Brightcove is looking for a Software Engineer -- Security and Scalability (clojure) in *Boston, MA*

Fri 11 Jan 2013

Algebras of Applicatives

Posted by Gershom Bazerman under Uncategorized
No Comments

While the previous post in this series was relatively immediately applicable, this one has constructions I definitely wouldn't recommend in production code. However, they do take us further in exploring the universe of applicative functors, and, more broadly, exploring which data types provide which properties by construcion.

It's well known that if you have any Functor `F a`, you can take its "fixpoint", creating a structure of infinitely nested Fs, like so. `F (F (F (...) ) )` Since we can't have infinite types directly in Haskell, we introduce the Fix newtype:

```
newtype Fix f = Fix (f (Fix f))
```

This "wraps up" the recursion so that GHC accepts the type. `Fix f` is a `Fix` constructor, containing an "f" of `Fix f` inside. Each in turn expands out, and soforth. Fixpoints of functors have fixedpoints of functors inside 'em. And so on, and so on, ad infinitum.

(Digression: We speak of "algebraic data types" in Haskell. The "algebra" in question is an "F-algebra", and we can build up structures with fixpoints of functors, taking those functors as initial or terminal objects and generating either initial algebras or terminal coalgebras. These latter two concepts coincide in Haskell in the Fix description given above, as greatest and least fixpoints of data types in Haskell turn out to be the same thing. For more background, one can go to Wadler's "Recursive Types for Free," or Jacobs and Rutten's "Tutorial on (Co)Algebras and (Co)Induction" for starters.)

The family of functors built from our friends Const, Sum, Product, and Reader (exponentiation) are known as Polynomial Functors. If we take closure of these with a proper fixpoint construct (that lets us build infinite structures), we get things that are variously known as Containers, Shapely Types, and Strictly Positive types.

One irritating thing is that the fixpoint of a functor as we've written it is no longer itself a functor. The type constructor Fix is of kind `(* -> *) -> *`, which says it takes an "f" which takes one argument (e.g. "Maybe" or "Identity" or etc.) and returns a proper type (i.e. a value at the type level of kind *).

We want a fixpoint construction that gives back something of kind `* -> *` — i.e. something that is a type constructor representing a functor, and not just a plain old type. The following does the trick.

```
newtype FixF f a = FixF (f (FixF f) a)
deriving instance (Show (f (FixF f) a)) => Show (FixF f a)
```

(I learned about FixF from a paper by Ralf Hinze, but I'm sure the origins go back much further).

FixF is of kind `((* -> *) -> * -> *) -> * -> *`. It takes the fixpoint of a "second-order Functor" (a Functor that sends a Functor to another Functor, i.e. an endofunctor on the functor category of hask), to recover a standard "first order Functor" back out. This sounds scary, but it isn't once you load it up in ghci and start playing with it. In fact, we've encountered second order functors just recently. Product, Sum, and Compose are all of kind `(* -> *) -> (* -> *) -> * -> *`. So they all send two functors to a third functor. That means that `Product Const`, `Sum Identity` and `Compose Maybe` are all second-order functors, and things appropriate to take our "second-order fixpoint" of.

Conceptually, "Fix f" took a value with one hole, and we filled that hole with "Fix f" so there was no room for a type parameter. Now we've got an "f" with two holes, the first of which takes a functor, and the second of which is the hole of the resulting functor.

Unlike boring old "Fix", we can write Functor and Applicative instances for "FixF", and they're about as simple and compositional as we could possibly hope.

```
instance Functor (f (FixF f)) => Functor (FixF f) where
    fmap f (FixF x) = FixF $ fmap f x

instance Applicative (f (FixF f)) => Applicative (FixF f) where
    pure x = FixF $ pure x
    (FixF f) < *> (FixF x) = FixF (f < *> x)
```

But now we run into a new problem! It seems like this "a" parameter is just hanging out there, doing basically nothing. We take our classic functors and embed them in there, and they still only have "one hole" at the value level, so don't actually have any place to put the "a" type we now introduced. For example, we can write the following:

```
-- FixF . Compose . Just . FixF . Compose $ Nothing
-- > FixF (Compose (Just (FixF (Compose Nothing))))
-- :t FixF (Compose (Just (FixF (Compose Nothing))))
-- > FixF (Compose Maybe) a
```

We now introduce one new member of our basic constructions — a second order functor that acts like "const" on the type level, taking any functor and returning Identity.

```
data Embed (f :: * -> *) a = Embed a deriving (Show)

instance Functor (Embed f) where
    fmap f (Embed x) = Embed $ f x

instance Applicative (Embed f) where
    pure x = Embed x
    (Embed f) < *> (Embed x) = Embed (f x)
```

Now we can actually stick functorial values into our fixpoints:

```
-- FixF $ Embed "hi"
-- > FixF (Embed "hi")

-- fmap (++ " world") $ FixF (Embed "hi")
-- > FixF (Embed "hi world")

-- FixF . Product (Embed "hi") .
--        FixF . Product (Embed "there") . FixF $ undefined
-- > FixF (Product (Embed "hi")
--   (FixF (Product (Embed "there")
--   (FixF *** Exception: Prelude.undefined
```

You may have noticed that we seem to be able to use "product" to begin a chain of nested fixpoints, but we don't seem able to stick a "Maybe" in there to **stop** the chain. And it seems like we're ot even "fixing" where we intend to be:

```
-- :t FixF . Product (Embed "hi") . FixF . Product (Embed "there") . FixF $ un
-- > FixF (Product (Embed f)) [Char]
```

That's because we're applying Product, takes and yields arguments of kind `* -> *` in a context where we really want to take and yield second-order functors as arguments — things of kind `(* -> *) -> * -> *`. If we had proper kind polymorphism, "Product" and "ProductF" would be able to collapse (and maybe, soon, they will). But at least in the ghc 7.4.1 that I'm working with, we have to write the same darn thing, but "up a kind".

```
data ProductF f g (b :: * -> *) a =
      ProductF (f b a) (g b a) deriving Show

instance (Functor (f b), Functor (g b)) =>
         Functor (ProductF f g b) where
                fmap f (ProductF x y) = ProductF (fmap f x) (fmap f y)

instance (Applicative (f b), Applicative (g b)) =>
         Applicative (ProductF f g b) where
               pure x = ProductF (pure x) (pure x)
              (ProductF f g) < *> (ProductF x y) = ProductF (f < *> x) (g < *>
```

We can now do the following properly.

```
yy = FixF . ProductF (Embed "foo") $ InL $ Const ()
xx = FixF . ProductF (Embed "bar") . InR .
        FixF . ProductF (Embed "baz") . InL $ Const ()
```

So we've recovered proper lists in Haskell, as the "second-order fixpoint" of polynomial functors. And the types look right too:

```
-- :t yy
-- > FixF (ProductF Embed (Sum (Const ()))) [Char]
```

Because we've built our Applicative instances compositionally, we have an applicative for our list list construction automatically:

```
-- (++) < $> yy < *> xx
-- > FixF (ProductF (Embed "foobar") (InL (Const ())))
```

This is precisely the "ZipList" applicative instance. In fact, our applicative instances from this "functor toolkit" are all "zippy" — matching up structure where possible, and "smashing it down" where not. This is because Sum, with its associated natural transformation logic, is the only way to introduce a disjoint choice of shape. Here are some simple examples with Sum to demonstrate:

```
-- liftA2 (++) (InL (Identity "hi")) $
--      InR (Product (Identity " there") (Const ([12::Int])))
-- > InL (Identity "hi there")
-- liftA2 (++) (InR (Identity "hi")) $ InL (Product (Identity " there") (Const
-- > InL (Product (Identity "hi there") (Const [12]))
```

We're always "smashing" towards the left. So in the first case, that means throwing away half of the pair. In the second case, it means injecting Const mempty into a pair, and then operating with that.

In any case, we now have infinite and possibly infinite branching structures. And not only are they Functors, but they're also Applicatives, and in a way that's uniform and straightforward to reason about.

In the next post, we'll stop building out our vocabulary of "base" Functors (though it's not quite complete) and instead push forward on what else these functors can provide "beyond" Applicative.

LEAVE A REPLY

Name (required)

Mail (will not be published) (required)

Website

Simplify log(exp(12))

Submit Comment

Archived Entry

Post Date :
Friday, Jan 11th, 2013 at 12:03 pm
Category :
Uncategorized
Do More :
You can leave a response, or trackback from your own site.

Wiki

Old Journal
Source
Harmless Algorithms

Tag Cloud

Search

Search

Ads

RSS Feeds

Posts | Comments