CS4012 Topics in Functional Programming

Glenn Strong

Maybe Monad

`IO` is not the only monad.

Here is a datatype that might be familiar to you:

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

Using `Maybe`

A typical use of the `Maybe` type could involve some tree lookups (which can fail):

```haskell
f tree = case (findEntry "foo" tree) of
           Nothing -> Nothing
           Just x  -> case (findEntry "bar" tree) of
                        Nothing -> Nothing
                        Just y -> Just (x,y)
```

What's that?

Lots of "plumbing"?

Using `Maybe` monadically

We can rewrite this, because `Maybe` is a monad:

```haskell
f tree = do
           x <- findEntry "foo" tree
           y <- findEntry "bar" tree
           return (x,y)
```

Using `Maybe` monadically

What do the "unit" and "bind" operations look like?

```haskell
return x = Just x

Nothing  >>= k = Nothing
(Just x) >>= k = k x
```

Implementing Monad instances

The implication here is that we are using the Haskell "class" mechanism to overload `(>>=)` and `return`

The `do` notation is then desugared to chain the various "actions" together.

Up until GHC 7.10 (March 2015) if you wanted to implement your own `Monad` instance you would instantiate this class:

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

We will look at the post 7.10 world shortly (what we see here will carry over, don't worry!)

For example:

```haskell
instance Monad (Maybe a) where
  (Just x) >>= k = k x
  Nothing  >>= _ = Nothing

  return = Just

  fail _ = Nothing
```

Monad laws

All monad implementations are expected to satisfy three identities, called the *monad laws*.

- Left identity: `return a >>= f = f a`
- Right identity: `m >>= return = m`
- Associativity: `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

Not enforced by the compiler.

Fairly common-sense, once you think about them.

Example: The State Monad

Let's look at a slightly more substantial example of monadic programming.

In principle this is similar to the way we described the IO monad passing the notional "World" value about, except that we want to provide controlled access.

First we need a data type to be the instance of `Monad`. The standard Haskell state representation used for this is a type that looks like:

```
newtype State s a = State s -> (a,s)
```

- The type `s` represents the type of the state we are carrying around, and
- the type `a` is result type of our stateful computations.

So, for example:

```
f :: State String Int
```

is some computation that has maintains a state of type `String` and computes a value of type `Int`

As a matter of convenience we usually represent this as a *record* instead:

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

You'll recall from last year that a Haskell record like this produces a function:

```
runState :: State s a -> s -> (a,s)
```

We can make this an instance of `Monad` with a few declarations

First,

```
instance Monad (State s) where
    return a = State (\s -> (a,s))
```

Combining two stateful actions looks like this:

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

Our state monad is not really useful yet, because there are no operations to access the state. So we should declare some ways to manipulate the state:

```
get :: State s s
get = State $ \s -> (s,s)

put :: s -> State s ()
put s = State $ \_ -> ((),s)
```

As an exercise to see if you understand how `State` works, try implementing:

```haskell
modify :: (s -> s) -> State s ()
```

which adjusts the state using a function.

State example: threading random numbers

In `System.Random` there is a standard random number generator.

```haskell
getStdGen :: IO StdGen
randomR   :: (Random a, RandomGen g) => (a, a) -> g -> (a, g)
random    :: (Random a, RandomGen g) =>           g -> (a, g)
```

A somewhat contrived example:

```haskell
data Val = Val Int Bool Char Int
           deriving Show

makeRandomValue :: StdGen -> (Val, StdGen)
makeRandomValue g = let (n,g1) = randomR (1,100) g
                        (b,g2) = random g1
                        (c,g3) = randomR ('a', 'z') g2
                        (m,g4) = randomR (-n, n) g3
                     in (Val n b c m, g4 )
```

Bit ugly. Let's clean it up.

I will create a set of State monad based wrappers for the Generator actions

```haskell
randomRState :: (Random a) => (a,a) -> State StdGen a
randomRState bounds = do g <- get
                         (x,g') <- return $ randomR bounds g
                         put g'
                         return x
```

Actually, this is a bit long winded:

```haskell
randomRState bounds = State (randomR bounds)
```

is sufficient!

So we wrap the generators in the state:

```haskell
randomState :: (RandomGen g, Random a) => State g a
randomState = State random

randomRState :: (RandomGen g, Random a) => (a,a) -> State g a
randomRState bounds = State (randomR bounds)
```

And now we can generate the values:

```haskell
MkRandomValueST :: RandomGen g => State g Val
mkRandomValueST = do
```

```
  n <- randomRState (1,100)
  b <- randomState
  c <- randomRState ('a','z')
  m <- randomRState (-n,n)
  return $ Val n b c m

main = do
         g <- getStdGen
         print $ fst $ makeRandomValue g
         print $ fst $ runState mkRandomValueST g
```

Other "standard" monads

It's not just `Maybe` from the Prelude that can get this treatment

```
instance Monad [] where
  return a = [a]
  lst >>= f = concat (map f lst)
```

What does it mean to say that lists form a monad? It represents the type of computations that may return 0, 1, or more results.

More specifically, it combines actions by applying the operations to all possible values.

```
cart xs ys = do
                x <- xs
                y <- ys
                return (x,y)
```

What will `cart [1,2,3] [97,98,99]` do?

```
Prelude> cart [1,2,3] [97,98,99]
[(1,97),(1,98),(1,99),(2,97),(2,98),(2,99), (3,97),(3,98),(3,99)]
```

Cartesian product! All combinations of the two lists.

Applicative Functors

Everything I told you was true. At one time.

Starting in GHC 7.10 some extra abstraction was introduced. We might like to ignore it, but we can't. The `Monad` class was refactored into three pieces: `Functor`, `Applicative`, and `Monad`.

Functors

A `Functor` is something that can apply functions to "wrapped" values.

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

Taking `Maybe` as our example, it can be made an instance of `Functor`

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just a) = Just (f a)
```

`Data.Functor` also contains this synonym which is sometimes used when infix expressions read more cleanly:

```
(<$>) = fmap
```

Functors are expected to obey two equational laws. Nothing in the compiler enforces this, but many library functions that accept `Functor` values expect them to be true!

```
fmap id      = id
fmap (g . h) = fmap g . fmap h
```

Applicatives

An `Applicative` is a `Functor`, with a bit more going on.

We might like to allow an `fmap`-like operation with more arguments.

For example, if we had

```
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
```

then this would do something useful:

```
fmap2 (+) (Just 1) (Just 2)
```

Why stop at `fmap2`, why not have `fmap3` and `fmap4` and so on?

This isn't how we like to solve problems in the functional world!

Applicative

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

we can now write `fmap2 g x y = pure g <*> x <*> y`

Haskell 7.10 Monads

The actual Monad class in modern Haskell is:

```haskell
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b

  (>>) :: m a -> m b -> m b
  m >> k = m >>= \_ -> k

  return :: a -> m a
  return = pure

  fail :: String -> m a
  fail s = errorWithoutStackTrace s
```

fmap Vs liftM

It's good to know the following. Given that:

```haskell
fmap :: Functor f => (a -> b) -> f a -> f b
```

There is a utility function in the Prelude called `liftM` which "lifts" a function into a monad:

```haskell
liftM   :: (Monad f) => (a -> b) -> f a -> f b
liftM f m1              = do { x1 <- m1; return (f x1) }
```

in many (most) cases where you are creating a Monad instance `fmap = liftM` for your monad.

So the complete (modern) set of instances look like this…

```haskell
instance Functor (State s) where
  fmap = liftM

instance Applicative (State s) where
  pure a = State (\s -> (a,s))

instance Monad (State s) where
  m >>= k = State (\s -> let (a,s') = runState m s
                         in runState (k a) s')
```