# Systematic Design of Monads

John Hughes & Magnus Carlsson

January 9, 1996

**Abstract**

Many useful monads can be designed in a systematic way, by successively adding facilities to a trivial monad. The capabilities that can be added in this way include state, exceptions, backtracking, and output. Here we give a brief description of the trivial monad, each kind of extension, and sketches of some interesting operations that each monad supports.

A more detailed treatment of the art of combining monads is found in [LHJ95], which also uses Gofer to give a more elegant presentation by the use of constructor classes.

## 1 The Identity Monad

```
type I a = a
unit v = v
x `bind` f = f x
```

The identity monad represents computations as the values they deliver. It can support no interesting operations, but is the starting point to which other capabilities can be added.

## 2 Adding Capabilities

Assume that we already have a monad `M`, with operations `unitM` and `bindM`. We can then define new monads extending the capabilities of `M`. This is done by defining a new `bind`-operation, in terms of `bindM`, and a function `liftNew`,

```
liftNew :: M a -> MNew a
```

which turns operations in the underlying monad into the new monad. Not all monad operations can be lifted in this simple way, but most of them. In particular, the new `unit` can be expressed as

```
unitNew a :: MNew a
unitNew a = liftNew (unitM a)
```

## 2.1 State

```
type MSt s a = s -> M (a,s)
x `bindSt` f = \s -> x s `bindM` \(v,s') -> f v s'

liftSt :: M a -> MSt s a
liftSt m = \s -> m `bindM` \v -> unitM (v,s)
```

Useful operations include

```
fetchSt :: MSt s s
fetchSt = \s -> unitM (s,s)

storeSt :: s -> MSt s ()
storeSt s = \_ -> unitM ((),s)
```

One useful application of state monads is to manage input. Let the state be the list of input tokens remaining to be read. Then we can read a token with

```
get :: MSt [t] t
get = fetchSt `bindSt` \(t:ts) ->
      storeSt ts `bindSt` \_ ->
      liftSt (unitM t)
```

## 2.2 State Reader

The 'state reader' monad manages a state that is never updated — perhaps better thought of as a hidden parameter. You might use it if you want access to, say, the list of command line arguments, without passing it explicitly as a parameter throughout your program. It can also be used for recording variable environments which may be extended for subcomputations by means of `extendSR`.

```
type MSR s a = s -> M a
x `bindSR` f = \s -> x s `bindM` \v -> f v s

liftSR :: M a -> MSR s a
liftSR m = \s -> m

fetchSR :: MSR s s
fetchSR = unitM

extendSR :: s -> MSR s a -> MSR s a
extendSR s m = \_ -> m s
```

## 2.3 Monoid

Given any associative operator

```
(<+>) :: t -> t -> t
```

with unit element `z`, we can define a monad by

```
type MMon t a = M (a,t)
x `bindMon` f = x `bindM` \~(v,a) ->
                f v `bindM` \~(w,b) ->
                unitM (w,a<+>b)

liftMon :: M a -> MMon t a
liftMon m = m `bindM` \v -> unitM (v,z)

putMon :: t -> MMon ()
putMon t = unitM ((),t)
```

The ~ symbols on the pair patterns make pattern matching lazy.

### 2.3.1 Output

An important application is when the type `t` is a list of outputs produced. Here the operator `<+>` is list concatenation (++), and its unit is the empty list. This application motivates the lazy pattern matching in `bindMon`: it's important that the output is produced lazily, in particular that the output from the first sub-computation can be produced before the second sub-computation is started. Thanks to the second ~, the result of `f v` isn't required until either we try to evaluate the result `w`, or the output `b` from `f v`.

## 2.4 Exceptions

```
type MEx e a = M (Error e a)
data Error e a = Yes a | No e

x `bindEx` f = x `bindM` \y -> case y of
             Yes v -> f v
             No e -> unitM (No e)


liftEx m = m `bindM` \v -> unitM (Yes v)
unitEx v = unitM (Yes v)
```

Operations to raise and handle exceptions can be defined as shown:

```
raiseEx :: e -> MEx e a
raiseEx e = unitM (No e)

handleEx :: MEx e' a -> (e' -> MEx e a) -> MEx e a
x `handleEx` h = x `bindM` \y -> case y of
               Yes v -> unitM (Yes v)
               No e -> h e
```

Of course the exception value `e` can be omitted if not needed.

It can make a big difference if the exception monad is added on top of `M`, or if `M` is added on top of the exception monad. Try to build a state monad on top of an exception monad, and vice versa. What is the difference?

### 2.5 Backtracking

Backtracking can be added to any monad in a similar way:

```
type MBt a = M [a]
x `bindBt` f = x `bindM` \vs ->
             foldr orelseBt failBt (map f vs)

liftBt :: M a -> MBt a
liftBt m = m `bindM` \v -> unitM [v]

infixl 5 `apM`
apM :: M (a -> b) -> M a -> M b
fm `apM` fx = fm `bindM` \f ->
              fx `bindM` \x ->
              unitM (f x)

orelseBt :: MBt a -> MBt a -> MBt a
x `orelseBt` y = unitM (++) `apM` x `apM` y

failBt :: MBt a
failBt = unitM []
```

But this monad lets us only perform *one* `M`-computation to produce a list of results. We might instead want to perform an `M`-computation for *each* new possible answer. To do this we need a recursive monad:

```
type MRBt a = M (Ans a)
data Ans a = Ans a (MRBt a) | NoAns
x `bindRBt` f = x `bindM` \y -> case y of
             NoAns -> failRBt
             Ans v x' -> f v `orelseRBt` (x' `bindRBt` f)
```

```
liftRBt m = m `bindM` \v -> unitM (Ans v failRBt)

failRBt = unitM NoAns
x `orelseRBt` y = x `bindM` \z -> case z of
               NoAns -> y
               Ans v x' -> unitM (Ans v (x' `orelseRBt` y))
```

## 2.6 Continuation I/O

If we want to build monads on top of a continuation based programming paradigm, such as stream processors or continuation based I/O in Haskell, we need to build a monad around the continuation based operations. The monad type reflects that every computation takes a *continuation* as a parameter, and applies it to the value to get a final result:

```
type MCio r a = (a -> r) -> r
unitCio v = \k -> k v
x `bindCio` f = \k -> x (\v -> f v k)
```

Typically, in continuation based programming, one doesn't have a `bind`- operation, but instead two different kinds of operations. One is used for doing side-effects, and these have the type

```
r -> r
```

The other kind of operation passes information (of type `i`) to the continuation, and the type of these is

```
(i -> r) -> r
```

We need to lift these two kinds of operations into monadic operations:

```
liftc :: (r -> r) -> MCio r ()
liftc d = \k -> d (k ())

lifti :: ((a -> r) -> r) -> MCio r a
lifti d = d
```

### 2.6.1 Examples

Here are examples of the two kinds of operations, from stream processors and dialogue I/O:

```
putCio :: o -> MCio (SP i o) ()
putCio o = liftc (putSP o)
```

```
getCio :: MCio (SP i a) i
getCio = lifti getSP

writeFileCio :: String -> String -> MCio Dialogue ()
writeFileCio file content = liftc (writeFile file content exit)

readFileCio :: String -> MCio Dialogue String
readFileCio f = lifti (readFile f exit)
```

### 2.6.2 Manipulating the continuation

A continuation monad also let us model control operations similar to *goto* in imperative languages. To do this, we need an operation to catch the continuation and return it as a value, and an operation to jump to a continuation (see the example *Devils and Angels* for an application of these operations):

```
callccCio :: ((a -> r) -> MCio r a) -> MCio r a
callccCio f = \k -> f k k

jumpCio :: (a -> r) -> a -> MCio r b
jumpCio k v = \_ -> k v
```

## 2.7 Continuation monad, continued

The operations `callcc` and `jump` could be useful to add to any monad. The continuation monad can in fact be added to any monad, by letting the result type be a monad value. In addition to the above operations, we only need the lift operation:

```
liftCio :: M a -> MCio (M r) a
liftCio m = \k -> m `bindM` \v -> k v
```

# 3 An Example: Dialogue I/O, State and Exceptions

Suppose we want to do dialogue I/O in our monad, as well as manipulate state and handle exceptions on top of that. Combining the types given above, we get the type

```
type MSt d s a = s -> MCio d (a,s)
type MEx e d s a = MSt d s (Error e a)
type MyMonad a = MEx MyError Dialogue MyState a
type MyError = ...
type MyState = ...
```

If we want, we can rename the operations in the topmost monad if we don't want to reveal the structure of it:

```
bind :: MyMonad a -> (a -> MyMonad b) -> MyMonad b
bind = bindEx

raise :: MyError -> MyMonad a
raise = raiseEx

handle :: MyMonad a -> (MyError -> MyMonad a) -> MyMonad a
handle = handleEx
```

Operations from the lower monads need to be lifted:

```
unit :: a -> MyMonad a
unit = liftEx . liftSt . unitCio

fetch :: MyMonad MyState
fetch = liftEx fetchSt

store :: MyState -> MyMonad MyState
store s = liftEx (storeSt s)

dodia :: (Dialogue -> Dialogue) -> MyMonad ()
dodia = liftEx . liftSt . liftc

getdia :: ((a -> Dialogue) -> Dialogue) -> MyMonad a
getdia = liftEx . liftSt . lifti
```

How can we invoke a computation of type `MyMonad a`? By expanding the type, we get

```
type MyMonad a =
  MyState -> ((Error MyError a,MyState) -> Dialogue) -> Dialogue
```

which indicates a suitable way of invoking it on the toplevel in a program:

```
invoke :: MyMonad a -> MyState -> Dialogue
invoke m startstate = m startstate (\_ -> done)
```

Here, we ignore the final state, as well as the final (error-)value.

# References


[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995.