Monads Part II

CIS 194 Week 8
18 March 2015

Suggested reading:

- **Applicative Functors** from Learn You a Haskell
- **The Typeclassopedia**

Applicative Functors

Motivation

Consider the following `Employee` type:

```
type Name = String

data Employee = Employee { name    :: Name
                         , phone   :: String }
                deriving Show
```

Of course, the `Employee` constructor has type

```
Employee :: Name -> String -> Employee
```

That is, if we have a `Name` and a `String`, we can apply the `Employee` constructor to build an `Employee` object.

Suppose, however, that we don't have a `Name` and a `String`; what we actually have is a `Maybe Name` and a `Maybe String`. Perhaps they came from parsing some file full of errors, or from a form where some of the fields might have been left blank, or something of that sort. We can't necessarily make an `Employee`. But surely we can make a `Maybe Employee`. That is, we'd like to take our `(Name -> String -> Employee)` function and turn it into a `(Maybe Name -> Maybe String -> Maybe Employee)` function. Can we write something with this type?

```
(Name -> String -> Employee) ->
(Maybe Name -> Maybe String -> Maybe Employee)
```

Sure we can, and I am fully confident that you could write it in your sleep by now. We can imagine how it would work: if either the name or string is `Nothing`, we get `Nothing` out; if both are `Just`, we get out an `Employee` built using the `Employee` constructor (wrapped in `Just`). But let's keep going…

Consider this: now instead of a `Name` and a `String` we have a `[Name]` and a `[String]`. Maybe we can get an `[Employee]` out of this? Now we want

```
(Name -> String -> Employee) ->
([Name] -> [String] -> [Employee])
```

We can imagine two different ways for this to work: we could match up corresponding `Name`s and `String`s to form `Employee`s; or we could pair up the `Name`s and `String`s in all possible ways.

Or how about this: we have an `(e -> Name)` and `(e -> String)` for some type `e`. For example, perhaps `e` is some huge data structure, and we have functions telling us how to extract a `Name` and a `String` from it. Can we make it into an `(e -> Employee)`, that is, a recipe for extracting an `Employee` from the same structure?

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

No problem, and this time there's really only one way to write this function.

Generalizing

Now that we've seen the usefulness of this sort of pattern, let's generalize a bit. The type of the function we want really looks something like this:

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

Hmm, this looks familiar… it's quite similar to the type of `fmap`!

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

The only difference is an extra argument; we might call our desired function `fmap2`, since it takes a function of two arguments. Perhaps we can write `fmap2` in terms of `fmap`, so we just need a `Functor` constraint on `f`:

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

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

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

Note that we can also write the type of `h` as `a -> (b -> c)`. So, we have a function that takes an `a`, and we have a value of type `f a`… the only thing we can do is use `fmap` to lift the function over the `f`, giving us a result of type:

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

OK, so now we have something of type `f (b -> c)` and something of type `f b`… and here's where we are stuck! `fmap` does not help any more. It gives us a way to apply functions to values inside a `Functor` context, but what we need now is to apply a functions *which are themselves in a `Functor` context* to values in a `Functor` context.

Applicative

Functors for which this sort of "contextual application" is possible are called *applicative*, and the `Applicative` class (defined in `Control.Applicative`) captures this pattern.

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

The `(<*>)` operator (often pronounced "ap", short for "apply") encapsulates exactly this principle of "contextual application". Note also that the `Applicative` class requires its instances to be instances of `Functor` as well, so we can always use `fmap` with instances of `Applicative`. Finally, note that `Applicative` also has another method, `pure`, which lets us inject a value of type `a` into a container. For now, it is interesting to note that `fmap0` would be another reasonable name for `pure`:

```
pure  :: a             -> f a
fmap  :: (a -> b)      -> f a -> f b
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
```

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
```

In fact, this pattern is so common that `Control.Applicative` defines `(<$>)` as a synonym for `fmap`,

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

so that we can write

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

What about `liftA3`?

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

(Note that the precedence and associativity of `(<$>)` and `(<*>)` are actually defined in such a way that all the parentheses above are unnecessary.)

Nifty! Unlike the jump from `fmap` to `liftA2` (which required generalizing from `Functor` to `Applicative`), going from `liftA2` to `liftA3` (and from there to `liftA4`, …) requires no extra power—`Applicative` is enough.

Actually, when we have all the arguments like this we usually don't bother calling `liftA2`, `liftA3`, and so on, but just use the `f <$> x <*> y <*> z <*> ...` pattern directly. (`liftA2` and friends do come in handly for partial application, however.)

But what about `pure`? `pure` is for situations where we want to apply some function to arguments in the context of some functor `f`, but one or more of the arguments is *not* in `f`—those arguments are "pure", so to speak. We can use `pure` to lift them up into `f` first before applying. Like so:

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

Applicative examples

Maybe

Let's try writing some instances of `Applicative`, starting with `Maybe`. `pure` works by injecting a value into a `Just` wrapper; `(<*>)` is function application with possible failure. The result is `Nothing` if either the function or its argument are.

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

Let's see an example:

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

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

```
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

A *parser* is an algorithm which takes unstructured data as input (often a `ByteString`) and produces structured data as output. For example, when you load a Haskell file into `ghci`, the first thing it does is *parse* your file in order to turn it from a long `ByteString` into an *abstract syntax tree* representing your code in a more structured form. You used abstract syntax trees in Homework 3 to write an interpreter.

For the rest of the assignment, we will be using the parsing package `Attoparsec`. This is the same library that is used by `Aeson` which we used in Homework 5 to parse JSON data. `Attoparsec` has many simple parsers already defined. For example, we can write the following parser that parses a word (sequence of letters):

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

Now, let's write a parser for names. For our purposes, a name is just a capitalized word. In order to do this we will have to combine two different parsers. Namely, a parser for a single capital letter, and a parser for a sequence of lowercase ones:

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

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

Ideally, we would like to `cons` the `Word8` that is obtained by running the `upper` parser, on to the `ByteString` obtained by running `lword`. To do this we need to use a *parser combinator*. Luckily for us, `Parser` has an `Applicative` instance! Using Applicative Functors, we can apply the `cons` function inside the `Parser`:

```
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
```

Now, suppose we want to parse full names (ie, first and last) instead of just single names. Instead of returning a `ByteString`, we might want to structure the output data in some way. In this example, we will return a tuple containing the first and last names as separate `ByteString`s. Before we do this, we will need some way of skipping over the whitespace between words. We can use the following parser for this:

```
skipSpace :: Parser ()
skipSpace = skipWhile isSpace_w8
```

Note that the type of `skipSpace` is `Parser ()`. This is because we don't care about the exact value of the whitespace that we are skipping over, we just want to discard it. We will also need some way of running a parser, but not including the result in the output. The `(*>)` operator does exactly that! Now, let's write the full name parser:

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

This parser uses the `Applicative` instance for `Parser` on the `(,)` data constructor to contstruct a parser for a tuple of `ByteString`s.

This parser for names works, but it does not accomodate people who have middle names. Ideally, we would like to have a parser that can inspect the input and decide whether the name includes a middle name or not. In particular, we want to target the following data type:

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

Unfortunately, there is no way to do this using applicative functors as our combinator. The reason is that the `Applicative` interface only allows us to handle computations that take place over a *fixed* structure. For example, we could write a parser for the `TwoName` data constructor since it gets applied to a *fixed* number of arguments, but we cannot write a general parser for the `Name` data type since it can be constructed in two different ways. In order to support this sort of parsing pattern, we will have to use something stronger than applicative.

Monads to the Rescue!

Surprise! The *stronger* abstraction that we are going to use is the parser *Monad*. 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. We can think of each `Parser` as a single action which can be combined with other `Parsers` using monadic sequencing. 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)
```

This was a very simple parser, so using monads is sort of overkill. It is generally preffered to use applicative functors whenever possible. However, as we just saw, sometimes it is not possible to use applicative functors. In order to write the full name parser we need monads. The idea is simple, first we will parse two names, then we will attempt to parse a third name and decide what to do based on whether or not the third parse succeeds.

```
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.

By allowing sequencing, monads greatly increase the power of the `Parser` type. In particular, they allow decisions about the parsing computation to be made based on previous data that has been parsed.

Let's consider another example of a parser that requires the power of monads. Instead of just parsing a single name, we may need to parse a list of names. However, we may not know where the boundaries of the names are. For example, we might have the 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"]
```

Obviously, the second one is correct! Haskell Brooks Curry and Simon Peyton Jones are both very important people in the world of functional programming! But how would a computer know that? There is no way to disambiguate the parser based on the input string alone. To fix this, we will include a list of booleans stating whether or not each person in the list has a middle name. So, instead of attempting to parse the (ambiguous) string:

```
"Haskell Brooks Curry Simon Peyton Jones"
```

we will parse:

```
"[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. First, let's write some code to parse the boolean list:

```
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 arbitrary spacing. We can now use this list of `Bool`s to figure out how to parse the names. However, unlike the version of the `fullName` parser that itself decides whether to construct a `Name` using `TwoName` or `ThreeName`, we need to choose which parser to run based on the list of `Bool`s.

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

Again, we see here that the output does not have a fixed structure. Given some input string, the output is a list of arbitrary length and each element of the list has one of two *shapes* that we must choose between based on some metadata. This sort of behavior is impossible to capture using Applicative Functors; we really need to use the power of monadic sequencing.

If you would like to see more examples of monadic parsers in real-world code, check out the **Haskell Thrift Protocols**.

`Generated 2015-03-18 10:53:08.032822`