Functional Programming for Logicians - Lecture 3

Folding, Functors, Monads

Malvin Gattinger

6 June 2018

```
module L3 where
```

(Slides partly copied from Jan van Eijck)

Outline

- Folding
- Hello World
- Category Theory in one slide
- Functors and Applicatives
- ~~Monads~~ Warm Fuzzy Things
- Input and Output — the IO `Monad`

Folding

Spot the pattern!

```haskell
mySum :: [Integer] -> Integer
mySum []     = 0
mySum (x:xs) = x + mySum xs

myAnd :: [Bool] -> Bool
myAnd []     = True
myAnd (x:xs) = x && myAnd xs

myMap :: (a -> b) -> [a] -> [b]
myMap f []     = []
myMap f (x:xs) = f x : myMap f xs
```

foldl

```
mySum :: [Integer] -> Integer
mySum xs = foldl (+) 0 xs

myAnd :: [Bool] -> Bool
myAnd xs = foldl (&&) True xs

myMap :: (a -> b) -> [a] -> [b]
myMap f xs = foldl (\x -> (f x :)) [] xs
```

Folding left and right

```
λ> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
λ> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Exercise: Which of `foldl` and `foldr` can work on infinite lists? Why?

See also: Haskell wiki: https://wiki.haskell.org/Foldr_Foldl_Foldl'

Hello World

Obligatory Meme

Hello World

IO (Input/Output) is interaction with the outside world. IO in Haskell is different from IO in imperative or object-oriented languages, because the functional paradigm isolates the purely functional kernel of the language from the outside world. Hence `Hello World` is not the simplest possible program to write in Haskell. But it is also not *that* difficult:

```
λ> putStrLn "Hello World"
Hello World
```

This outputs a string to the screen. A more elaborate version would first ask for user input, which means that interaction with the outside world takes place. Let's see that in Haskell . . .

Hello World 2.0

```haskell
helloWorld :: IO ()
helloWorld = do putStrLn "What is your name?"
                x <- getLine
                putStrLn ("Hello " ++ x ++ "!")
```

Such interaction is called a *side effect*. Pure functions do not have side effects; they just compute values, and for this they do not use information from the outside world. This purity is a very good thing, for it allows us to reason about functional computation in a mathematical way.

Input-Output involves interaction with the outside world, hence side effects. The Haskell method of dealing with side effects, and isolating them from the pure functional kernel of the language is by putting them in a wrapper. Such wrappers like `IO` are called `Monads`.

Theory first!

The concept of monads is borrowed from category theory. Monadic programming *can* be understood without tracing this connection.

In the same way, IO in Haskell can be understood without understanding monads. But as you are Master of Logic students, a bit of theory is appropriate.

The Category of Types

Category Theory Basics

- Objects
- Arrows
  - identity
  - composition
- Functors:
  - maps objects to objects and arrows to arrows
  - maps the identity to the identity
  - commutes with arrow composition

(Recommended talk: *Category Theory for the Working Hacker* by Philip Wadler
https://youtu.be/V10hzjgoklA)

The Category of Types

- Objects: Types `a`
- Arrows: Function Types `a -> b`

Functors

Functors: `fmap`

Just another type class, like `Eq` or `Show`.

```
λ> :i Functor
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  {-# MINIMAL fmap #-}
```

As we learned in category theory, a functor should fulfill two conditions:

1. Functors must preserve identity morphisms:

   ```
   fmap id == id
   ```

2. Functors preserve composition of morphisms:

   ```
   fmap (f . g)  ==  fmap f . fmap g
   ```

Recall: Maybe

Recall that `Maybe` is predefined like this:

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

Example usage:

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

Example: `instance Functor Maybe`

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

Maybe is a functor:

- on objects: `a` is mapped to `Maybe a`
- on arrows: `a -> b` is mapped to `Maybe a -> Maybe b`

```haskell
instance  Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap    _           Nothing  = Nothing
  fmap    f           (Just a) = Just (f a)
```

Example: instance Functor Maybe

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

Maybe is a functor:

- on objects: `a` is mapped to `Maybe a`
- on arrows: `a -> b` is mapped to `Maybe a -> Maybe b`

```
instance  Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap    _           Nothing  = Nothing
  fmap    f           (Just a) = Just (f a)
```

Does it fulfill both conditions?

- `fmap id = id`
- `fmap (f . g)  ==  fmap f . fmap g`

Functor is just a type class

Just like Eq, Ord, Show and Functor is a type class, which means that certain functions are predefined for it. An important difference however is that Functor is a type class which does not apply to concrete types $*$ but to *an abstract type* $* \to *$. For example, instance Eq String where ... makes the equality $==$ available on Strings.

```
λ> :k Eq
Eq :: * -> Constraint
```

Functor is just a type class

Just like `Eq`, `Ord`, `Show` and `Functor` is a type class, which means that certain functions are predefined for it. An important difference however is that `Functor` is a type class which does not apply to concrete types $*$ but to *an abstract type* $* \to *$. For example, `instance Eq String where ...` makes the equality $==$ available on Strings.

```
λ> :k Eq
Eq :: * -> Constraint
```

In contrast, `instance Functor Maybe where ...` does not give us functions working on things of type `Maybe` because nothing is of type `Maybe`. Things can be of Type `Maybe Int`, `Maybe String`, etc.

```
λ> :k Functor
Functor :: (* -> *) -> Constraint
```

Functor is just a type class

Just like `Eq`, `Ord`, `Show` and `Functor` is a type class, which means that certain functions are predefined for it. An important difference however is that `Functor` is a type class which does not apply to concrete types $*$ but to *an abstract type* $* \to *$. For example, `instance Eq String where ...` makes the equality $==$ available on Strings.

```
λ> :k Eq
Eq :: * -> Constraint
```

In contrast, `instance Functor Maybe where ...` does not give us functions working on things of type `Maybe` because nothing is of type `Maybe`. Things can be of Type `Maybe Int`, `Maybe String`, etc.

```
λ> :k Functor
Functor :: (* -> *) -> Constraint
```

See also: Kinds and some type-foo in LYHFGG

A functor is a *thing that can be mapped over*, and the function to do so is `fmap`.

```
λ> fmap succ (Just (3::Int))
Just 4
λ> fmap (map toUpper) (Just "hello")
Just "HELLO"
λ> fmap succ [1,5,100::Int]
[2,6,101]
```

The last example shows that `[ ]` is also a functor

Function Application inside a Functor

Suppose we are processing inside a functor, and we need to apply a *pure* function, say `toUpper` from `Data.Char`. Then this is how to do it:

```
λ> fmap (map toUpper) (Just "jan")
Just "JAN"
```

The function `fmap` turns the string operation `map toUpper` into a monadic function.

There is also an alias for infix notation:

```
λ> map toUpper <$> Just "malvin"
Just "MALVIN"
```

This shows that `fmap` can be used as the monadic version of `$`.

Applicatives

Applicative Functors

An `Applicative` is a functor with extra structure, but not yet a Monad. (See Applicative Functors.)

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

A functor with application provides a way to:

- embed pure expressions (with `pure :: a -> f a`)
- sequence computations and combine their results (with `(<*>) :: f (a -> b) -> f a -> f b`).

Applicative Laws

```
pure id <*> == id

pure (.) <*> u <*> v <*> w = u <*> (v <*> w)

pure f <*> pure x = pure (f x)

u <*> pure y = pure ($ y) <*> u
```

The second law is the compostion law. It says that if `<*>` is used for composition, then the composition is associative.

In the fourth law, note that `($ y)` is the function that supplies `y` as argument to another function.

Relation between `fmap` and `<*>`:

```
fmap f x = pure f <*> x
```

Example: instance Applicative Maybe

```
instance  Applicative Maybe where
  pure :: a -> Maybe a
  pure = Just
  (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
  (<*>)    Nothing              _          = Nothing
  (<*>)    (Just f)             Nothing    = Nothing
  (<*>)    (Just f)             (Just x) = Just (f x)
```

```
λ> Just succ <*> Just 3
Just 4
```

Example: `instance Functor/Applicative []`

Another `Functor` you already know is `[]` for lists!

- Objects: `a` is mapped to `[a]`
- Arrows: `a -> b` is mapped to `[a] -> [b]`

⇒ Exercises: How are the Functor and Applicative functions defined for lists?

- `fmap`
- `pure`
- `<*>`

Try to write down the definitions yourself and check that `fmap f xs = pure f <*> xs` holds.

Monads

Monads: >>=

Our biggest mistake: Using the scary term "monad" rather than "warm fuzzy thing". — *Simon Peyton-Jones*

```
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  {-# MINIMAL (>>=) #-}
```

The function `(>>=)` is called `bind`.

Example: instance Monad Maybe

Maybe is a Monad, and in this particular case, the bind function is defined as:

```
instance Monad Maybe where
  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  (>>=)    (Just x)   f                = f x
  (>>=)    Nothing    _                = Nothing
```

Example: instance Monad Maybe

Maybe is a Monad, and in this particular case, the bind function is defined as:

```
instance Monad Maybe where
  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  (>>=)    (Just x)   f                 = f x
  (>>=)    Nothing    _                 = Nothing
```

Examples of the use of (>>=):

```
λ> Just (3::Int) >>= \x -> Just (succ x)
Just 4
λ> Just (3::Int) >>= \x -> lookup x [(3,5),(7,9::Int)]
Just 5
λ> Just (1::Int) >>= \x -> lookup x [(3,5),(7,9::Int)]
Nothing
```

Example: Using Maybe for Exceptions

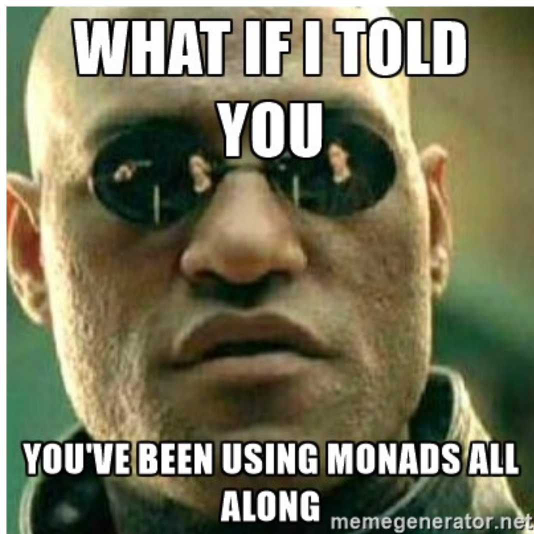
```
table :: [(Int,Int)]
table = map (\x -> (x,x^(3::Int))) [1..100]
```

We want to look up two numbers and add them.

```
process :: Int -> Int -> Maybe Int
process m n = lookup m table >>= \v -> lookup n table
                             >>= \w -> return (v+w)
```

```
λ> lookup 3 table
Just 27
λ> lookup 200 table
Nothing
λ> process 3 5
Just 152
```

What does `process 0 3` and `process 3 200` give? Why?


WHAT IF I TOLD YOU
YOU'VE BEEN USING MONADS ALL ALONG
memegenerator.net

Example: `instance Monad []`

Now let's look at the list container as a monad. List types are functors, and their mapping function `fmap` is `map`. List types are also monads. Putting a single thing in a list container is done by the function `\x -> [x]`, which can be abbreviated as `(:[])`. Hence the `return` for lists is `(:[])`.

```
λ> return 3 :: [Int]
[3]
```

Let's make a few variations on `x -> [x]` and see what we get:

```
λ> [1,2,3] >>= \x -> [x,x]
[1,1,2,2,3,3]
λ> [1,2,3] >>= \x -> [x,x,x]
[1,1,1,2,2,2,3,3,3]
λ> [1,2,3] >>= \x -> [x,x,x,x]
[1,1,1,1,2,2,2,2,3,3,3,3]
```

Keep Calm

Again, saying that something is a monad just means that it is of kind `* -> *` and in the type class `Monad`. Which means that >>= works on it.

The internet has so many Monad tutorials that you could probably spend the rest of this month reading them. Here are three:

- LYHGG: *A Fistful of Monads*
  http://learnyouahaskell.com/a-fistful-of-monads
- Wiki books: *Understanding Monads*
  https://en.wikibooks.org/wiki/Haskell/Understanding_monads
- Stephen Diehl: *Monads Made Difficult*
  http://www.stephendiehl.com/posts/monads.html

The Monad Laws

- `return a >>= k  =  k a`.
- `m >>= return  =  m`.
- `m >>= \x -> k x >>= h  =  (m >>= k) >>= h`.

The first law says that `Just 5 >>= Just` and `Just 5` are equivalent.

The second law says that `getLine >>= return` is equivalent to `getLine`.

The third law expresses the associativity of `>>=`:

```
getLine >>= \n -> putStrLn n >>= return
```

is equivalent to: `getLine >>= putStrLn >>= return`

which by the second Monad Law is in turn equivalent to

```
getLine >>= putStrLn
```

>> then

The function `(>>)` can be defined in terms of `(>>=)` as follows:

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

This function (also called *then* is a more primitive version of `(>>=)` (called *bind*): it discards the value passed to it by its first argument.

join

```
λ> :t join
join :: Monad m => m (m a) -> m a
```

This is predefined for every Monad like this:

```
join xss = xss >>= id
```

Thinking of `m` as a box, we see that `join` *flattens* a box in a box to a single box. The implementation for the case of `Maybe` is easy:

```
join :: Maybe (Maybe a) -> Maybe a
join (Just x) = x
join _        = Nothing
```

Examples:

```
λ> join (Just (Just 3))
Just 3
λ> join (Just Nothing)
Nothing
λ> join Nothing
Nothing
λ> join [[1],[2,3],[4::Int]]
[1,2,3,4]
```

The last result shows that `join` for lists is in fact the `concat` function.

Overview: Functor, Monad, Applicative

$$\text{Monads} \subseteq \text{Applicatives} \subseteq \text{Functors}$$

Functor:

- `fmap :: (a -> b) -> f a -> f b`

Applicative:

- `pure :: a -> f a`
- `(<*>) :: f (a -> b) -> f a -> f b`

Monad:

- `(>>=) :: m a -> (a -> m b) -> m b` (called *bind*)
- `return :: a -> m a`

10

Hello World, again

You can think of `IO a` as *actions* which give a result of type a.

```
helloWorld :: IO ()
helloWorld = do putStrLn "What is your name?"
                x <- getLine
                putStrLn ("Hello " ++ x ++ "!")
```

This is in fact the same as:

```
hello :: IO ()
hello = putStrLn "What is your name?"
          >> getLine
            >>= \name -> putStrLn ("Hello " ++ name)
```

⇒ Questions: What are the types of `getLine` and `putStrLn`? Why do we first use >> and then >>=?

IO in a single line in ghci

Note that what we did with Maybe also works with the IO monad:

```
λ> import Data.Char
λ> fmap (map toUpper) getLine
jan
"JAN"
```

(The line `jan` is entered by the user!)

Again, also with the infix `<$>` operator:

```
λ> map toUpper <$> getLine
malvin
"MALVIN"
```

Do Notation

First consider monad objects connected by `then` operators. Example:

```
λ> putStrLn "x" >> putStrLn "y" >> putStrLn 'z'
"x"
"y"
'z'
λ> do putStrLn "x" ; putStrLn "y" ; putStrLn 'z'
"x"
"y"
'z'
```

Think of the monad objects as a list of actions, and of the *do* notation of a way of presenting this list like a sequential program.

In a similar way we can translate `bind` to `do` notation:

```
λ> getLine >>= \ x -> putStrLn ("hello " ++ x)
jan
"hello jan"
λ> do x <- getLine; putStrLn ("hello " ++ x)
jan
"hello jan"
```

Do notation with <-

We can string more than two actions together:

```
greetings :: IO ()
greetings = do putStrLn "First name?"
               x <- getLine
               putStrLn "Second name?"
               y <- getLine
               putStrLn  ("Hello " ++ x ++ " " ++y)
```

Note that the semicolons are superfluous now.

You can think of `x <- y` as *doing the action y to get the value x*.

The function `greetigns` is equivalent to (or *syntactic sugar* for):

```
greetz :: IO ()
greetz = putStrLn "First name?" >>
         getLine >>= \ x ->
         putStrLn "Second name?" >>
         getLine >>= \ y ->
         putStrLn  ("Hello " ++ x ++ " " ++ y)
```

See also: Wiki books on do notation

No exercise session today, but ...

- you should practice!
- see website for research/project topics!
- further reading: Why IO Input Types Are Bad

See you tomorrow at 10:00.

Garfield tried learning Haskell