The essence of functional programming

The paper by Philip Wadler

Martin Dybdal Troels Henriksen Ulrik Rasmussen

Datalogisk Institut, Københavns Universitet

27. april 2009

Agenda

What's a monad?

A triplet:

- A unary type constructor M
- A lifting function `unitM`[1] `:: a -> M a` that lifts a simple value into the monad. Creating a *monadic value*.
- A composition function `bindM`[2] `:: M a -> (a -> M b) -> M b` that applies a monadic function to a monadic value.

---

[1] This function is called `return` in Haskell and *val* in Andrzej Filinski nomenclature.

[2] »= in Haskell.

What's a monad?

A triplet:

- A unary type constructor M
- A lifting function `unitM`[1] `:: a -> M a` that lifts a simple value into the monad. Creating a *monadic value*.
- A composition function `bindM`[2] `:: M a -> (a -> M b) -> M b` that applies a monadic function to a monadic value.

Obeying three laws (discussed later):

1. `(unitM v) ‘bindM‘ f = f v`
2. `v ‘bindM‘ unitM = v`
3. `(m ‘bindM‘ f) ‘bindM‘ g = m ‘bindM‘ (\x -> (f x) ‘bindM‘ g)`

[1] This function is called `return` in Haskell and *val* in Andrzej Filinski nomenclature.

[2] »= in Haskell.

Running example: Interpreter

Datatypes:

```
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)]
```

Running example: Interpreter

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

Define new 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
```

We modify the interpreter to use this monad.

Modify interpreter

```
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)
```

`test term0` ⟶ "Success: 42"

`test (App (Con 1) (Con 2))` ⟶ "Error: should be function: 1"

Define monad based on the E monad

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

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)

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

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

- Special control flow implicit, not explicit.
- Easy to extend monadic program.
- Cleanly separates different parts of program logic.

The Output monad

- Type constructor:

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

- Lifting:

```
unitO :: a -> O a
unitO a = ("", a)
```

- Composition:

```
bindS :: O a -> (a -> O b) -> O b
m `bindS` k = let (r, a) = m
                  (s, b) = k a
              in (r ++ s, b)
```

Writing output:

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

The State monad

- Type constructor:

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

- Lifting:

```
unitS :: a -> S a
unitS a = \s -> (a, s)
```

- Composition:

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

Remark:

We are not actually specifying the type of value our state should hold.

Example: Counting reductions

The State monad

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

unitS a = \s -> (a, s)

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

Example: Counting reductions

Our reduction counter is represented by an integer:

```
type State = Integer
```

„Running“ the monad:

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

Updating and fetching the state:

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

The State monad

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

unitS a = \s -> (a, s)

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

Example: Counting reductions

Updating and fetching the state:

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

Doing the actual counting

```
apply (Fun k) a =
  tickS `bindS` (\() -> k a)

add (Num i) (Num j) =
  tickS `bindS` (\() -> unitS (Num (i+j)))
```

The State monad

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

unitS a = \s -> (a, s)

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

Braintwister: Backward state

Warning: This next example will hurt your brain!

We change the bind-operation from the State monad, so the State-information flows backward:

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

becomes

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

Computing the fibonacci sequence „backwards“

Backward state bind

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

Computing the fibonacci sequence „backwards“

Backward state bind

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

```
computeFibs = evalState []
      (fetchS `bindS` \fibs -> modify cumulativeSums
              `bindS` \_ -> updateS (1:fibs)
              `bindS` \_ -> unitS fibs)
```

Computing the fibonacci sequence „backwards“

Backward state bind

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

```
computeFibs = evalState []
      (fetchS `bindS` \fibs -> modify cumulativeSums
              `bindS` \_ -> updateS (1:fibs)
              `bindS` \_ -> unitS fibs)

updateS   =  \a s -> ((), a)
evalState start s = snd (s [])
```

Computing the fibonacci sequence „backwards“

Backward state bind

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

```
computeFibs = evalState []
      (fetchS `bindS` \fibs -> modify cumulativeSums
              `bindS` \_ -> updateS (1:fibs)
              `bindS` \_ -> unitS fibs)

updateS    =  \a s -> ((), a)
evalState start s = snd (s [])

>> take 15 computeFibs
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]
```

Found at http://lukepalmer.wordpress.com/2008/08/10/mindfuck-the-reverse-state-monad/

Non-deterministic choice

We modify the interpreter to deal with a non-deterministic language that returns a list of possible answers. We therefore need to define bind and return for lists:

```
type L a = [a]

unitL a       = [a]
m `bindL` k = concat (map k m)
zeroL         = []
l `plusL` m = l ++ m
```

Non-deterministic choice, cont'd

Extend the interpreted language:

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

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

Now, interpreting the expression

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

returns `"[2,4]"`.

Call-by-name

Modify the interpreter to call-by-name instead of call-by-value. Representations of functions should now be functions from computations to computations, and the environment should store computations instead of values:

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

type Environment = [(Name, M Value)]
```

Call-by-name, cont'd

Subtle modifications to the code is also required. When applying a value to a lambda expression, only the function is evaluated:

```
interp (App t u) e  = interp t e `bindM` (\f ->
                      apply f (interp u e))
```

When looking up variables in the environment, we no longer need to lift the values into the monad, as they are already computations:

```
lookup x []          = unitM Wrong
lookup x ((y,n):e) = if x==y then n else lookup x e
```

Call-by-name, cont'd

Example: If implemented for a non-deterministic language, interpreting the expression

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

Now returns `"[2,3,3,4]"`.