System F

1 Restating our goals

Our journey began with rewrite systems, which were way too general. But gave us a nice framework to discuss language designs. Next, we looked at the untyped lambda calculus, which had some nice features, but also had some really powerful, scary stuff. From a language design perspective, it's full of foot-guns. We tried to eliminate those dangers with a type system, the Simply Typed Lambda Calculus. This worked, eliminating terms like $\lambda x.\, x\, x$. We also saw that STLC has progress and preservation, so good things are happening.

But STLC had (at least) two weaknesses. We saw some weirdness around the identity function. Lambda calculus has a term $\lambda x.\, x$. But STLC requires and type annotation on that λ, so there is no single identity function. This is not morally wrong, just a bit awkward. We also had to introduce ground types in the language a priori. This is a bit weirder, since it seems to dramatically limit what a user can express. We're used to languages that let users define their own types, after all!

Today, we will introduce System F, which presents alternative tradeoffs. We will recover a unique identity function, at the cost of some rather complex terms. We will also be able to express new types internally in system F without extending the term language. But that will cost us type inference.

2 Defining System F

Note the F in system F is just a historical coincidence, nothing special. Sometimes it's called the polymorphic lambda calculus. We'll stick to system F. Our plan is as before. Define types, define a a big set of terms, including type annotations. Then we'll define our β rule(s), and a typing judgement And lastly define system F to be the well-typed terms under βs. Our type language is

$$t := \alpha \mid \forall \alpha.\, t \mid t \to t$$

We have type-variables α, binders $\forall \alpha$, and arrows. Examples of such types are

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

Several things to note. We don't have ground types. We will talk about why at length later. The quantifiers can appear anywhere. We also don't require the types to be closed, necessarily. We will mention when we want that.

The our term syntax is almost like the annotated lambda calculus, but with a little extra power.

$$e := x \mid \lambda x : t.\, e \mid e\, e \mid \Lambda \alpha.\, e \mid e\, t$$

The annotated types come from the system F types. And that Λ is called type abstraction, and the corresponding type application.

As usual, to define β, we need a notion of substitution. Annoyingly, we need three substitutions. The first two are the obvious ones from the syntax of terms and types.

1. term-in-term: $e_1[x := e_2]$
   (a) $x[x := e] = e$
   (b) $y[x := e] = y$

(c) $(\lambda x : t.\, e_1)[x := e_2] = \lambda x : t.\, e_1$
(d) $(\lambda y : t.\, e_1)[x := e_2] = \lambda y : t.\, e_1[x := e_2]$
(e) $(e_1\, e_2)[x := e_3] = e_1[x := e_3]\, e_2[x := e_3]$
(f) $(\Lambda\alpha.\, e_1)[x := e_2] = \Lambda\alpha.\, e_1[x := e_2]$
(g) $(e_1\, t)[x := e_2] = \Lambda\alpha.\, e_1[x := e_2]\, t$

2. type-in-type: $t_1[\alpha := t_2]$

(a) $\alpha[\alpha := t] = t$
(b) $\beta[\alpha := t] = \beta$
(c) $(\forall\alpha.\, t_1)[\alpha := t_2] = \forall\alpha.\, t_1$
(d) $(\forall\beta.\, t_1)[\alpha := t_2] = \forall\beta.\, t_1[\alpha := t_2]$
(e) $(t_1 \to t_2)[\alpha := t_3] = t_1[\alpha := t_t] \to t_2[\alpha := t_3]$

3. type-in-term: $e[\alpha := t]$

(a) $x[\alpha := t] = x$
(b) $(\lambda x : t, e)[\alpha := t] = \lambda x : (t[\alpha := t]).\, e[\alpha := t]$
(c) $(e_1\, e_2)[\alpha := t] = e_1[\alpha := t]\, e_2[\alpha := t]$
(d) $(\Lambda\alpha.\, e)[\alpha := t] = \Lambda\alpha.\, e$
(e) $(\Lambda\beta.\, e)[\alpha := t] = \Lambda\beta.\, e[\alpha := t]$
(f) $(e\, t_1)[\alpha := t_2] = e[\alpha := t_2]\, t_1[\alpha := t_2]$

Note there is an asymmetry: no term-in-type substitution. This is not an accident! More on this later. With our substitutions defined, we can now build our βs Since there are two abstraction/application pairs, we will need two

$$(\beta_{\text{term}}) \quad (\lambda x : t.\, e_1)\, e_2 \to e_1[x := e_2]$$

and

$$(\beta_{\text{type}}) \quad (\Lambda\alpha.\, e)\, t \to e[\alpha := t]$$

And lastly, we need our judgements. But since we have two sorts of abstractions, we also need two sorts of contextual info. We don't need to record any extra info about Λ other than the name of the thing we introduced. But we'll write $(\alpha, *)$ for symmetry with (x, t) for members of the context.

$$\text{Cxt}\ \frac{(x,t) \in \Gamma}{\Gamma \vdash x : t} \qquad \text{App}\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\, e_2 : t_2} \qquad \text{Abs}\ \frac{\Gamma, (x, t_1) \vdash e : t_2}{\Gamma \vdash \lambda x : t_1.\, e : t_1 \to t_2}$$

and the corresponding type rules

$$\text{TypeApp}\ \frac{\Gamma \vdash e : \forall\alpha.\, t_1}{\Gamma \vdash e\, t_2 : t_1[\alpha := t_2]} \qquad \text{TypeAbs}\ \frac{\Gamma, (\alpha, *) \vdash e : t}{\Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, t}$$

And now, finally, we can define system F to be the well typed terms of this language, along with β_{term} and β_{type}. We have just spent a month learning how to ask good questions about languages. Let's ask them.

3 System F's basic properties

As usual, we really like when our programming languages are confluent. Thankfully system F is indeed confluent. One way to do this proof is to just use the same parallel β trick as before. You'll end up needing that term-in-term and type-in-term substitutions commute.

$$e_1[\alpha := t][x := e_2] = e_1[x := e_2][\alpha := t]$$

We won't go through the proof in detail here, since we went through it full detail before.

Let's consider some examples of system F terms. We previously complained that there was no general identity for STLC. But system F has

$$\Lambda\alpha.\, \lambda x : \alpha.\, x$$

with type derivation

$$\dfrac{\dfrac{\dfrac{(x,\alpha) \in (x,\alpha),(\alpha,*)}{(x,\alpha),(\alpha,*) \vdash x : \alpha}}{(\alpha,*) \vdash (\lambda x : \alpha.\, x) : \alpha \to \alpha}}{\vdash (\Lambda\alpha.\, \lambda x : \alpha.\, x) : \forall\alpha, \alpha \to \alpha}$$

So we are happy about this. Note further that the simply typed lambda calculus embeds faithfully into system F. So all those terms come too. We also know to ask about normalization.

But if we want to write real code, how do we do it? For starters, where are if statements and boolean? Little idea: given a boolean, you can branch on it. So we'd like a type like

$$\forall\alpha, \mathit{Bool} \to \alpha \to \alpha \to \alpha$$

but what is Bool? Big idea: given a boolean you can *only ever branch on it.* In other words, a bool is defined by it's branching behavior:

$$\mathit{Bool} := \forall\alpha, \alpha \to \alpha \to \alpha$$

This seems crazy, since it's not clear how many members of Bool there are. Certainly there are at least two

$$T := \Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, x \quad \text{and} \quad F := \Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, y$$

Our medium term goal is to use theorems for free to prove these are the only two!

Likewise, how would we write a list of bools? Little idea: given a list, we can fold over it.

$$\forall\alpha.\, \mathit{BoolList} \to \alpha \to (\mathit{Bool} \to \alpha \to \alpha) \to \alpha$$

Big idea: the only thing you can do with a list is fold over it.

$$\mathit{BoolList} := \forall\alpha.\, \alpha \to (\mathit{Bool} \to \alpha \to \alpha) \to \alpha$$

I'm using caution not to call this thing $(\mathit{List}\, \mathit{Bool})$ because in Haskell, $\mathit{List} :: * \to *$, and we don't have any way to describe that here. In particular there is no App for types. We can build an empty list of bools

$$\mathit{BLNil} := \forall\alpha.\, \lambda(x : \alpha).\, \lambda(y : \mathit{Bool} \to \alpha \to \alpha).\, x$$

and a cons of type $\forall\alpha, \mathit{Bool} \to \mathit{BoolList} \to \mathit{BoolList}$

$$\mathit{BLCons} := \forall\alpha.\, \lambda(b : \mathit{Bool}).\, \lambda(l : \mathit{BoolList}).\, \lambda(x : \alpha).\, \lambda(y : \mathit{Bool} \to \alpha \to \alpha).\, y\, b\, (l\, \alpha\, y\, x)$$

And that's with some syntax sugar! Withe everything fully inlined we get

$$
\begin{aligned}
BLCons := \forall\alpha.\\
&\lambda(b : \forall\beta.\, \beta \to \beta \to \beta).\\
&\lambda(l : \forall\beta.\, \beta \to ((\forall\beta'.\, \beta' \to \beta' \to \beta') \to \beta \to \beta) \to \beta).\\
&\lambda(x : \alpha).\\
&\lambda(y : (\forall\beta.\, \beta \to \beta \to \beta) \to \alpha \to \alpha).\\
&y\, b\, (l\, \alpha\, y\, x)
\end{aligned}
\tag{1}
$$

Ok, that's a 7 symbol term with a paragraph of type annotations. Something awful has happened, and we'll need to spend some time to fix this. On the other hand, we use these type parameters to expression some useful computations. System F is servicably powerful to do some interesting stuff here, but we'll need to take some time to understand it's limits. Next up, we have

1. normalization by evaluation for system F (and beyond?).
2. inference and the Hindle-Milner fragment
3. Scott encodings of haskell ADTs
4. theorems for free, and the semantics of these encodings