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.

To be presented as an invited talk at *19'th Annual Symposium on Principles of Programming Languages*, Santa Fe, 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