Middle School Algebra with Applicative Functors

Rushi Shah

25 February 2016

What are applicative functors? And how are they actually useful? After reading the Typeclassopedia, and Week 10's lecture for CIS194, I started to formulate a way of explaining it to myself in terms of basic algebra.

1 Factoring Review

Remember back in middle school when you learned the Distributive Property? To remind you of a simple example:

$$2x + 2y = 2(x + y)$$

If you are dividing by $(x + y)$, you can't use the equation in its $2x + 2y$ form, you have to factor it first, cancel the terms, and then wrap the result (which would just be 1 in this case) by multiplying by two.

$$2x + 2y * \frac{1}{x+y}$$
$$2(x + y) * \frac{1}{x+y}$$
$$2 * 1$$

If you think about it, 2 is acting like an applicative functor here. You have two terms that are wrapped in a Two, and you want to apply a function on the wrapped values (ignoring the wrapper) until the very end, at which point you slap the wrapper back on.

2 The `Two` Functor

To see what I mean, start by defining a type for Two. This type is clearly an instance of Functor, `fmap` would just apply the function to whatever 2 is being multiplied by.

```
data Two a = Two a
           deriving (Show, Eq)

instance Functor Two where
  fmap g (Two z) = Two (g z)
```

3 The Two Applicative

It is also an instance of Applicative, where `pure` just multiplies by 2. The only other thing we need to define is `<*>`. Given two separate terms which are both multiplied by two, we would factor out the two and return the two other terms, right? The complete applicative instance for `Two` is given below:

```haskell
instance Applicative Two where
  pure z = Two z
  (Two x) <*> (Two y) = Two (x y)
```

Its important to note that the applicative instance does not actually do anything with x and y. All it does is factor out the two. When the instance is actually used, x and y will be passed to the function as arguments, and whoever is calling the function can decide how to combine them.

4 Using the Two Applicative

Now, like you learned in middle school algebra, if you want to add two things that are both multiplied by two, you can add them both together and just multiply the result by two. In other words if you factor $2x + 2y$, you'll pull out the 2 (which our applicative does), and then multiply that 2 by $x + y$. This lets us make Two a partial instance of `Num` (if we enable FlexibleInstances)

```haskell
instance Num (Two String) where
  (+) a b = (\x y -> (x ++ "+" ++ y)) `fmap` a <*> b
  --Two "x" + Two "y" -> Two "x+y"
```

5 Using the Two Functor

To finish off the example I started earlier, let's define a quick function for canceling two like terms. This function, because it will presumably be used in other contexts other than just for this contrived example, will operate on any term (even if they aren't wrapped in a two).

```haskell
cancel :: String -> String -> String
cancel t1 t2 = if t1 == t2
               then "1"
               else ("(" ++ t2 ++ ")/(" ++ t1 ++ ")")
```

But if we go to apply this function (`cancel "x+y"`) to the results of our `factorOutTwo (Two "x") (Two "y")`, we run into an issue because we expect a type of `String` but we actually have something of type `Two String`. No worries, because before we could define an Applicative instance, we had to define a Functor instance. Now, we can just fmap the function like so:

```
fmap (cancel "x+y") ((Two "x") + (Two "y"))
--Two "1"

fmap (cancel "x") ((Two "x") + (Two "y"))
--Two "(x+y)/(x)"
```

6 Conclusion

Hopefully that clarified how Applicative Functors work with a familiar example. Even if it didn't, it helped me wrap my head around them so ¯_(ツ)_/¯.