Lab 11. Functors & Monads

11.0. Understanding Functors and Monads

Functor

The **Functor** typeclass represents the mathematical functor: a mapping between categories.

In Haskell, Functors are defined as follows:

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

Remember the type of `map`?

```
map :: (a -> b) -> [a] -> [b]
```

This is a specific implementation of `fmap`, that exists in Haskell for historic reasons, a list also has a `Functor` instance defined, with `fmap = map`

We can declare a `Functor` instance for any abstract type `f a` which has the ability to 'map' over it's value, while preserving the structure of `f`.

Declaring `Functor` instance for `f`, allows us to use functions relating to mapping for any type `f a`.

For `fmap` to have a predictible behaviour, any `Functor` instance we define needs to obey the **Functor Laws**:

```
-- Functors must preserve identity morphisms
fmap id = id


-- Functors preserve composition of morphisms
fmap (f . g) = fmap f . fmap g
```

Monad

A **monad** is an algebraic structure used to describe computations as sequences of steps, and to handle side effects such as state and IO. They also provide a clean way to structure our programs.

In Haskell, Monads are defined as follows:

```
class Monad m where
```

```
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

You can think of Monads as **containers** that keep a variable inside a context. We will see several examples in the next sections.

For monads to behave correctly, a monad instance needs to obey the **Monad Laws**:

```
-- left identity
return a >>= h = h a


-- right identity
m >>= return = m


-- associativity
(m >>= g) >>= h = m >>= (\x -> g x >>= h)
```

NOTE: any Monad instance has a corresponding Functor and Applicative (we will not discuss applicatives for this course, but Haskell requires it). If you defined a Monad instance for a class `Foo`, you can define the Functor and Applicative as follows:

Click to hide ↖

```
instance Functor Foo where
  fmap f fx = px >>= (\x -> return (f x))


instance Applicative Foo where
  pure = return
  ff <*> fx = ff >>= (\f -> fx >>= (\x -> return (f x)))
```

Do notation

You can probably notice a lot of monadic code uses binds and lambda functions. Take for example the following snippet that uses monads to unpack 2 `Maybe` values and add them.

```
add ma mb = ma >>= (\a ->
            mb >>= (\b ->
            return (a + b)))
```

Haskell provides **do-notation**, which is a syntactic sugar for this kind of expressions (also called hanging lambdas).

```
add ma mb = do
    a <- ma
    b <- mb
    return (a + b)
```

11.1. Working with 'Maybe'

`Maybe` is already defined as a Functor and Monad:

```
instance Functor Maybe where
  fmap f x =
    case x of
       Just v -> Just (f v)
       Nothing -> Nothing


instance Monad Maybe where
  return = Just
  mx >>= f =
      case mx of
        Just v -> f v
        Nothing -> Nothing
```

This section is meant to accommodate you to using Functors and Monads by playing around with an already implemented and familiar Monad: Maybe.

11.1.1. Implement a function `add5` that adds 5 to a `Maybe Int`, use `fmap` instead of `case` or pattern matching.

```
add5 :: Maybe Int -> Maybe Int
```

11.1.2. Implement functions `add`, `sub` and `mult` that add, substract and multiply `Maybe Int`, use `do` notation instead of `case` or pattern matching.

```
add :: Maybe Int -> Maybe Int -> Maybe Int
sub :: Maybe Int -> Maybe Int -> Maybe Int
mult :: Maybe Int -> Maybe Int -> Maybe Int
```

11.2. Working with IO

We can finally learn about IO in Haskell. Because it's a pure functional language, where we have no side-effects, IO is not really possible, reading external values or printing values is inherently a side-effect. Because of this, IO is defined as a Monad, where the context is the external world.

Very simply put, something with type `IO a` is a value of type `a` coming from the external world. When we print something, the type is simply `IO ()`, because there is no value coming from the external world, we just interact with it.

Let's look at the type annotations of some usual IO functions:

```
print :: Show a => a -> IO ()

putStrLn :: String -> IO ()

getLine :: IO String
```

As a side-point, now that we know IO we can also write complete haskell programs that are executable.
The main function in Haskell has the following type annotation:

```
main :: IO ()
```

This means a `main` function is just a function that interacts with the external world.

You can use the commandline utility `runhaskell main.hs` to quickly test your Haskell programs without compiling them.

11.2.1. Write a `Hello World` program and execute it.

11.2.2. Write a program that reads a value `n` and prints the `n-th` fibonacci number.

You can use `read` to convert from strings to certain types (is the opposite of `show`).

11.2.3. Write a program that reads a value `n`, a vector of `n` numbers and prints them in sorted order.

Don't forget `IO` is also a `Functor`, we can `fmap` over yet 'unread' values.

11.2.4. Write a program that reads a value `n`, and prints the sequences given by the **Collatz conjecture**. If the current number `x` is even, the next number is `x / 2`, if `x` is odd the next number is `3 * x + 1`, the sequence stops when you hit the number 1.

11.3. Lists

Let's take a moment and think about lists, we already experimented with Functors over lists a lot. It's simply the `map` function we are extremely familiar with, but what would mean to **sequence** 2 lists using (>>=)(>>=)?
Hint: We actually have already seen this behaviour before while using list comprehensions.
Click to hide ⇱
Sequencing lists is achieved by taking the cartesian product of the 2 lists and flattening it.

```haskell
[1, 2, 3] >>= (\x -> [4, 5, 6] >>= (\y -> return (x, y))) = [(1, 4), (1, 5), (1, 6),
(2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]


-- isn't this familiar?

[(x, y) | x <- [1, 2, 3], y <- [4, 5, 6]]
```

What is the context a list puts a value in? **Non-determinism**, a list represent the fact that the object can have multiple possible values.

We will take a simple example where the list monad might be very helpful. Conside a 8×8 chess board. A position would be:

```haskell
type Pos = (Int, Int)
```

11.3.1. Make a function that given a position, tells you where a Knight might be able to move (how a Knight moves).

```haskell
moveKnight :: Pos -> [Pos]

moveKnight (x, y) = do

   ???
```

11.3.2. Make a function that given a start position and a target position, tells you if a Knight can get to the target position in exactly 3 moves.

```haskell
canReachIn3 :: Pos -> Pos -> Bool

canReachIn3 = undefined
```

11.3.3.(* * *) Make a function that given a number `k`, a start position and a target position, tell you if a Knight can get to the target position in exactly `k` moves.

```haskell
canReachInK :: Int -> Pos -> Pos -> Bool

canReachInK = undefined
```

11.4. Probability Distributions (M3 flashbacks)

Let's also try to define our own Monads. Keep in mind that we should never have the goal of making something a monad, rather we should define types that model aspects of our problem, and if we notice that they represent types with context that can be abstracted using monads, use them for simplicity.

One such type is a Probability Distribution, we can define it as a list of pairs of value and probability.

```haskell
newtype Prob a = Prob [(a, Float)] deriving Show
```

11.4.1. Define a `Functor` instance for `Prob`, mapping over a probability should change the value, but not affect the probability. Check that the functor laws apply.

```haskell
instance Functor Prob where
    fmap = undefined


prob :: Prob Int
prob = Prob [(1, 0.5), (2, 0.25), (3, 0.25)]


-- fmap (+3) prob = Prob [(4, 0.5), (5, 0.25), (6, 0.25)]
```

11.4.2. Define a auxiliary function `flatten` that takes a probability of probabilities (`Prob (Prob a)`) and returns a probability (`Prob a`).

```haskell
flatten :: Prob (Prob a) -> Prob a
flatten = undefined


nested_prob :: Prob (Prob Int)
nested_prob = Prob [(prob, 0.7), (prob, 0.3)]


-- flatten nested_prob = Prob [(1,0.35),(2,0.175),(3,0.175),(1,0.15),(2,7.5e-2),(3,7.5e-2)]
```

11.4.3. Using `flatten`, define a `Monad` instance for `Prob` (it should behave in a similar fashion with the List monad, but also keep the probability context). Check that the monad laws apply.

```haskell
instance Monad Prob where
    return x = undefined
    m >>= f = undefined


-- Monad expects Applicative to be declared as well, take this as it is
instance Applicative Prob where
    pure = return
    pf <*> px = do
        f <- pf
        x <- px
```

```
        return (f x)
```

Let's play around with our monad, let's define a coin:

```
data Coin = Heads | Tails deriving Show


coin :: Prob Coin
coin = Prob [(Heads, 0.5), (Tails, 0.5)]


unfair_coin :: Prob Coin
unfair_coin = Prob [(Heads, 0.6), (Tails, 0.4)]


flip :: Prob [Coin]
flip = do
  x <- coin
  y <- coin
  z <- coin
  return [x, y, z]
```

11.4.4. Make a function that returns the probability distribution of a fair **n**-sided die.

```
die :: Int -> Prob Int
die n = undefined


-- (,) <$> (die 20) <*> (die 6) -- the probability distribution of rolling a d20
followed by a d6
```

11.4.5. Let's use this framework to solve a M3 problem: “Jo has took a test for a disease. The result of the test is either positive or negative and the test is 95% reliable: in 95% of cases of people who really have the disease, a positive result is returned, and in 95% of cases of people who do not have the disease, a negative result is obtained. 1% of people of Jo’s age and background have the disease. Jo took the test, and the result is positive. What is the probability that Jo has the disease?”