BAHUG 101 - Lecture 8

7th April 2016

Outline of Today's Lecture

- Applicatives
- Parsing
- More monads!

Motivation

Employee Example

Consider the following **`Employee`** type:

```
type EName = String

data Employee = Employee { employee_name    :: EName
                         , employee_phone   :: String }
                deriving Show
```

Employee Example

Consider the following **Employee** type:

```
type EName = String

data Employee = Employee { employee_name    :: EName
                         , employee_phone   :: String }
                deriving Show
```

If we want to make an **Employee**, we of course can use the constructor.

```
Employee :: EName → String → Employee
```

Maybe Employee

Can we convert a function of the following type

```
EName -> String -> Employee
```

to one that uses `Maybe`s?

```
Maybe EName -> Maybe String -> Maybe Employee
```

Maybe Employee

Can we convert a function of the following type

```
EName -> String -> Employee
```

to one that uses `Maybe`s?

```
Maybe EName -> Maybe String -> Maybe Employee
```

```
maybeEmployee :: Maybe EName -> Maybe String -> Maybe Employee
maybeEmployee  Nothing  _         = Nothing
maybeEmployee  _        Nothing   = Nothing
maybeEmployee (Just n) (Just p) = Just (Employee n p)
```

Lists of Employees

Now, can we convert a function of the following type

```
EName -> String -> Employee
```

to one that uses `List`?

```
[EName] -> [String] -> [Employee]
```

Lists of Employees

Now, can we convert a function of the following type

```
EName -> String -> Employee
```

to one that uses `List`?

```
[EName] -> [String] -> [Employee]
```

This could work in two ways

- Match each name with the associated phone in the second list.

Lists of Employees

Now, can we convert a function of the following type

```
EName -> String -> Employee
```

to one that uses `List`?

```
[EName] -> [String] -> [Employee]
```

This could work in two ways

- Match each name with the associated phone in the second list.
- The Cartesian product of the name and phone lists.

Using Reader with Employee?

Well what if we have the functions `e -> EName` and `e -> String` for some `e`. Can we make a function of

```
EName -> String -> Employee
```

into the following?

```
(e -> EName) -> (e -> String) -> (e -> Employee)
```

Using Reader with Employee?

Well what if we have the functions `e -> EName` and `e -> String` for some `e`. Can we make a function of

```
EName -> String -> Employee
```

into the following?

```
(e -> EName) -> (e -> String) -> (e -> Employee)
```

There is only one way to write this one.

How do we generalize these examples?

What we would like to generalize our examples is a function like this

```
(a → b → c) → (f a → f b → f c)
```

How do we generalize these examples?

What we would like to generalize our examples is a function like this

```
(a -> b -> c) -> (f a -> f b -> f c)
```

Well that kinda looks like `fmap`.

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

How do we generalize these examples?

What we would like to generalize our examples is a function like this

```
(a -> b -> c) -> (f a -> f b -> f c)
```

Well that kinda looks like `fmap`.

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

But we need an fmap that can work for functions of two arguments

```
fmap2 :: Functor f => (a -> b -> c) -> (f a -> f b -> f c)
fmap2 h fa fb = undefined
```

But we can't use Functor!

Try hard as we might, however, **Functor** does not quite give us enough to implement **fmap2**. We have

```
h  :: a → (b → c)
fa :: f a
fb :: f b
```

But we can't use `Functor`!

Try hard as we might, however, `Functor` does not quite give us enough to implement `fmap2`. We have

```
h  :: a -> (b -> c)
fa :: f a
fb :: f b
```

If we try to apply `fmap`, we get a function in our functor that we can't apply.

```
h         :: a -> (b -> c)
fmap h    :: f a -> f (b -> c)
fmap h fa :: f (b -> c)
```

Applicative

Applicative Definition

Functors for which this sort of “contextual application” is possible are called `Applicative`.

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

On top of what a `Functor` offers, we have the following functions

- `pure`: promotes a value into the functor (ex. `a -> [a]`)
- `(<*>)`: applies a function inside a functor

Implementing `fmap2`

Now that we have `(<*>)`, we can implement `fmap2`, which in the standard library is actually called `liftA2`:

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 h fa fb = (h `fmap` fa) <*> fb
```

Implementing `fmap2`

Now that we have `(<*>)`, we can implement `fmap2`, which in the standard library is actually called `liftA2`:

```haskell
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 h fa fb = (h `fmap` fa) <*> fb
```

This pattern is common, so `(<$>)` as a synonym for `fmap`,

```haskell
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

so that we can write

```haskell
liftA2 h fa fb = h <$> fa <*> fb
```

What about liftA3?

Defining **liftA3** is as simple as it was to define **liftA2**.

```
liftA3 :: Applicative f => (a -> b -> c -> d) ->
                           f a -> f b -> f c -> f d
liftA3 h fa fb fc = h <$> fa <*> fb <*> fc
```

We can use this pattern to create any **liftA#** without a new typeclass.

What about `liftA3`?

Defining `liftA3` is as simple as it was to define `liftA2`.

```
liftA3 :: Applicative f => (a -> b -> c -> d) ->
                           f a -> f b -> f c -> f d
liftA3 h fa fb fc = h <$> fa <*> fb <*> fc
```

We can use this pattern to create any `liftA#` without a new typeclass.

We can also use `Applicative` on regular values since we can put them in a `Functor` by using `pure`.

```
liftX :: Applicative f => (a -> b -> c -> d) ->
                          f a -> b -> f c -> f d
liftX h fa b fc = h <$> fa <*> pure b <*> fc
```

Defining Applicative for Maybe

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

Applicative Maybe examples

```haskell
m_name1, m_name2 :: Maybe EName
m_name1 = Nothing
m_name2 = Just "Brent"

m_phone1, m_phone2 :: Maybe String
m_phone1 = Nothing
m_phone2 = Just "555-1234"

ex01, ex02, ex03, ex04 :: Maybe Employee
ex01 = Employee <$> m_name1 <*> m_phone1
ex02 = Employee <$> m_name1 <*> m_phone2
ex03 = Employee <$> m_name2 <*> m_phone1
ex04 = Employee <$> m_name2 <*> m_phone2
```

Parsing

Parser Overview

A *parser* is an algorithm which takes unstructured data and produces structured data.

Examples include

- json parsers.
- the `ghci` REPL.
- any interpreter.

Parser Overview

A *parser* is an algorithm which takes unstructured data and produces structured data.

Examples include

- json parsers.
- the **ghci** REPL.
- any interpreter.

For this lecture, we will be using the parsing package **Attoparsec**. It allows us to define simple parsers like so.

```
word :: Parser ByteString
word = takeWhile $ inClass "a-zA-Z"
```

Parsers for words

Now, let's write a parser for names. Names start with a capital letter and then contain lower case letters.

```
upper :: Parser Word8
upper = satisfy $ inClass "A-Z"

lword :: Parser ByteString
lword = takeWhile (inClass "a-z")
```

Parsers for names

We can use these two primitive parsers, with **ByteString.cons**, to make the **name** parser. This is possible using the **Parser**'s **Applicative** definition.

```
name :: Parser ByteString
name = BS.cons <$> upper <*> lword
```

Or, alternatively, we can *lift* the **cons** function into the **Parser** using **liftA2**:

```
name' :: Parser ByteString
name' = liftA2 BS.cons upper lword
```

Parsing full names

Suppose we want to parse full names (ie, first and last). Instead of returning a `ByteString`, we want to structure the output data. But first we need a way to skip spaces.

```
skipSpace :: Parser () -- Don't care about captures characters.
skipSpace = skipWhile isSpace_w8
```

Combining `skipSpace` with `(*>)` allows us to ignore the parser's output. Now, let's write the full name parser:

```
firstLast :: Parser (ByteString, ByteString)
firstLast = (,) <$> name <*> (skipSpace *> name)
```

Full names with potential middle names

We want to target the following data type:

```
data Name = TwoName ByteString ByteString
          | ThreeName ByteString ByteString ByteString
  deriving Show
```

Unfortunately, there is no way to do this using applicative functors as our combinator because we can't make decisions with `Applicative`. We will have to use something stronger than applicative.

Monads to the Rescue!

Recall that the **Monad** type class exposes two function:

```
class Monad m where
    return :: a → m a
    (>>=)  :: m a → (a → m b) → m b
```

In general, monads can be used to sequence actions and make choices on prior results.

First and last name parser

Let's write a parser for first and last name using monads instead of applicative functors like we did above.

```
firstLast' :: Parser (ByteString, ByteString)
firstLast' = do
  fname <- name
  lname <- skipSpace *> name
  return (fname, lname)
```

When possible, it is good practice to use `Applicative` instead of `Monad`.

Full name parser

The full name parser can be written this way.

```
fullName :: Parser Name
fullName = do
  n1 ← name
  n2 ← skipSpace *> name
  mn ← skipSpace *> optional name
  case mn of
    Just n3 → return $ ThreeName n1 n2 n3
    Nothing → return $ TwoName n1 n2
```

Note that we used the `optional :: Parser a → Parser (Maybe a)` function above. This is a function defined by Attoparsec that allows a parser to fail without terminating the entire computation.

Parsing an ambiguous strings

Consider the following string.

`”Haskell Brooks Curry Simon Peyton Jones”`.

Parsing an ambiguous strings

Consider the following string.

`"Haskell Brooks Curry Simon Peyton Jones"`.

Should this string be parsed as the list

```
[TwoName "Haskell" "Brooks", TwoName "Curry" "Simon",
 TwoName "Peyton" "Jones"]
```

or the list

```
[ThreeName "Haskell" "Brooks" "Curry",
 ThreeName "Simon" "Peyton" "Jones"]
```

Updated string

Since the prior string was ambiguous, we will parse the following.

`"[true, true] Haskell Brooks Curry Simon Peyton Jones"`

This signifies that there are two names in the sequence of words and both of them have middle names.

Creating a bool list parser

We can create the bool list parser as follows.

```
bool :: Parser Bool
bool = do
  s ← word
  case s of
    "true"  → return True
    "false" → return False
    _       → fail $ show s ++ " is not a bool"
```

Creating a bool list parser

We can create the bool list parser as follows.

```haskell
bool :: Parser Bool
bool = do
  s <- word
  case s of
    "true"  -> return True
    "false" -> return False
    _       -> fail $ show s ++ " is not a bool"

list :: Parser a -> Parser [a]
list p = char '[' *> sepBy p comma <* char ']'
    where comma = skipSpace *> char ',' <* skipSpace
```

Creating a bool list parser

We can create the bool list parser as follows.

```
bool :: Parser Bool
bool = do
  s ← word
  case s of
    "true"  → return True
    "false" → return False
    _       → fail $ show s ++ " is not a bool"

list :: Parser a → Parser [a]
list p = char '[' *> sepBy p comma <* char ']'
    where comma = skipSpace *> char ',' <* skipSpace

boolList :: Parser [Bool]
boolList = list bool
```

Note that the **sepBy** function creates a parser for a list of values that are separated by some other parser. In this case, the parser that separates the list elements is a comma surrounded by

Complete full name parser

We can now use this list of `Bool`s to figure out how to parse the names.

```
names :: Parser [Name]
names = boolList >>= mapM bToP
  where
        bToP :: Bool -> Parser Name
        bToP True  = ThreeName <$> sn <*> sn <*> sn
        bToP False = TwoName   <$> sn <*> sn

        sn :: Parser ByteString
        sn = skipSpace *> name
```

This can be called using

```
result :: [Name]
result = fromRight $ parseOnly names
  "[true, true] Haskell Brooks Curry Simon Peyton Jones"
```