Lab 09 - Haskell From Functors To Monads

CS 1XA3

March 12^{th}, 2018

The Either DataType

- The Either datatype is similar to the Maybe type, but provides a choice of another type rather than just Nothing

```
data Either a b = Left a | Right b
```

- Like Maybe, the Either type can be thought of like a container you have to pull values out of with pattern matching

```
eitherToDouble :: Either Int Double -> Double
eitherToDouble e = case e of
      Left i  -> fromIntegral i
      Right d -> d
```

Recap: Functors and Applcatives

- Recall the Functor typeclass that generalizes the Mapping operation

```haskell
instance Functor (Etiher a) where
  fmap f (Right x) = Right (f x)
  fmap f (left x)  = Left x
```

- An instance Either a of Functor can be defined simlar to how Maybe is, with an occurance of Left treated like Nothing

```haskell
instance Applicative (Either a) where
  pure val = Right val
  Right f <*> Right x = Right $ f x
  Right _ <*> Left x  = Left x
  Left x  <*> _       = Left x
```

Lists: We haven't Forgotten You

- You can define the list datatype with

```haskell
data [a] = a : [a] | []
-- not legal syntax
data List a = Cons a (List a) | Empty
-- legal but not as pretty
```

- A Functor instance for lists should already seem trivial to you, but what about Applicative?

```haskell
instance Applicative [] where
  pure x = [x]
  (f:fs) <*> xs = (fmap f xs) ++ (fs <*> xs)
  []     <*> _  = []
```

Monoids

- A Monoid is the typeclass of data structures that have associative binary operator with an identity

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

- Example: Lists are Monoids

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Monads: What the hell are they?

- Steps to learning Monads
  1. Get a Ph.D in Category Theory
  2. Throw it away
  3. Learn about Functors, Applicatives, and Monoids. Practice by programming practical implementations of them on different data types.
  4. Repeat Step 3 for Monads: notice the similarities / how Monads are a natural extension from the simpler concepts of Functors / Monoids
- Pro Tip: you can skip Steps 1 - 2

Monads: The bind operator

- The bind operator ($>>=$) takes a wrapped value and a function to apply like a Functor, however note the type of the function

```
class Applicative m => Monad m where
  (>>=) :: m a            -- wrapped value (like Just 3)
        -> (a -> m b) -- function (returns wrapped)
        -> mb  -- result of function application
```

- Consider the following function

```
half :: Integral a => a -> Maybe a
half x = let
   x' = div x 2
 in if even x' then Just x' else Nothing
```

Monads: A Maybe Instance

Note: the class definition of Monad is bound by Applicative which in turn is bound by Functor. If we want a Monad definition we need the others

```
instance Functor Maybe where
  fmap f (Just x) = Just $ f x
  fmap _ Nothing  = MyNothing

instance Applicative Maybe where
  pure x = Just x
  Just f <*> x = fmap f x
  _      <*> _ = Nothing

instance Monad Maybe where
  Nothing >>= _ = Nothing
  Just x  >>= f = f x
```

Using Monads for Chaining

- Why would we want Maybe to be a Monad?
- Consider the following code

```
oneEighth :: Integral a => a -> Maybe a
oneEighth x = Just x
        >>= half
        >>= half
        >>= half
```

- Imagine writing a corresponding function without (>>=); would be fairly tedious. Monads can be used to chain functions like this together, taking care of the wrapping / unwrapping automatically

Using Monads for Sequencing

- Consider the following conventional Haskell code

```haskell
someFunc1 = let
  x = a
  y = b
 in (x,y)
```

```haskell
someFunc2 = let
    y = b
    x = a
 in (x,y)
```

- Both of the above functions are evaluated like expressions and are the same, i.e order doesn't matter
- The order in a Monad however, does matter

```haskell
someIO = getLine
     >>= readFile
     >>= putStrLn
```

The Full Set of Monad Operations

- One need only define (>>=) when giving an instance of Monad, but the class provides more operations automatically

```haskell
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  -- useful for Monads with side-effects
  return :: a -> m a
  -- same as pure in Applicative
  fail :: String -> m a
  -- at any point, a chain can fail
```

The IO DataType

- First, keep in mind alot of properties of IO are unique to IO, not a general Monad thing (common cause of misconceptions)

- The definition of the IO type is a lower level system based one, and is purposely hidden

```
data IO a = ... -- who knows, who cares
```

- Since we have no value constructors for the IO type, we can't pull values out of an IO wrapper like we do with Maybe, etc

The IO Monad

- The fact that we can't pull values out of IO isn't an unfortunate accident. IO operations contain side-effects, we want to seperate them from the rest of the code
- For this reason, IO values can only be accesed through Functors / Monads.
- A colorful IO example

```
someIO :: IO ()
someIO = return "filepath.txt" >>= readFile
         >>= putStrLn >> putStrLn "EndFile"
```

The Do Syntax

- The do syntax provides us with a "pretty" way of writing Monad sequences
- Every chain of $(>>=), (>>)$ operations has a corresponding do syntax and vice versa

```
someIO = return "filepath.txt"
          >>= readFile
          >>= putStrLn
          >> putStrLn "EndFile"
-- same as
someIO = do f <- readFile "filepath.txt"
            putStrLn f
            putStrLn "EndFile"
```

Challenge

- Create a Monad instance for

```haskell
data MyEither a b = MyLeft a | MyRight b
```

- Create a Monoid and Monad instance for

```haskell
data List a = Cons a (List a) | Empty
```