. In *Conference on Functional Programming Languages and Computer Architecture*, Cambridge, Massachusetts; LNCS 523, Springer Verlag, August 1991.

[SS76] G. L. Steele, Jr. and G. Sussman, Lambda, the ultimate imperative. MIT, AI Memo 353, March 1976.

[Tur90] D. A. Turner, An overview of Miranda. In D. A. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.

[Wad85] P. Wadler, How to replace failure by a list of successes. *Conference on Functional Programming Languages and Computer Architecture*, Nancy, France; LNCS 201, Springer-Verlag, September 1985.

[Wad90] P. Wadler, Comprehending monads. In *Conference on Lisp and Functional Programming*, Nice, France; ACM, June 1990.

[Wad91] Is there a use for linear logic? *Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, New Haven, Connecticut; ACM, June 1991.