Applicative functors, Part II

CIS 194 Week 11
1 April 2012

Suggested reading:

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

We begin with a review of the `Functor` and `Applicative` type classes:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

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

Every `Applicative` is also a `Functor`—so can we implement `fmap` in terms of `pure` and `(<*>)`? Let's try!

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

Well, that has the right type at least! However, it's not hard to imagine making `Functor` and `Applicative` instances for some type such that this equality does not hold. Since this would be a fairly dubious situation, we stipulate as a *law* that this equality must hold—this is a formal way of stating that the `Functor` and `Applicative` instances for a given type must "play nicely together".

Now, let's see a few more examples of `Applicative` instances.

More Applicative Examples

Lists

How about an instance of `Applicative` for lists? There are actually two possible instances: one that matches up the list of functions and list of arguments elementwise (that is, it "zips" them together), and one that combines functions and arguments in all possible ways.

First, let's write the instance that does all possible combinations. (For reasons that will become clear next week, this is the default instance.) From this point of view, lists represent nondeterminism: that is, a value of type `[a]` can be thought of as a single value with multiple possibilities. Then `(<*>)` corresponds to nondeterministic function application—that is, the application of a nondeterministic function to a nondeterministic argument.

```
instance Applicative [] where
  pure a        = [a]          -- a "deterministic" value
  [] <*> _      = []
  (f:fs) <*> as = (map f as) ++ (fs <*> as)
```

Here's an example:

```
names  = ["Joe", "Sara", "Mae"]
phones = ["555-5555", "123-456-7890", "555-4321"]

employees1 = Employee <$> names <*> phones
```

Maybe this particular example doesn't make that much sense, but it's not hard to imagine situations where you want to combine things in all possible ways like this. For example, we can do nondeterministic arithmetic like so:

```
(.+) = liftA2 (+)    -- addition lifted to some Applicative context
(.*) = liftA2 (*)    -- same for multiplication
```

```haskell
-- nondeterministic arithmetic
n = ([4,5] .* pure 2) .+ [6,1] -- (either 4 or 5) times 2, plus either 6 or 1

-- and some possibly-failing arithmetic too, just for fun
m1 = (Just 3 .+ Just 5) .* Just 8
m2 = (Just 3 .+ Nothing) .* Just 8
```

Next, let's write the instance that does elementwise combining. First, we must answer an important question: how should we handle lists of different lengths? Some thought reveals that the most sensible thing to do is to truncate the longer list to the length of the shorter, throwing away the extra elements. Of course there are other possible answers: we might, for instance, extend the shorter list by copying the last element (but then what do we do when one of the lists is empty?); or extend the shorter list with a "neutral" element (but then we would have to require an instance of `Monoid`, or an extra "default" argument for the application).

This decision in turn dictates how we must implement `pure`, since we must obey the law

```haskell
pure f <*> xs === f <$> xs
```

Notice that the right-hand side is a list with the same length as `xs`, formed by applying `f` to every element in `xs`. The only way we can make the left-hand side turn out the same… is for `pure` to create an infinite list of copies of `f`, because we don't know in advance how long `xs` is going to be.

We implement the instance using a `newtype` wrapper to distinguish it from the other list instance. The standard Prelude function `zipWith` also comes in handy.

```haskell
newtype ZipList a = ZipList { getZipList :: [a] }
  deriving (Eq, Show, Functor)

instance Applicative ZipList where
  pure = ZipList . repeat
  ZipList fs <*> ZipList xs = ZipList (zipWith ($) fs xs)
```

An example:

```haskell
employees2 = getZipList $ Employee <$> ZipList names <*> ZipList phones
```

Reader/environment

Let's do one final example instance, for `(->) e`. This is known as the *reader* or *environment* applicative, since it allows "reading" from the "environment" `e`. Implementing the instance is not too hard, we just have to use our nose and follow the types:

```haskell
instance Functor ((->) e) where
  fmap = (.)

instance Applicative ((->) e) where
  pure = const
  f <*> x = \e -> (f e) (x e)
```

An `Employee` example:

```haskell
data BigRecord = BR { getName         :: Name
                    , getSSN          :: String
                    , getSalary       :: Integer
                    , getPhone        :: String
                    , getLicensePlate :: String
                    , getNumSickDays  :: Int
                    }

r = BR "Brent" "XXX-XX-XXX4" 600000000 "555-1234" "JGX-55T3" 2

getEmp :: BigRecord -> Employee
getEmp = Employee <$> getName <*> getPhone

ex01 = getEmp r
```

Aside: Levels of Abstraction

`Functor` is a nifty tool but relatively straightforward. At first glance it seems like `Applicative` doesn't add that much beyond what `Functor` already provides, but it turns out that it's a small addition with a huge impact. `Applicative` (and as we will see next week, `Monad`) deserves to be called a "model of computation", while `Functor` doesn't.

When working with things like `Applicative` and `Monad`, it's very important to keep in mind that there are *multiple levels of abstraction* involved. Roughly speaking, an *abstraction* is something which *hides details* of a lower level, providing a "high-level" interface that can be used (ideally) without thinking about the lower level—although the details of the lower level often "leak through" in certain cases. This idea of layers of abstraction is widespread. Think about user programs—OS—kernel—integrated circuits—gates—silicon, or HTTP—TCP—IP—Ethernet, or programming languages—bytecode—assembly—machine code. As we have seen, Haskell gives us many nice tools for constructing multiple layers of abstraction *within Haskell programs themselves*, that is, we get to dynamically extend the "programming language" layer stack upwards. This is a powerful facility but can lead to confusion. One must learn to explicitly be able to think on multiple levels, and to switch between levels.

With respect to `Applicative` and `Monad` in particular, there are just two levels to be concerned with. The first is the level of implementing various `Applicative` and `Monad` instances, *i.e.* the "raw Haskell" level. You gained some experience with this level in your previous homework, when you implemented an `Applicative` instance for `Parser`.

Once we have an `Applicative` instance for a type like `Parser`, the point is that we get to "move up a layer" and program with `Parsers` *using the `Applicative` interface*, without thinking about the details of how `Parser` and its `Applicative` instance are actually implemented. You got a little bit of experience with this on last week's homework, and will get a lot more of it this week. Programming at this level has a very different feel than actually implementing the instances. Let's see some examples.

The Applicative API

One of the benefits of having a unified interface like `Applicative` is that we can write generic tools and control structures that work with *any* type which is an instance of `Applicative`. As a first example, let's try writing

```haskell
pair :: Applicative f => f a -> f b -> f (a,b)
```

`pair` takes two values and pairs them, but all in the context of some `Applicative f`. As a first try we can take a function for pairing and "lift" it over the arguments using `(<$>)` and `(<*>)`:

```haskell
pair fa fb = (\x y -> (x,y)) <$> fa <*> fb
```

This works, though we can simplify it a bit. First, note that Haskell allows the special syntax `(,)` to represent the pair constructor, so we can write

```haskell
pair fa fb = (,) <$> fa <*> fb
```

But actually, we've seen this pattern before—this is the `liftA2` pattern which got us started down this whole `Applicative` road. So we can further simplify to

```haskell
pair fa fb = liftA2 (,) fa fb
```

but now there is no need to explicitly write out the function arguments, so we reach our final simplified version:

```haskell
pair = liftA2 (,)
```

Now, what does this function do? It depends, of course, on the particular `f` chosen. Let's consider a number of particular examples:

- `f = Maybe`: the result is `Nothing` if either of the arguments is; if both are `Just` the result is `Just` their pairing.
- `f = []`: `pair` computes the Cartesian product of two lists.
- `f = ZipList`: `pair` is the same as the standard `zip` function.
- `f = IO`: `pair` runs two `IO` actions in sequence, returning a pair of their results.
- `f = Parser`: `pair` runs two parsers in sequence (the parsers consume consecutive sections of the input), returning their results as a pair. If either parser fails, the whole thing fails.

Can you implement the following functions? Consider what each function does when `f` is replaced with each of the above types.

```haskell
(*>)       :: Applicative f => f a -> f b -> f b
mapA       :: Applicative f => (a -> f b) -> ([a] -> f [b])
sequenceA  :: Applicative f => [f a] -> f [a]
replicateA :: Applicative f => Int -> f a -> f [a]
```