Functional Programming (H) – Lecture 9 – Mon 28 Oct

More Monads

Jeremy.Singer@glasgow.ac.uk

Today we are going on a deep dive with the monad typeclass. Our *learning objectives* are:

- Gain increasing familiarity with monad concepts and functions
- Recognize a number of common library monads
- Become confident at using appropriate monads in your Haskell programs

Introduction

So, let's revisit the `Monad` typeclass. Recall it has two characteristic methods, i.e. `return` and bind `(>>=)`. The `return` function "puts something into" the monad — i.e. boxes it up in the structure. The `(>>=)` function injects a monad value into arbitrary function `f` as a parameter, with the added constraint that `f` returns a value in the same monad — i.e. `(>>=)` wangles the types so functions can take monadic values as parameters.

Maybe Monad

We will motivate these (somewhat abstract) conepts with the now-familiar `Maybe` datatype, which is monadic.

Consider a function `allButLast :: [Char] -> Maybe [Char]`. This function will return the first (n-1) characters of a string with (n) characters, wrapped up as a `Just` value. If the input string is empty, then the function evaluates to `Nothing`.

```
allButLast [] = Nothing
allButLast [x] = Just []
allButLast (x:xs) =
  let rest = allButLast xs
  in case rest of
    Nothing -> Just [x]
    (Just xs') -> Just (x:xs')
```

Now let's think about what happens when we repeatedly apply this function to a string input value … eventually we will run out of characters so we will bottom out at the `Nothing` value:

```
take 10 $ iterate (\x -> x >>= allButLast) (Just "abcdef")
```

Can you see what happens when we try to bind a function with a `Nothing` value? The result (thanks to the definition of `(>>=)`) is always `Nothing`.

Exercise for the reader: Rework the above example with the `Either` datatype.

Identity Monad

Next we will look at the `Identity` monad, which might seem a little pointless but it's a useful base case.

We need to `import Control.Monad.Identity`, which might require some library configuration in ghci … perhaps `:set -package mtl` on the interactive prompt.

We can put something into the `Identity` monad with the `return` function:

```
(return 42) :: Identity Int
```

and we can apply a function to `Identity` monadic values with `(>>=)`:

```
((return 42) :: Identity Int) >>= (\x -> return (x+1))
```

We extract a value out of an `Identity` computation by *running* the monad, using the `runIdentity :: Identity a -> a` function.

```
runIdentity $ ((return 42) :: Identity Int) >>= (\x -> return (x+1))
```

I suppose the equivalent case of *running* the IO monad is the invocation of the `main` function at top-level in Haskell.

Exercise for the reader: Can you define the `return` and `(>>=)` functions for `Identity`. They are completely trivial!

List monad

We can think of using a monad as being like putting a value into a structure, *elevating* or *lifting* the value into the monad. What structures do we already know in Haskell? Well, the list is probably the simplest … and guess what? Yes, list is a monad.

List `return` simply puts value into a singleton list (i.e. syntactically, just put square brackets around it).

```
listreturn :: a -> [a]
listreturn x = [x]
```

List bind takes each element out of the list, applies a function to that element, giving a list result, then concatenates all the results into a single, new results list. The key point (which confuses some people) is that all the results are glued together into a single list, rather than being separate sublists … like they might be with a `map` function call.

```
listbind :: [a] -> (a->[b]) -> [b]
listbind xs f = concat $ map f xs
```

Here is a simple example of list bind:

```
ghci> let f x = [x,x]
ghci> f 2
[2,2]
```

```
ghci> [1,2,3] >>= f
[1,1,2,2,3,3]
```

Here is another example:

```
ghci> let f = \x -> (show x) ++ " mississippi... "
ghci> [1,2,3] >>= f
"1 mississippi... 2 mississippi... 3 mississippi... "
```

So we see that list bind is reminiscent of a `join` in Python, or a `flatMap` in Java/Scala.

Reader, Writer and State Monads

Warning: now we are accelerating to rocket speed for the second half of this lecture.

So far we have looked at fairly straightforward monads, many of which you have seen before. We are going to look at three useful library Monads. For each one, we will look at their API and a typical use case. These examples *might* be helpful for your coursework, coming up in a few weeks.

These three monads involve an *environment*, which we pass around, threading it from function context to function context. Typical use cases for each monad are as follows:

- `Reader` - shared environmental configuration
- `Writer` - logging operations
- `State` - computing a large value recursively

Reader

The `Reader` monad is also known as the *environment* monad. It's useful for reading fixed values from a shared state environment, and for passing this shared state between multiple function calls in a sequence.

Here is a trivial example:

```
import Control.Monad.Reader
--  ^^^ may need some GHCi hackery ...

hi = do
  name <- ask
  return ("hello " ++ name)

bye = do
  name <- ask
  return ("goodbye " ++ name)

conversation = do
  start <- hi
  end <- bye
  return (start ++ " ... " ++ end)
```

```
main = do
  putStrLn $ runReader conversation "jeremy"
```

The Reader structure has two parameters, one of which is the environment (here a string) and the other is the result of the computation. So the Reader datatype is parameterized on two type variables: `Reader environment result`

The `runReader :: Reader r a -> r -> a` function takes a Reader expression to execute, along with initial environment, and extracts the final value from it.

Writer

The `Writer` monad builds up a growing sequence of state (think about logging a sequence of operations). Officially, this state is a *monoid*, but we will only consider `String` values for today.

The `tell` function appends something to the log.

```
import Control.Monad.Writer

addOne :: Int -> Writer String Int
addOne x = do
  tell ("incremented " ++ (show x) ++ ",")
  return (x+1)

double :: Int -> Writer String Int
double x = do
  tell ("doubled " ++ (show x) ++ ",")
  return (x*2)

compute =
  tell ("starting with 0,") >> addOne 0 >>= double >>= double >>= addOne

main = do
  print $ runWriter (compute)
```

State

Finally, the `State` monad. This is very similar to the `Reader` and `Writer` monads, in that it allows us to pass around a shared state, but also to update it.

We can `get` the current state, or `put` a new value into the state.

Let's do this with the fibonacci function ...

```
import Control.Monad.State

fibState :: State (Integer, Integer, Integer) Integer
fibState = do
  (x1,x2,n) <- get
```

```
  if n == 0
  then return x1
  else do
    put (x2, x1 + x2, n - 1)
    fibState

main = do
  print $ runState fibState (0,1,100)
```

Homework and Further Reading

- Can you redefine another interesting recursive function using the `State` monad? Why is it more efficient?
- Read about these utility monads at https://learnyouahaskell.com/for-a-few-monads-more or https://www.adit.io/posts/2013-06-10-three-useful-monads.html
- Here is a more complex piece about the `State` monad: https://hacklewayne.com/state-monad-a-bit-of-currying-goes-a-long-way
- Did Simon warn you about monad analogies? Look up the *monads as burritos* or *monads as spacesuits* explanations online. Do these make sense?

Finally, don't worry if we went way too fast today. We will consolidate this material on Thursday and in the lab on Friday morning.