Data Definitions in System F

In the spirit of "System F is a programming language", we are missing some things. We defined the type alias

$$Bool := \forall \alpha, \alpha \to \alpha \to \alpha$$

and observed that the following terms are instances of $Bool$

$$T := \Lambda\alpha\lambda x : \alpha, \lambda y : \alpha, x \qquad F := \Lambda\alpha\lambda x : \alpha, \lambda y : \alpha, y$$

Notice that we actually should not drop the annotations here, lest we infer principal types as

$$\vdash T : \forall \alpha, \forall \beta, \alpha \to \beta \to \alpha$$

We want T and F to have the same type, so we are explicit. There are two big questions we left open from last time

1. Are these the only two members of $Bool$?
2. Can other standard data types translate into System F?

The answer to both of these question will come from a mix of Theorems for free and the Church encodings.

Let's start with a quick theorems for free refresher.

1 theorems for free refresher

The theorems for free trick is to use the same relations we build during the normalization theorem. We work on relations $\mathcal{A} \subset A \times A'$, and we want to build an interpretation of types into relations. Recall we have two relation level operations corresponding to arrows and type abstractions.

1. $\mathcal{A} \to_R \mathcal{B} := \{(f, f') | \mathcal{A}(d, d') \Rightarrow \mathcal{B}(f\, d, f\, d')\}$
2. $\forall_R \mathcal{F} := \{(g, g') | \forall A, A', \mathcal{A} \subset A \times A', \mathcal{F}(\mathcal{A})(gA, g'A')\}$

I'm skipping over some nuances about domains that are more distracting than helpful here. Then given an environment mapping type variables to relations, we can define

1. $[\![\alpha]\!]_\rho := \rho(\alpha)$
2. $[\![A \to B]\!]_\rho := [\![A]\!]_\rho \to_R [\![B]\!]_\rho$
3. $[\![\forall \alpha, A]\!]_\rho := \forall_R(\mathcal{A} \mapsto [\![A]\!]_{(\rho[\alpha:=\mathcal{A}])})$

Then the parametricity theorem states that whenever you have environments ρ, ρ' which behave sensibly with respect to substitution in some context Γ, and $\Gamma \vdash t : A$

$$[\![A]\!]_\rho(\rho(t), \rho'(t))$$

The trick we saw last time where we quantify over relations in (3) seems to be enough to circumvent the impredicativity. So apparently the induction on t goes through fine. The requirement on the environment to respect substitution is very simple, so no interesting realizability stuff is required.

Unpacking these relations is a painful and uninteresting process, but generally when things are functions you get commutativity-looking things.

1.1 The Unit Type

Let's define

$$Unit := \forall \alpha, \alpha \to \alpha$$

and notice that $\vdash (\Lambda \alpha, \lambda x : \alpha, x) : Unit$. The theorem free says that if $\Gamma \vdash h : Unit$, A and B are types, and $\Gamma \vdash f : A \to B$ then

$$f \circ h\, A = h\, B \circ f$$

Then for any $\Gamma \vdash e : A$, choose $f = Te$. So for any

$$e = T\, e\, (h\, A\, e) = (T\, e \circ h\, A)\, e = (h\, A \circ T\, e)e \; = h\, A\, (T\, e\, e) = h\, A\, e$$

In other words, all terms of type $Unit$ behave like I. It's actually quite reasonable to call this thing $Unit$. Note it has more than one term, since $\vdash T\, I\, I : Unit$ but that's fine.

1.2 The Bool Type, and What System F knows

The theorem for free for $Bool$ is: Suppose $\Gamma \vdash e : Bool$, A and B are types, $\Gamma \vdash a : A$, $\Gamma \vdash a' : A$, and $f : A \to B$. Then that

$$e\, B\, (f\, a)(f\, a') = f(e\, A\, a\, a')$$

Which gives us a bunch of nice things like

$$(e\, T\, F)\, a\, a' = e\, (T\, a\, a')\, (F\, a\, a') = b\, a\, a' \Rightarrow e\, T\, F = e$$

But this isn't quite strong enough to prove what we want. Apparently there is a proof that involves observing all possible normal forms for $Bool$ have the shape of T or F. But that's a bit unsatisfying since System F can't do this kind of reasoning internally.

That deserves a moment of consideration. The theorems for free paper mentions an interesting quirk. There is no polymorphic equality $\forall \alpha, \alpha \to \alpha \to Bool$. The theorem for free for that thing is False. Which means we really can't do the kind of term introspection and normalization required in general. What we want is a term $\vdash viewAsTF : Bool \to Bool$ which clearly sends every element of $Bool$ to one of T or F. This seems like a trivial distinction, but we'll see shortly that it matters.

What would be sufficient is if we had a type A with an internal equality such that

$$eq_A\, x\, y : A \to A \to Bool \;\text{ such that }\; eq_A\, x\, y = T \text{ if } x = y \text{ and } y = F \text{ if } x \neq y$$

and two distinct, x, y, then we could finish the proof by

$$b = b\, T\, F = b(eq_A\, x\, x)(eq_A\, x\, y) = eq_A\, x\, (b\, x\, y)$$

So then b must be equal to T or F. This looks is a bit cheating, because we are nearly assuming our conclusion.

1.3 Extending System F with Datas

Turns out I want to cheat. I want to say "Here's a type A that I think looks like Booleans. Do you agree¿' And then system F responds with a either

1. terms $f : A \to Bool$ and $g : Bool \to A$ with $f \circ g = I\, Bool$ and $g \circ f = I\, A$
2. a function that somehow witnesses they are distinct. That is a tricker topic.

To describe what I mean as a boolean, in haskell, we would write

$$\textbf{data Bool := True} \,|\, \textbf{False}$$

To explain what the means to system F, I extend it's type with a new constant $Bool_H$ and the following typing judgements

$$\text{True}\,\frac{}{\Gamma \vdash True_H : Bool_H} \qquad \text{False}\,\frac{}{\Gamma \vdash False_H : Bool_H}$$

and the elimination rule

$$\text{elim}\,\frac{\Gamma \vdash b : Bool_H \; \Gamma \vdash e_1 : A \; \Gamma \vdash e_2 : A}{\Gamma \vdash \textbf{if}\, b\, \textbf{then}\, e_1\, \textbf{else}\, e_2}$$

Note then that such a type has exactly the property we were expecting. So there really are only two values of $Bool$. Furthermore, the term

$$Beq := \lambda b\, \textbf{if}\, b\, \textbf{then}\, T\, \textbf{else}\, F$$

is a decidable equality on this type. So we can build our pair

$$(\lambda b, b\, True_H\, False_H) + Beq$$

So system F can take an arbitrary $Bool$, and convert faithfully into a T or F. This is done internally with a system F term. Ok, but can we do this in general? Was there something special about the relationship between $Bool_H$ and Bool? Notice the relationship between the system F type $Bool$ and the if statement?

1.4 The Church Encoding

As a note on naming, there are Scott encodings, and Church encodings. To be honest I don't totally understand the difference. But it appears that they agree on non-recursive types anyway. So let's call them Church encodings for now.

Turns out we the trick is that the type $Bool$ matches the elimination form of $Bool_H$. To do this in general, let's start with a nonrecusrive, nongeneric, ADTs. They are structured hierarchically in 3 parts.

$$Top ::= Name := R$$

A (potentially empty) union component

$$R ::= P_1 : | : ... : | : P_N$$

And a (potentially empty) product component

$$P ::= T_1 : * : ... : * : T_N$$

where T_i are also ADTs. We build the Church encoding we'll denote as a $\hat{A}$

1. $Name := R$ becomes $\forall \alpha, \hat{R} \to$
2. $P_1 : | : ... : | : P_N$ becomes $\hat{P_1} \to ... \to \hat{P_N} \to \alpha$
3. $T_1 : * : ... : * : T_N$ becomes $(\hat{T_1} \to \alpha) \to ... \to (\hat{T_N} \to \alpha)$

You'll notice that our definition of *Bool* satisfies this because each case is empty, and there are two cases. Same for our definition of *Unit*.

There are several things missing to deal with types we want day to day. First, what about recursive types? Second, what about type arguments? Turns out these will use the same trick that we will see next time. But already we can start encoding some interesting things.

Some theorems for free and a bit of effort will show that all of these extensions are boring.

Theorem: Consider a (non-recursive, parameter free) ADT $\mathbf{D}$, in system F extended by $\mathbf{D}$ (that is with the introduction and elimination rules). Then there are functions $f : \mathbf{D} \to \hat{\mathbf{D}}$ and $g : \hat{\mathbf{D}} \to \mathbf{D}$ which show they are isomorphic.

In other words, these two types are isomorphic internally in System F. Ok, so what's the big deal with the internal isomorphisms?

The problem is Church encodings are not in the Hindley Milner fragment. You can nest ADTs, but you can't nest their encodings in HM. However, if we introduce (finitely many) new ground types, the classic Hindley Milner inference algorithm will still work! So we can extend HM with all the of ADTs we need, and write code with good inference.

This is the critical bit: how do we eliminate those ADTs so we can get back to pure System F for evaluation? The trick is that we can soundly insert isomorphisms anytime we want during compilation. So you can write a compiler pass to convert your "HM + ADTs $\Rightarrow$ F". This only works because the isomorphism are actual terms in system F, and not just theoretic creations.

1.5 Some More Examples

We can write down some easy ones. For fixed types A and B, we have

$$Either_{(A,B)} = \forall\alpha, (A \to \alpha) \to (B \to \alpha) \to \alpha$$

and

$$Pair_{(A,B)} = \forall\alpha, (A \to B \to \alpha) \to \alpha$$

For some weirder cases, consider the ADT **data Void** with encoding

$$Bottom := \forall\alpha, \alpha$$

Which, by a theorem for free, is uninhabited. For an even stranger example, consider the encoding of **data IdentA = Ident A**

$$Ident_A := \forall\alpha, (A \to \alpha) \to \alpha$$

You might recognize this as the continuation passing transform. But we keep having the same problem as before. There is no longer a polymorphic identity *encoding*. Even though there is a polymorphic identity *function*. We will fix this next time, and address the lie just told above. Haskell is not system F, bur rather system F_ω.