The Comonad.Reader
types, (co)monads, substructural logic

Thu 2 May 2013

Representing Applicatives

Posted by Gershom Bazerman under Uncategorized

No Comments

In the previous two posts, we've built up a whole range of applicatives, out of Const, Identity, Reader, Compose, Product, Sum, and Fix (and some higher-order analogues). Sum has given us the most trouble, but in some sense has been the most powerful, letting us write things like possibly eventually terminating lists, or trees, or in fact any sort of structure with branching alternatives. In this post, I want to think a bit more about why it is that Sum is the trickiest of the bunch, and more generally, what we can say about when two applicative structures are the "same". In the process of doing so, we'll invent something a lot like Traversable en passant.

Let's do some counting exercises. `Product Identity Identity` holds exactly two things. It is therefore isomorphic to `((->) Bool)`, or if we prefer, `((->) Either () ())`. That is to say that a pair that *holds* two values of type `a` is the same as a function that *takes a two-valued type* and *yields* a value of type `a`. A product of more functors in turn is isomorphic to the reader of the sum of each of the datatypes that "represent" them. E.g. `Product (Product Identity Identity) (Product (Const ()) Identity)` is iso to `((->) (Either (Either () ()) ()))`, i.e. a data type with three possible inhabitants. In making this move we took Product to Either -- multiplication to sum. We can pull a similar trick with Compose. `Compose (Product Identity Identity) (Product Identity Identity)` goes to ((->) (Either () (),Either () ())). So again we took Product to a sum type, but now we took Compose to a pair -- a product type! The intuition is that composition *multiplies* the possibilities of spaces in each nested functor.

Hmm.. products go to sums, composition goes to multiplication, etc. This should remind us of something -- these rules are exactly the rules for working with exponentials. x^n * x^m = x^(n + m). (x^n)^m = x^(n*m). x^0 = 1, x^1 = x.

Seen from the right standpoint, this isn't surprising at all, but almost inevitable. The functors we're describing are known as "representable," a term which derives from category theory. (See appendix on representable functors below).

In Haskell-land, a "representable functor" is just any functor isomorphic to the reader functor `((->) a)` for some appropriate a. Now if we think back to our algebraic representations of data types, we call the arrow type constructor an exponential. We can "count" `a -> x` as x^a, since e.g. there are 3^2 distinct functions that inhabit the type 2 -> 3. The intuition for this is that for each input we pick one of the possible results, so as the number of inputs goes up by one, the number of functions goes up by multiplying through by the set of possible results. 1 -> 3 = 3, 2 -> 3 = 3 * 3, (n + 1) -> 3 = 3 * (n -> 3).

Hence, if we "represent" our functors by exponentials, then we can work with them directly as exponentials as well, with all the usual rules. Edward Kmett has a library encoding representable functors in Haskell.

Meanwhile, Peter Hancock prefers to call such functors "Naperian" after John Napier, inventor of the logarithm (See also here). Why Naperian? Because if our functors are isomorphic to exponentials, then we can take their logs! And that brings us back to the initial discussion of type mathematics. We have some functor F, and claim that it is isomorphic to -^R for some concrete data type R. Well, this means that R is the logarithm of F. E.g. `(R -> a, S -> a) =~ Either R S -> a`, which is to say that if log F = R and log G =~ S, then log (F * G) = log F + log G. Similarly, for any other data type n, again with log F = R, we have `n -> F a =~ n -> R -> a =~ (n * R) -> a`, which is to say that log (F^n) =~ n * log F.

This gives us one intuition for why the sum functor is not generally representable -- it is very difficult to decompose log (F + G) into some simpler compound expression of logs.

So what functors are Representable? Anything that can be seen as a fixed shape with some index. Pairs, fixed-size vectors, fixed-size matrices, any nesting of fixed vectors and matricies. But also infinite structures of regular shape! However, not things whose shape can vary -- not lists, not sums. Trees of fixed depth or infinite binary trees therefore, but not trees of arbitrary depth or with ragged structure, etc.

Representable functors turn out to be extremely powerful tools. Once we know a functor is representable, we know exactly what its applicative instance must be, and that its applicative instance will be "zippy" -- i.e. acting pointwise across the structure. We also know that it has a monad instance! And, unfortunately, that this monad instance is typically fairly useless (in that it is also "zippy" -- i.e. the monad instance on a pair just acts on the two elements pointwise, without ever allowing anything in the first slot to affect anything in the second slot, etc.). But we know more than that. We know that a representable functor, by virtue of being a reader in disguise, cannot have effects that migrate outwards. So any two actions in a representable functor are commutative. And more than that, they are entirely independent.

This means that all representable functors are "distributive"! Given any functor f, and any data type r, then we have

```
distributeReader :: Functor f => f (r -> a) -> (r -> f a)
distributeReader fra = \r -> fmap ($r) fra
```

That is to say, given an arrow "inside" a functor, we can always pull the arrow out, and "distribute" application across the contents of the functor. A list of functions from `Int -> Int` becomes a single function from `Int` to a list of `Int`, etc. More generally, since all representable functors are isomorphic to reader, given g representable, and f any functor, then we have: `distribute :: (Functor f, Representable g) => f (g a) -> g (f a)`.

This is pretty powerful sauce! And if f and g are *both* representable, then we get the transposition isomorphism, witnessed by `flip`! That's just the beginning of the good stuff. If we take functions and "unrepresent" them back to functors (i.e. take their logs), then we can do things like move from `((->) Bool)` to pairs, etc. Since we're in a pervasively lazy language, we've just created a library for memoization! This is because we've gone from a function to a data structure we can index into, representing each possible argument to this function as a "slot" in the structure. And the laziness pays off because we only need to evaluate the contents of each slot on demand (otherwise we'd have a precomputed lookup table rather than a dynamically-evaluated memo table).

And now suppose we take our representable functor in the form `s -> a` and paired it with an "index" into that function, in the form of a concrete `s`. Then we'd be able to step that `s` forward or backwards and navigate around our structure of `a`s. And this is precisely the Store Comonad! And this in turn gives a characterization of the lens laws.

What this all gives us a tiny taste of, in fact, is the tremendous power of the Yoneda lemma, which, in Haskell, is all about going between values and functions, and in fact captures the important universality and uniqueness properties that make working with representable functors tractable. A further tiny taste of Yoneda comes from a nice blog post by Conal Elliott on memoization.

Extra Credit on Sum Functors

There in fact is a log identity on sums. It goes like this:

```
log(a + c) = log a + log (1 + c/a)
```

Do you have a useful computational interpretation of this? I've got the inklings of one, but not much else.

Appendix: Notes on Representable Functors in Hask.

The way to think about this is to take some arbitrary category C, and some category that's basically Set (in our case, Hask. In fact, in our case, C is Hask too, and we're just talking about endofunctors on Hask). Now, we take some functor F : C -> Set, and some A which is an element of C. The set of morphisms originating at A (denoted by Hom(A,-)) constitutes a functor called the "hom functor." For any object X in C, we can "plug it in" to Hom(A,-), to then get the set of all arrows from A to X. And for any morphism X -> Y in C, we can derive a morphism from Hom(A,X) to Hom(A,Y), by composition. This is equivalent to, in Haskell-land, using a function `f :: x -> y` to send `g :: a -> x` to `a -> y` by writing "functionAToY = f . g".

So, for any A in C, we have a hom functor on C, which is C -> Set, where the elements of the resultant Set are homomorphisms in C. Now, we have this other arbitrary functor F, which is also C -> Set. Now, if there is an isomorphism of functors between F, and Hom(A,_), then we say F is "representable". A representable functor is thus one that can be worked with entirely as an appropriate hom-functor.