BAHUG 101 - Lecture 5

9th November 2015

Outline of Today's Lecture

- IO
- **do** syntax

Pure Functions

Problems with purity

We have discussed how to compute things in Haskell, but we have been missing two ways of computing.

Problems with purity

We have discussed how to compute things in Haskell, but we have been missing two ways of computing.

1. Functions may not have any external effects, like printing to screen.

Problems with purity

We have discussed how to compute things in Haskell, but we have been missing two ways of computing.

1. Functions may not have any external effects, like printing to screen.

2. Functions may not depend on external things, such as file systems, etc.

The `IO` type

IO values denote computations that may cause an effect on the system.

The `IO` type

IO values denote computations that may cause an effect on the system.

- `IO ()` denotes an effectful computation that returns "nothing" (`Unit` or `()`).

```
ex1 :: IO ()
ex1 = putStrLn "Hello World!" -- Puts a string on the console.
```

The `IO` type

IO values denote computations that may cause an effect on the system.

- `IO ()` denotes an effectful computation that returns "nothing" (`Unit` or `()`).

```
ex1 :: IO ()
ex1 = putStrLn "Hello World!" -- Puts a string on the console.
```

- `IO String` denotes an effectful computation that returns a string.

```
ex2 :: IO String
ex2 = getLine -- Read a line from standard input.
```

The `IO` type

IO values denote computations that may cause an effect on the system.

- `IO ()` denotes an effectful computation that returns "nothing" (`Unit` or `()`).

```
ex1 :: IO ()
ex1 = putStrLn "Hello World!" -- Puts a string on the console.
```

- `IO String` denotes an effectful computation that returns a string.

```
ex2 :: IO String
ex2 = getLine -- Read a line from standard input.
```

Note that while `IO` suggests an effectful computation, one does not necessarily happen.

The `IO` type

IO values denote computations that may cause an effect on the system.

- `IO ()` denotes an effectful computation that returns "nothing" (`Unit` or `()`).

```
ex1 :: IO ()
ex1 = putStrLn "Hello World!" -- Puts a string on the console.
```

- `IO String` denotes an effectful computation that returns a string.

```
ex2 :: IO String
ex2 = getLine -- Read a line from standard input.
```

Note that while `IO` suggests an effectful computation, one does not necessarily happen.

But a value not of type `IO` can never cause an effectful computation.

What can you do with an IO value?

`IO` can be stored in a data structure

```
ex3 :: [IO ()]
ex3 = [putStr "Lists", putStr " ", putStr "are",
       putStr " ", putStr "cool", putStr "\n"]
```

What can you do with an IO value?

`IO` can be stored in a data structure

```
ex3 :: [IO ()]
ex3 = [putStr "Lists", putStr " ", putStr "are",
       putStr " ", putStr "cool", putStr "\n"]
```

In order to run something, it must be called by `main`.

```
runEx3 :: IO ()
runEx3 = sequence_ ex3 -- type [IO ()] -> IO ()

main :: IO ()
main = runEx3
```

There is no `String` "inside" an `IO String`

If we remember **Maybe**, we can convert from **Maybe** into a concrete value.

```
data Maybe a = Nothing | Just a

replaceNothing :: Maybe Double -> Double
replaceNothing (Nothing) = 0
replaceNothing (Just x) = x
```

There is no `String` "inside" an `IO String`

If we remember **Maybe**, we can convert from **Maybe** into a concrete value.

```
data Maybe a = Nothing | Just a

replaceNothing :: Maybe Double -> Double
replaceNothing (Nothing) = 0
replaceNothing (Just x) = x
```

We *cannot* do this for **IO**.

```
ex5 :: IO String -> String
ex5 = undefined — bottom
```

There is no `String` "inside" an `IO String`

If we remember `Maybe`, we can convert from `Maybe` into a concrete value.

```
data Maybe a = Nothing | Just a

replaceNothing :: Maybe Double -> Double
replaceNothing (Nothing) = 0
replaceNothing (Just x) = x
```

We *cannot* do this for `IO`.

```
ex5 :: IO String -> String
ex5 = undefined -- bottom
```

In a sense, `IO` *taints* values to force them to always be used in an effectful environment.

Sequencing `IO` actions

We can sequence `IO` actions together using `do` notation.

```
sillyExchange :: IO ()
sillyExchange = do
  putStrLn "Hello, user!"
  putStrLn "What is your name?"
  name <- getLine -- string "pulled out of" IO String
  putStrLn $ "Pleased to meet you, " ++ name ++ "!"
```

Must be used in `main` or `ghci` in order to run.

Sequencing `IO` actions

We can sequence `IO` actions together using `do` notation.

```
sillyExchange :: IO ()
sillyExchange = do
  putStrLn "Hello, user!"
  putStrLn "What is your name?"
  name <- getLine -- string "pulled out of" IO String
  putStrLn $ "Pleased to meet you, " ++ name ++ "!"
```

Must be used in `main` or `ghci` in order to run.

Note that `<- getLine` does not have a type; it is just syntax.

A slightly larger example

```
jabber :: IO ()
jabber = do
  wockylines ← fmap (drop 2 . lines) $ readFile "jabberwocky.txt"
  count ← printFirstLines wockylines
  putStrLn $ "There are " ++ show count ++ " stanzas."

printFirstLines :: [String] → IO Int
printFirstLines ls = do
  let first_lines = unlines . extractFirstLines $ ls
  putStr first_lines
  return $ length first_lines

extractFirstLines :: [String] → [String]
extractFirstLines []          = []
extractFirstLines [_]         = []
extractFirstLines ("" : first : r) = first : extractFirstLines r
extractFirstLines (_ : r) = extractFirstLines r
```

`return` in Haskell

`return` is a bit strange. In the above example, it has the type

```
return :: a → IO a
```

[1] the actual type of return is `return :: Monad m ⇒ a → m a`

`return` in Haskell

`return` is a bit strange. In the above example, it has the type

```
return :: a → IO a
```

`do` notation requires that the last line is something like `IO a`.[1]

[1] the actual type of return is `return :: Monad m ⇒ a → m a`

More Types!

Record Syntax

Suppose we have a data type such as

```
data D = C T1 T2 T3
```

Record Syntax

Suppose we have a data type such as

```
data D = C T1 T2 T3
```

We could also declare this data type with *record syntax* as follows:

```
data D = C { field1 :: T1, field2 :: T2, field3 :: T3 }
```

This creates a "projection function" for each field.

```
field1 :: D -> T1
field1 (C f1 _ _) = f1

field2 :: D -> T2
field2 (C _ f2 _) = f2

field3 :: D -> T3
field3 (C _ _ f3) = f3
```

Modifying record values

There is special syntax for *constructing*, *modifying*, and *pattern-matching* on values of type `D` (in addition to the usual syntax for such things).

Modifying record values

There is special syntax for *constructing*, *modifying*, and *pattern-matching* on values of type `D` (in addition to the usual syntax for such things).

We can construct values with the following syntax.

```
C { field3 = ..., field1 = ..., field2 = ... }
```

Modifying record values

There is special syntax for *constructing*, *modifying*, and *pattern-matching* on values of type `D` (in addition to the usual syntax for such things).

We can construct values with the following syntax.

```
C { field3 = ..., field1 = ..., field2 = ... }
```

We can "modify" values using the following syntax (for a value `d`)

```
d { field3 = ... }
```

This creates a new value; no mutation happens.

Modifying record values

There is special syntax for *constructing*, *modifying*, and *pattern-matching* on values of type `D` (in addition to the usual syntax for such things).

We can construct values with the following syntax.

```
C { field3 = ..., field1 = ..., field2 = ... }
```

We can "modify" values using the following syntax (for a value `d`)

```
d { field3 = ... }
```

This creates a new value; no mutation happens.

We can pattern match

```
foo (C { field1 = x }) = ... x ...
```

ByteStrings

`String` is a bit inefficient (a linked list of `Char`). More efficient strings can be done using `ByteString`, which is imported as follows

```
import qualified Data.ByteString as BS
```

ByteStrings

`String` is a bit inefficient (a linked list of `Char`). More efficient strings can be done using `ByteString`, which is imported as follows

```
import qualified Data.ByteString as BS
```

The `qualified` bit means that we need to reference functions using `BS.*`. For example, `BS.length`

ByteStrings

`String` is a bit inefficient (a linked list of `Char`). More efficient strings can be done using `ByteString`, which is imported as follows

```
import qualified Data.ByteString as BS
```

The `qualified` bit means that we need to reference functions using `BS.*`. For example, `BS.length`

Often import `ByteString` itself (unqualified) so we don't need to write `BS.ByteString` in the type.

```
import Data.ByteString (ByteString)
```

Overloaded Strings

In Haskell, if you type `4`, it could be either a `Float`, `Double`, `Int`, `Integer`, etc. This is called *overloaded numbers*.

Overloaded Strings

In Haskell, if you type `4`, it could be either a `Float`, `Double`, `Int`, `Integer`, etc. This is called *overloaded numbers*.

We can do this with strings as well using the `Overloaded Strings` extension.

```
{-# LANGUAGE OverloadedStrings #-}
```

Overloaded Strings

In Haskell, if you type `4`, it could be either a `Float`, `Double`, `Int`, `Integer`, etc. This is called *overloaded numbers*.

We can do this with strings as well using the `Overloaded Strings` extension.

```
{-# LANGUAGE OverloadedStrings #-}
```

This puts either the `fromString` call in front of a value

Overloaded Strings

In Haskell, if you type `4`, it could be either a `Float`, `Double`, `Int`, `Integer`, etc. This is called *overloaded numbers*.

We can do this with strings as well using the `Overloaded Strings` extension.

```
{-# LANGUAGE OverloadedStrings #-}
```

This puts either the `fromString` call in front of a value

Sometimes Haskell doesn't know what you want type of string you want, so you will have to give the string a type.

`ByteString` under the hood

- `ByteString`s are sequences of (more traditional) 8-bit characters (called `Word8`s.)
- Can't use character literals like `'a'` when dealing with `ByteString`s (although you can still use string literals!).
- `Word8`s are instances of `Num` and `Integral`, so you can use all of your favorite numeric functions on them!

IO is a Functor!

Here is a simple function that takes in some input and makes a list of words.

```
getWords :: IO [ByteString]
getWords = do
  ln ← BS.getLine
  return $ BS.split 32 ln — 32 is the ASCII code for ' '
```

IO is a Functor!

Here is a simple function that takes in some input and makes a list of words.

```
getWords :: IO [ByteString]
getWords = do
  ln ← BS.getLine
  return $ BS.split 32 ln — 32 is the ASCII code for ' '
```

IO is a Functor, so we can make this quite a bit cleaner.

```
getWords' :: IO [ByteString]
getWords' = BS.split 32 <$> BS.getLine
```

This simply maps the (pure) splitting operation over the result of the IO action **BS.getLine**.