Functional Programming for Logicians - Lecture 3

Folding, Functors, Monads

Malvin Gattinger

14 January 2022

```haskell
module L3 where

import Control.Monad
```

(Based on material by Jan van Eijck)

Outline

- Folding
- Hello World
- Category Theory in one slide
- Functors and Applicatives
- ~~Monads~~ Warm Fuzzy Things
- Input and Output

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
--
```

foldl, point-free

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

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

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

Folding left and right

```
λ> :t foldl
foldl :: (b -> a -> b) -> b -> [a] -> b
λ> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Exercise: Does `foldl` or `foldr` work for infinite lists? Why?

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

Folding left and right

```
λ> :t foldl
foldl :: (b -> a -> b) -> b -> [a] -> b
λ> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Exercise: Does `foldl` or `foldr` work for infinite lists? Why?

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

Side note: The actual type of `folds` is more general, using the "Foldable" type class!

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.

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. It is *not* the same as this:

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

A more elaborate version would first ask for user input, but this means interaction with the outside world takes place!

Hello World 2.0

Pure functions do not have side effects: they just compute values, and for this they do not use information from the outside world. Purity is good: it allows us to reason about computation in a mathematical way.

Input-Output involves interaction with the outside world, hence side effects. Haskell isolates side effects from the "pure" language is by "wrapping" them. This wrapper is called the IO monad.

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

Theory first!

The concept of monads is borrowed from Category Theory.

IO in Haskell can be understood without understanding monads.

But as you are Master of Logic students, theory is appropriate.

The Category of Types

Category Theory Basics

- Objects
- Arrows
  - identity
  - composition

Category Theory Basics

- Objects
- Arrows
  - identity
  - composition
- Functors:
  - map objects to objects
  - map arrows to arrows

  such that:

  - map identity to the identity
  - commute with arrow composition

Category Theory Basics

- Objects
- Arrows
  - identity
  - composition
- Functors:
  - map objects to objects
  - map arrows to arrows

  such that:

  - map identity to the identity
  - commute with arrow composition

A closed category for talking about Haskell:

- Objects: Types, for example `a`
- Arrows: Function Types, for example `a -> b`

Functors

Functors: `fmap`

A type class, like `Eq` or `Show`.

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

Functors: `fmap`

A type class, like `Eq` or `Show`.

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

As 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
   ```

Functors: `fmap`

A type class, like `Eq` or `Show`.

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

As 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
   ```

It is our responsibility to define `fmap` in a way that fulfills this.
(Similar to how it was our job to make `(==)` useful.)

Recall: Maybe

Recall that `Maybe` is predefined like this:

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

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) = if k == k' then Just v
                                     else myLookup k xs
```

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) = if k == k' then Just v
                                   else myLookup k xs

λ> myLookup 2 [ (1,2), (3,4) ]
Nothing
λ> myLookup 3 [ (1,2), (3,4) ]
Just 4
```

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`

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`

But how?

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`

But how?

```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`

But how?

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

Does this fulfill both functor laws?

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

Proof: `fmap` for `Maybe` adheres Functor Law 1

Law 1: `fmap id = id`

```
LHS:  id :: a -> a
      fmap id :: Maybe a -> Maybe a
RHS:  id :: Maybe a -> Maybe a
```

Claim: For all `(x :: Maybe a)` we have `fmap id x == id x`.

Proof: Consider cases for x.

If `x = Nothing`, then:

```
LHS: fmap id Nothing = Nothing
RHS: id Nothing = Nothing
```

If `x = Just y`, then:

```
LHS: fmap id (Just y) = Just (id y) = Just y
RHS: id (Just y) = Just y
```

Proof: `fmap` for `Maybe` adheres Functor Law 2

Law 2: `fmap (f . g)  ==  fmap f . fmap g`

```
f :: b -> c
g :: a -> b
f . g :: a -> c
fmap (f . g) :: Maybe a -> Maybe c
fmap f :: Maybe b -> Maybe c
fmap g :: Maybe a -> Maybe b
fmap f. fmap g :: Maybe a -> Maybe c
```

Claim: For all `(x :: Maybe a)` we have
`fmap (f . g) x == (fmap f . fmap g) x`.

Proof: Consider cases for x.

If x = Nothing, then:

```
... Exercise!
```

If x = Just x, then:

```
... Exercise!
```

`Functor` is just a type class

Like `Eq`, `Ord`, `Show` we saw before, also `Functor` is a type class!

But there is an important difference: `Functor` is a type class of the abstract kind $* \rightarrow *$ and not just $*$.

```
λ> :k Eq
Eq :: * -> Constraint

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

Functor is just a type class

Like Eq, Ord, Show we saw before, also Functor is a type class!

But there is an important difference: Functor is a type class of the abstract kind $* \to *$ and not just $*$.

```
λ> :k Eq
Eq :: * -> Constraint

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

Remember: `instance Eq String` makes `(==)` work on Strings.

In contrast, `instance Functor Maybe` does not give us functions working on things of type `Maybe` because nothing is of type `Maybe`. Instead, it gives us `fmap :: (a -> b) -> Maybe a -> Maybe b`.

`fmap` usage examples

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
```

`fmap` usage examples

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 sort (Just "hello")
Just "ehllo"
```

fmap usage examples

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 sort (Just "hello")
Just "ehllo"

λ> fmap succ [1,5,100::Int]
[2,6,101]
```

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

fmap usage examples

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 sort (Just "hello")
Just "ehllo"
```

```
λ> fmap succ [1,5,100::Int]
[2,6,101]
```

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

There is also an alias for infix notation:

```
λ> sort <$> Just "hello"
Just "ehllo"
```

Applicatives

Applicative Functors

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

Applicative Functors

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

An applicative provides a way to:

- embed pure values (with `pure`)
- sequence computations and combine results (with `(<*>)`)

Applicative Functors

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

An applicative provides a way to:

- embed pure values (with `pure`)
- sequence computations and combine results (with `(<*>)`)

Every `Applicative` is also a `Functor` and we demand:

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

Four Applicative Laws

```
-- preserve identity:
pure id <*>  =  id

-- pure is a homomorphism:
pure f <*> pure x  =  pure (f x)

-- Interchange:
u <*> pure y  =  pure ($ y) <*> u

-- Composition:
pure (.) <*> f <*> g <*> x  =  f <*> (g <*> 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]`

Exercise: Define the functions needed for the Functor and Applicative instances for lists?

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

Try to write down the definitions yourself.

Ensure that the laws and that `fmap f xs = pure f <*> xs` holds.

Monads

Monads: >>=

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

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

Monads: >>=

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

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

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

Example: `instance Monad Maybe`

`Maybe` is a `Monad`, and its 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 its bind function is defined as:

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

Examples:

```
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)
```

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?

The Monad Laws

Left identity:

`return a >>= k  =  k a`

(For example, `Just 5 >>= Just  ==  Just 5`)

Right identity:

`m >>= return  =  m`

(For example, `getLine >>= return == getLine`)

Associativity for bind:

`m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h`

>> then

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

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

This function is also called *then*.

It discards the a-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, `join` *flattens* a box in a box to a single box.

Examples for `join`

```
λ> join (Just (Just 3))
Just 3
λ> join (Just Nothing)
Nothing
λ> join Nothing
Nothing
```

Examples for `join`

```
λ> join (Just (Just 3))
Just 3
λ> join (Just Nothing)
Nothing
λ> join Nothing
Nothing

λ> join (Just (Just (Just "hello")))
Just (Just "hello")
λ> join (join (Just (Just (Just "hello"))))
Just "hello"
```

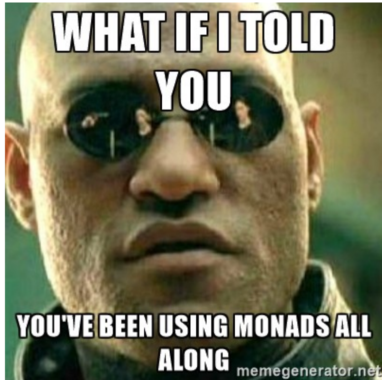
Examples for `join`

```
λ> join (Just (Just 3))
Just 3
λ> join (Just Nothing)
Nothing
λ> join Nothing
Nothing

λ> join (Just (Just (Just "hello")))
Just (Just "hello")
λ> join (join (Just (Just (Just "hello"))))
Just "hello"

λ> join [[1],[2,3],[4::Int]]
[1,2,3,4]
```

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


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

Example: `instance Monad []`

The list container is a monad!

List types are functors, and their mapping function `fmap` is `map`:

```
λ> map (+10) [1,2,3]
[11,12,13]
λ> fmap (+10) [1,2,3]
[11,12,13]
```

It is also applicative, `pure` is done by `\x -> [x]`:

```
λ> pure 3 :: [Int]
[3]
```

The bind operator combines repetition and concatenation:

```
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]
```

Keep Calm

Again, saying "X is a monad" just means that X 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

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` (called *sequence*)

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

What we did with `Maybe` also works with `IO`:

```
λ> fmap sort getLine
hello
"ehllo"
```

(The line `hello` was entered by the user!)

IO in a single line in ghci

What we did with `Maybe` also works with `IO`:

```
λ> fmap sort getLine
hello
"ehllo"
```

(The line `hello` was entered by the user!)

This is the solution to the "Haskell is useless" problem!

Interaction with the real world is represented using the `IO` monad.

do notation

Consider monadic values connected by `then` operators. Example:

```
λ> putStrLn "x" >> putStrLn "y" >> putStrLn "z"
x
y
z
```

do notation

Consider monadic values connected by `then` operators. Example:

```
λ> putStrLn "x" >> putStrLn "y" >> putStrLn "z"
x
y
z
```

Another way to write this is:

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

Think of the monadic values as a sequence actions, and of the *do* notation of a way of presenting it like a sequential program.

do notation with binding

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 and <-

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*.

Do notation is always redundant (but nice)

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

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

Further Reading

No exercise session today, but . . .

- prove (some of) the laws!
- practice, practice, practice!
- see website for research/project topics!

Further Reading

No exercise session today, but . . .

- prove (some of) the laws!
- practice, practice, practice!
- see website for research/project topics!

Further rading:

- LYHFGG: *Kinds and some type-foo*. http://learnyouahaskell.com/making-our-own-types-and-typeclasses#kinds-and-some-type-foo
- Philip Wadler: *Category Theory for the Working Hacker*. https://youtu.be/V10hzjgoklA
- Brent Yorgey: *Typeclassopedia*. https://wiki.haskell.org/Typeclassopedia
- MG: Why IO Input Types Are Bad. https://malv.in/posts/2016-11-16-why-io-input-types-are-confusing.html

See you on Monday at 09:00.
(Location and modus to be announced!)

Have a nice weekend!

Garfield tried learning Haskell