Applicative functors, Part I

CIS 194 Week 10
25 March 2012

Suggested reading:

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

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`:

```
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`,

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

so that we can write

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

What about `liftA3`?

```
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:

```
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 laws

There is only one really "interesting" law for `Applicative`:

```
f `fmap` x === pure f <*> x
```

Mapping a function `f` over a container `x` ought to give the same results as first injecting the function into the container, and then applying it to `x` with `(<*>)`.

There are other laws, but they are not as instructive; you can read about them on your own if you really want.

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.

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

Let's see an example:

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

m_phone1, m_phone2 :: Maybe String
m_phone1 = Nothing
```

```haskell
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
```

Generated 2013-04-04 15:15:39.515041