CSc 372

Comparative Programming Languages

20 : Haskell — Monads

Department of Computer Science
University of Arizona

collberg@gmail.com

The Monad

- Formally, a monad is defined as

```
class Monad m where
  (>>=)   :: m a -> (a -> m b) -> m b
  return  :: a -> m a
  (>>)    :: m a -> m b -> m b
  fail    :: String -> m a
```

- `return x` creates a "box" just containing the value `x`.
- `a >> b` takes a monad box `a`, throws away any computations it's done, and then returns the box `b`. What's important here is that the two actions are sequenced, one occurs before the other.

The Monad...

- Formally, a monad is defined as

```
class Monad m where
    (>>=)    :: m a -> (a -> m b) -> m b
    return   :: a -> m a
    (>>)     :: m a -> m b -> m b
    fail     :: String -> m a
```

- `a >>= f` is similar to `>>`, but the value that `a` constructs becomes the input to `f`, and the final result is whatever `f` returns.
- In fact, `>>` is defined in terms of `>>=`

```
m >> k = m >>= (\ _ -> k)
```

- `fail` also has a default definition:

```
fail s = error s
```

The do Notation

- The do notation that we saw earlier, is just syntactic sugar for sequencing using >>= and >>.
- These two definitions are identical:

```
test1a =
    do
        putStr "Welcome!\n"
        putStr "Please enter your name:\n "

test1b =
    putStr "Welcome!\n" >>
    putStr "Please enter your name:\n "
```

- Note how we're using >> since the value produced by the first `putStr` isn't needed (it's `()`).

The do Notation...

- Here, the value produced by the second line is needed by the third, so we use >>=:

```
test2a = do
    putStr "Please enter your name: "
    name <- getLine
    putStr ("Your name is '"++name++"'\n")
test2b =
    putStrLn "Please enter your name: " >>
    getLine >>= \name ->
    putStrLn ("Your name is "++name++"'\n")
```

- Note how in a `>>= f`, `f` is a function. `f` takes one argument, which is the value produced by a.
- Both >> and >>= sequence together actions in a particular order.

The IO Monad

- So, the `do` notation makes use of the `IO` monad:

```
class Monad IO where
    (>>=)    :: IO a -> (a -> IO b) -> IO b
    return   :: a -> IO a
    (>>)     :: IO a -> IO b -> IO b
    fail     :: String -> IO a
```

- Monads (and `do`) can be used in many other situations when we want to manipulate some sort of state.

The Maybe Monad

- One way of handling errors in Haskell is the `Maybe` datatype. It's a box that can either hold a value, or not:

```
data Maybe' a = Just' a | Nothing'
    deriving Show
```

- We can now add together values, with special cases when a value is missing:

```
add::Maybe' Int -> Maybe' Int -> Maybe' Int
add _ Nothing' = Nothing'
add Nothing' _ = Nothing'
add (Just' a) (Just' b) = Just' (a + b)
```

The Maybe Monad...

- Example:

```
> add Nothing' Nothing'
Nothing'
> add (Just' 5) Nothing'
Nothing'
> add (Just' 5) (Just' 6)
Just' 11
```

The Maybe Monad...

- We can turn `Maybe'` into a monad, and then use the `do` notation:

```
instance Monad Maybe' where
    (Just' x) >>= k = k x
    Nothing' >>= k  = Nothing'
    return x        = Just' x

test3a =
    do {x<-Just' 6;y<-Just' 7;return (x*y)}
test3b =
    do {x<-Just' 6;y<-Nothing';return (x*y)}
test3c =
    do {x<-Nothing';y<-Just' 7;return (x*y)}
test3d =
    do {x<-Nothing';y<-Nothing';return (x*y)}
```

Dealing with failure

- Assume that you have a sequence of actions you want to perform:

$$f \xrightarrow{a} g \xrightarrow{b} h \xrightarrow{c} \dots$$

  That is, f returns a which becomes input to g, and so on.
- Now what happens if one computation fails?

$$f \xrightarrow{a} g \xrightarrow{\text{fail}} h \xrightarrow{?} \dots$$

- Well, we probably want to propagate that failure all the way to the end:

$$f \xrightarrow{a} g \xrightarrow{\text{fail}} h \xrightarrow{\text{fail}} \dots$$

- We can use the `Maybe` monad to deal with failure in a sequence of computations.

The Maybe Monad — Example

- Say we want to look up someone on the government's noflylist, given the following databases:

```
name2ssn :: [(String, String)]
name2ssn = [("Alice", "612-88-8976"),
            ("Bob", "714-22-9852"),
            ("Charlies", "181-11-0987"),
            ("Dana", "091-08-1101")
           ]

ssn2passport :: [(String, String)]
ssn2passport = [("612-88-8976","123456987"),
                ("714-22-9852","222123908"),
                ("181-11-0987","789654120"),
                ("091-08-1101","890674123")
               ]
```

The Maybe Monad — Example...

```
noflylist :: [(String,Bool)]
noflylist = [("123456987",True),
             ("789654120",True)
            ]
```

The Maybe Monad — Example...

- Here's a lookup function:

```
lookup' :: Eq a => a -> [(a, b)] -> Maybe b
lookup' _ [] = Nothing
lookup' x ((a,b):xs)
              | x==a = Just b
              | otherwise = lookup' x xs
```

- Note that all the database may be missing entries, so we use the `Maybe` datatype to model lookup failure.

The Maybe Monad — Example...

- Here's how we chain together lookups in the three databases, without using monads:

```
mayfly a =
    case lookup' a name2ssn of
        Just b -> (
            case lookup' b ssn2passport of
                Just c -> (
                    case lookup' c noflylist of
                        Just d -> d
                        Nothing -> False
                    )
                Nothing -> False
            )
        Nothing -> False
```

The Maybe Monad — Example...

- And here's how we do it using monads:

```
mayfly' a =
    do
        b <- lookup' a name2ssn
        c <- lookup' b ssn2passport
        d <- lookup' c noflylist
        return d
```

- The `Maybe` monad propagates failures: it is defined so that if it encounters a `Nothing` it will just pass it on.

The State Monad

- Consider this implementation of a stack:

```
pop :: [Int] -> (Int, [Int])
pop (x:xs) = (x, xs)

push :: Int -> [Int] -> ((), [Int])
push x xs = ((), x:xs)

simulateStack s = let
    (_, s1) = push 3 s
    (x, s2) = pop s1
    (_, s3) = push (x * x) s2
    in pop s3

res = simulateStack [1, 2, 3]
```

- Note how `push` and `pop` return a pair *(value,new-stack)*.

The State Monad...

- Here's an alternative implementation, using the State monad:

```
pop' :: State [Int] Int
pop' = state (\(x:xs) -> (x, xs))

push':: Int -> State [Int] ()
push' x = state (\xs -> ((), x:xs))

simulateStack' = (push' 3) >>=
                 (\_ -> pop') >>=
                 (\x -> push' (x * x)) >>=
                 (\x -> pop')

res = runState simulateStack' [1, 2, 3]
```

- Note how `push` and `pop` return a pair *(value,new-stack)*.

Acknowledgments

- Brandon Simmons, *The State Monad: A Tutorial for the Confused?* http://brandon.si/code/the-state-monad-a-tutorial-for-the-confused
- Ryan Horn, http://brandon.si/code/the-state-monad-a-tutorial-for-the-confused
- *A physical analogy for monads*, http://monads.haskell.cz/html/analogy.html