An Introduction to Monads

Phillip Mates

March 6, 2012

Why Monads?

In a purely functional language:

- How do you encode actions with side-effects, such as reading and writing files?
- Is there an elegant way to pass around program state without explicitly threading it in and out of every function?
- How do you code up doubly nested for-loops?
- What about: Continuation passing style, Writing logs, Memory transactions...

What are Monads?

They're a very general abstraction idea that can be thought of as:

- containers that wrap values and are composable
- the inverse of pointers
- an abstraction for modeling sequential actions
- ...

Error handling with Maybe

```haskell
data Maybe a = Nothing
             | Just a

lookup :: a -> [(a, b)] -> Maybe b

animalFriends :: [(String, String)]
animalFriends = [ ("Pony", "Lion")
                , ("Lion", "Manticore")
                , ("Unicorn", "Lepricon") ]
```

```haskell
-- Does Pony's friend have a friend in animalMap?
animalFriendLookup :: [(String, String)] -> Maybe String
animalFriendLookup animalMap =
  case lookup "Pony" animalMap of
       Nothing -> Nothing
       Just ponyFriend ->
         case lookup ponyFriend animalMap of
              Nothing -> Nothing
              Just ponyFriendFriend ->
                case lookup ponyFriendFriend animalMap of
                     Nothing -> Nothing
                     Just friend -> Just friend
```

Monads are comprised of two functions

```haskell
-- Bind
(>>=) :: m a -> (a -> m b) -> m b

-- Inject value into a container
return :: a -> m a
```

Maybe Monad

```haskell
-- (>>=) :: m a -> (a -> m b) -> m b
Just x >>= k  =  k x
Nothing >>= _ =  Nothing

-- return :: a -> m a
return x      =  Just x
```

Using Maybe as a Monad

```
monadicFriendLookup :: [(String, String)] -> Maybe String
monadicFriendLookup animalMap =
  lookup "Pony" animalMap
  >>= (\ponyFriend -> lookup ponyFriend animalMap
  >>= (\pony2ndFriend -> lookup pony2ndFriend animalMap
  >>= (\friend -> Just friend)))
```

Using Maybe as a Monad

```
-- or even better:
sugaryFriendLookup :: [(String, String)] -> Maybe String
sugaryFriendLookup animalMap = do
  ponyFriend   <- lookup "Pony" animalMap
  ponyFriend'  <- lookup ponyFriend animalMap
  ponyFriend'' <- lookup ponyFriend' animalMap
  return friend
```

Threading program state

```haskell
type Sexpr = String

-- naive generation of unique symbol
transformStmt :: Sexpr -> Int -> (Sexpr, Int)
transformStmt expr counter = (newExpr, counter+1)
  where newExpr = "(define " ++ var ++ " " ++ expr ++ ")"
        var = "tmpVar" ++ (show counter)
```

Generalizing the threading of state

Let's drop

```
Int -> (Sexpr, Int)
```

from

```
transformStmt :: Sexpr -> Int -> (Sexpr, Int)
```

and replace it with a more general type constructor:

Generalizing the threading of state

Let's drop

```
Int -> (Sexpr, Int)
```

from

```
transformStmt :: Sexpr -> Int -> (Sexpr, Int)
```

and replace it with a more general type constructor:

```
newtype State s a = State {
      runState :: s -> (a, s)
    }
```

```
transformStmt :: Sexpr -> State Int Sexpr
```

State Monad

```haskell
-- return :: a -> State s a
return a = State (\s -> (a, s))

-- (>>=) :: State s a -> (a -> State s b) -> State s b
m >>= k = State (\s -> let (a, s') = runState m s
                      in runState (k a) s')
```

State Monad Example

What can be a Monad?

Type constructors with an arity of one, for instance:

```
-- this can't because it has arity 2:
ghci> :kind State
* -> * -> *

-- but these have arity 1:
ghci> :kind (State Int)
* -> *

ghci> :kind []
* -> *
```

Deriving the list monad

```
ghci> :type (>>=)
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b

ghci> :type map
map :: (a -> b) -> [a] -> [b]

ghci> :type flip map
flip map :: [a] -> (a -> b) -> [b]

ghci> :type concat
concat :: [[a]] -> [a]
```

The List monad models non-determinism

```
return x = [x]
xs >>= f = concat (map f xs)
```

The List monad models non-determinism

```
return x = [x]
xs >>= f = concat (map f xs)

-- monadic powerset
ghci> powerset = [1,2]
               >>= (\i -> [1..4]
               >>= (\j -> [(i, j)]))
[(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4)]
```

Desugaring do Blocks

```
do x <- foo        ===      foo >>= (\x -> bar)
   bar

do act1            ===      act1 >> act2
   act2
```

Further Topics & Reading

- Monad Transformers
- "Real World Haskell" by O'Sullivan, Stewart, and Goerzen
- Corresponding blog post:
  `quined.net/articles/monads.html`