Lec 08 - Haskell Functors and Applicatives

CS 1XA3

March 6^{th}, 2018

The Maybe Type: A Prime Example

- Parameterized Datatypes can often be thought of as containers for a parameter, like a box with a label you put values in
- A prime example of this is the Maybe type, which can be defined like

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

- You can pull a Maybe value out of it's "box" using pattern matching

```
stringOrEmpty :: Maybe String -> String
stringOrEmpty maybeStr = case maybeStr of
                           Just str -> str
                           Nothing  -> ""
```

Functors (the Mappable TypeClass)

- Think of the Functor type class as the class of Mappable datatypes (i.e data types for which the map) function can be defined over

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

- Note: the type parameter f is a data type that takes a single type parameter
- fmap is the same function as map that you've been using over lists (in fact you can use fmap instead of map)

```
fmap (\x -> x + 1) [1,2,3,4,5]
-- works just like map
```

Functors (the Mappable TypeClass)

- Imagine you wish to apply a function to a Maybe type, but you don't know what to do in the case of Nothing
- What should you do? In the event of Nothing, you probably want to return Nothing again! For Example

```
incrMaybe :: Maybe Int -> Maybe Int
incrMaybe mbInt = case mbInt of
                    Just int -> Just (int+!)
                    Nothing  -> Nothing
```

- All this wrapping and unwrapping of the Maybe type is tedious, luckily Maybe is a Functor

```
incrMaybe :: Maybe Int -> Maybe Int
incrMaybe mbInt = fmap (+1) mbInt
```

Functors (the Mappable TypeClass)

- How is the Functor instance for Maybe defined?

Functors (the Mappable TypeClass)

- How is the Functor instance for Maybe defined?
- ```
  instance Functor Maybe where
    fmap f (Just a) = Just (f a)
    fmap f Nothing  = Nothing
  ```
- There is a natural intuition for which types are Mappable, but we're good computer scientists and would prefer some formal rules to follow! Functors should obey the following laws

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

Note: the Functor Laws aren't automatically enforced by Haskell (yet!), it's up to the programmer to make sure they're obeyed
  ```

Applicatives (Functors on Steroids!)

- Functors give us a way to apply a function to a wrapped value. But what if the function itself is wrapped!!

```
fmap (Just (+1)) (Just 5) -- doesn't work!
```

- The Applicative class defines two functions over a datatype **f** that **MUST ALSO BE A** Functor

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

- This way we can define the previous attempted computation correctly

```
(Just (+1)) <*> (Just 5) -- works
(pure (+1)) <*> (Just 5) -- also works
```

Applicatives (Functors on Steroids!)

- How is the Maybe instance defined for Applicative?

Applicatives (Functors on Steroids!)

- How is the Maybe instance defined for Applicative?
- ```
  instance Applicative Maybe where
    pure = Just
    Nothing <*> _   = Nothing
    (Just f) <*> x = fmap f x
  ```
- Just like Functors, there are laws Applicatives should obey

```
pure id <*> v = v
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```
```

Coming Up Next: Monads!

- After we understand the power of Functors and Applicatives, we can enter the power of the Monad!
- I highly recommend reading `http://learnyouahaskell.com/a-fistful-of-monads` before our next lecture