Some Dependent Types and The Shit We've Done With Them

Gordon, Josh   Brandt, Eliza

January 29, 2018

Abstract

This document is written by Josh, and the words "I" and "Me" shall refer to him. I became more active on my math blog, and I mentioned to some dude the derivative of a type concept. He pointed me to Eliza, and we talked a whole lot about dependent types, among other things. This document serves to organize that work, and hopefully point out any logical errors we've made along the way. Additionally, it's gonna be a good reference for me, because I constantly forget the definitions of things like ICofree.

Contents

1 Conventions Used In This Paper

Haskell syntax is used to define concepts. Not all code is valid Haskell, however, as we are working entirely on the type level, and Haskell does not always supply nice ways to do that.

2 The Natural Numbers

This is where it starts. A type level definition of the natural numbers.

```
data Nat = Z | S Nat
```

Peano numbers are used for our purposes here. There isn't much relevant depth here; all these are used for is indexing.

3 Vec

```
data Vec (n :: Nat) a where
        VNil :: Vec 'Z a
        VCons :: a -> Vec n a -> Vec ('S n) a
```

This, as you might imagine, represents a list of a's of length n. This is useful for the reason that having n of something is sometimes much more convenient than only having one of that something.

4 IEither

This is another example of a way to raise something to the type level.

```
data IEither n a b where
  ILeft :: a -> IEither Z a b
  IRight :: b -> IEither (S n) a b
```

5 ICofree

5.1 Cofree

```
data Cofree f a = a :< f (Cofree f a)
```

Cofree is a way to turn a functor into a comonad, and a comonad is a way to capture a thing that usually has a context, and apply mapping operations that depend on context to all contexts of that thing.

5.2 Indexed Cofree

For our purposes here, we need to bring this onto the type level. This is done with a similar strategy used in Vec, and indexing possible states by natural numbers. Hence:

```
data ICofree (n :: Nat) f a where
  (:<) :: a -> f n (ICofree n f a) -> ICofree ('S n) f a
```

List is to Vec as Cofree is to ICofree.
This cofree works at the type level, and generates a tree-like structure where the n-th layer has n nodes. This is useful for representing states in a system.

6 Trees

6.1 Binary Trees

We can make a binary tree on the type level by using indices backwards. A zero index means the node is at the bottom of the tree, a.k.a a leaf.

```
data BinTree n a where
  Leaf :: BinTree Z a
  Node :: a -> BinTree n a -> BinTree n a -> BinTree (S n) a
```

6.2 K-Ary Trees

Why stop at 2?

```
data Kary n k a where
  Leaf :: Kary Z k a
  Node :: a -> Vec k (Kary n k a) -> Kary (S n) k a
```

Using Vec, we can make a k-ary tree.

6.3 Generalized K-ary Tree-Like Structures

Using a Vec to store nodes is all well and good, but why not allow any functor? This results in:

```
data GenKary f n k a where
  Leaf :: GenKary f Z k a
  Node :: a -> f k (GenKary f n k a) -> GenKary f (S n) k a
```

Now, the Node here looks a lot like the :< constructor for ICofree. This leads to our next generalization.

7 G

7.1 The Definition

One thing that has remained constant in all of these generalizations is the induction term. n has been transformed to S n, and k has remained k. Why not allow this to be determined by a parameter? This results in the monstrous type G.

```
data G f g h n k a where
  Node :: a -> f k (G f g h n k a) -> G f g h (g n) (h k) a
```

g and h are type functions that transform n and k respectively.

7.2 Defining Other Structures In Terms Of G

G, being so general, encapsulates nearly everything else defined here before it. Below are some examples:

```
type BinTree n a = G Vec S Id n 2 a
type KaryTree n k a = G Vec S Id n k a
type GenKary f n k a = G f S Id n k a
type ICofree f n a = G f S S n n a
```

8 Isomorphisms of G

With such a general structure, there are many interesting symmetries to be discovered involving it.

8.1 Inductive G And Offset Structures

When working with isomorphisms of G, in many cases, all one cares about is the first induction term. Thus, we define a shorthand

```
IG(someInductionTerm) = G f someInductionTerm h n k a
```

Now, one thing that has remained constant this entire time is the base case of the recursion for Nats. Z has always been the starting point. Here, we introduce the new concept of an offset indexed structure. An offset indexed structure is any type parameterized by a Nat where it's base case is not Z. For IG, an offset will be given as the second parameter. IG$(F(n), c)$ represents a G with induction term F and offset c where F is a type function and c is a constant.

At this point, the Haskell validity begins to break down. We will consider induction terms such as $x + a$ or x^2.

8.2 Significance of the induction term

What the induction term $f(x)$ means is that the G will only contain values of a such that their indices exist in the codomain of f. An induction term of S covers all of your bases; every natural number except zero can be said to be an output of the successor function, and you can start with zero. If our induction term was S . S, or +2 as I will begin to write it, only elements

with indices of the same parity as the starting index would be contained in the G.
Only one induction term matters for this, as `(G f g h n k a)` is isomorphic to `(G f h g k n a)`

From now on, this will be written as `Iso (G f g h n k a) (G f h g k n a)`

8.3 Additive IGs

When the induction term is of the form $x + a$, we get a G that holds every ath element. By having a of these IGs, each offset , we can get a structure that has an element at every index. Thus:

```
Iso IG(S) (Vec a IG(+a,c))
```

where c is the index of the IG in the Vector.