Type Level Shenanigans

1 What Is `List`

In haskell, we often talk about `List` or `Maybe`. But in system F it's not entirely clear how to encode this. We talked about encoding data types in System F using the church encoding last time. But when we think about

```
data Mabye t = Just t | Nothing
```

We haven't really talked about what to do with that `t`. It should, more-or-less look like

$$\forall \alpha, (t \to \alpha) \to \alpha \to \alpha$$

but `t` isn't a type. And we don't really have a sensible notion of an "open type". One natural choice is to put a quantifier around it.

$$\forall \beta, \forall \alpha, (\beta \to \alpha) \to \alpha \to \alpha$$

This is an OK choice, but it's a bit fishy. In particular we also talk about Maybe as it applies to thing. Such as `Maybe Unit`. However, **there is not type level application in system F**, so we can't write

$$\big(\forall \beta, \forall \alpha, (\beta \to \alpha) \to \alpha \to \alpha\big)\big(\forall \alpha, \alpha \to \alpha\big)$$

However, we can write `Maybe\, Unit` directly, as

$$\forall \alpha, \big(\big(\forall \alpha, \alpha \to \alpha\big) \to \alpha\big) \to \alpha \to \alpha\big)$$

But that's not really the type `Maybe` applied to the type `Unit`. It's a weird abomination, but you know what I mean. Let's start by explaining to System F what we mean.

2 F_1

We will add two features to our language. It may shock you to learn that it's going to be a pair of abstraction and application. First is the abstraction. The concept is called a universe. The universes are going to be tiered. We will start with the 0, which is just the set of system F types. We introduce a helper concept called a kind, and say they all have kind $*$. Then we add a new universe, and a new kind $* \to *$. These are the set of terms

$$\alpha \Rightarrow t$$

such that $\forall \alpha, t$ is kind $*$. In other words t is an open term with one free variable α.

The second thing is a type-level application, so given $\alpha \Rightarrow T : * \to *$, and a $t : *$, we define $T\,a$ as $T[\alpha := t]$.

This already lets us give meaning to `Maybe`. We can now define a kind $* \to *$ as

$$Maybe := \beta \Rightarrow \forall \alpha, (\beta \to \alpha) \to \alpha \to \alpha$$

In haskell terms, this setup eta expands everything, so `Maybe` is just shorthand for $\alpha \Rightarrow$ `Maybe` α. Of course haskell doesn't have type-level lambdas, so this is all a bit annoying to notate.

But now we can do all sorts of fun things. We can define functors, monads, traversable, and that whole family of behaviors. But we can't define monad transformers. A monad transformer takes a monad, and returns a monad. In other words it takes a $* \to *$ and returns a $* \to *$. This is illegal since

1. there are only two universes in F_1, just $*$ and $* \to *$.
2. Certainly this isn't a member of $*$, since it contains parts of F_1.
3. it can't be in F_1, since members of F_1 can only take members of $*$ as arguments.

So we need to generalize to F_2. Let's do this *to exhaustion*.

3 F_ω

The idea is to define F_{n+1} as follows. The universe $n+1$ is the term terms

$$\alpha \Rightarrow t$$

such that for any k in universe $\leq n$, $t[\alpha := k]$ is in universe n. So now we can build $(* \to *) \to (* \to *)$, which is the type of monad transformers. In fact, we can build all sorts fun things. Note that the distinction between kinds and universes shows up here, since

$$* \to (* \to *) \quad \text{and} \quad (* \to *) \to (* \to *)$$

are both in universe 2. Then we define higher-kinded type application the same way. Given kinds k_1 and k_2, and types $t : k_1 \to k_2$ and $t' : k_1$, we define $t\,t'$ as $t[\alpha := t']$. And then a simple induction proves that this has kind k_2.

So finally, we can define F_ω as the the union of F_n. Or, more formally for any category theorist, we observe that F_n embeds in F_{n+1}. So we can take a categorical limit.

The upshot is we can now define *any* type level operator we want, with parameters coming from anywhere. But we have to be careful. What happens to type-checking? What happens to inference? Is this confluent, normalizing, etc?

1. Hindle-Milner$_\omega$ type checks the same way. Basically any term only has finitely many universes, so it's not that bad.
2. Of course things normalize, we haven't changed any terms! All of the terms of System F have types whose kind is $*$.
3. There are still theorems for free. Since all terms of of type $*$ this isn't interesting.

4 Church Encodings in F_ω

Last time we saw church encodings for $*$ kinded types with no recursion. This generalizes easily to $k_1 \to *$, like we saw with `Maybe`.

But we can now finally address the question of recursive ADTs.

The first thing we do is **continuation passing transform at the type level**. That is, we will first perform a "haskell-to-haskell" data transform. Add an extra type parameter, and replace every recursive call with that parameter.

```
List t = Emp | t : List t   ⟶   ListK t r = Emp | t : r
```

Now we can encode `ListK` the standard way

$$ListK := t \Rightarrow r \Rightarrow \forall \alpha, \alpha \to (t \to r \to \alpha) \to \alpha$$

The short version is to replace r with α. If this is a bit unsatisfying, that ok. The interesting thing is that the result from before holds. If you add the corresponding constructors and elimination rules to System F, you'll find an isomorphism.

5 Folding over anything

This leads to a series of important features in system F_ω. The classic church numerals are now well-motivated. That is

$$\forall\alpha, (\alpha \to \alpha) \to \alpha \to \alpha$$

is exactly the encoding of

```
data Nat = S nat | Zero
```

So system F knows about arithmetic. In fact for any of these structures, the church encodings define "foreach loops" that respect the structure of the data. They are guaranteed to terminate. So we have loops, but also termination guarantees. This leads to the rather critical code design point: don't use general recursive when a church encoding will suffice.

We can also now talk about the haskell type `Functor`. Given a $t : * \to *$, we equip it with an

$$fmap : \forall\alpha, \forall\beta, (\alpha \to \beta) \to t\alpha \to t\beta$$

Along with laws that

$$fmap\ id = id \quad \text{and} \quad fmap(f \circ g) = fmap\ f \circ fmap\ g$$

Oddly, these are exactly the theorems for free for t.

6 A Moment of Reflection

We have now pretty much defined the core of the ML/Haskell languages. We've build the main objects of study in lambda calculus, although many more extensions are possible and interesting. And while we often talk about trading off expressiveness for analyzability, we have broken that down along several dimensions

1. Confluence (I want evaluation to have a unique meaning)
2. Normalization (I want things to terminate)
3. Loops (I want to iterate over things)
4. Term Polymorphism (I want only one identity function)
5. Type Polymorism AKA Universes (I want only one List type)
6. Impredicativity (I want a type with a "forall types" in it)
7. Principal types (I want to say "the type" of a term, not "a type")
8. Type computations (I want the computer to tell me "the" type)

The idea is that we can make more careful judgements about what kind of expresivity is needed, and where the tradeoffs can be made when designing a language.