1 Syntax

Types. The syntax of polymorphic types $T$ is given by

$$ T ::= X | T_1 \to T_2 | \forall X.T $$

where $X$ is from an infinite set of type variables. The type-form $\forall X.T$ is for *universal* types. We will use universal types to classify polymorphic functions.

Terms. We will work here directly with annotated terms and type computation (rules below), since unlike for the simply typed lambda calculus, for System F, it is in general undecidable whether or not an unannotated term can be assigned a polymorphic type. The syntax for annotated terms is as follows:

$$ t ::= x | (t_1 t_2) | \lambda x : T.t | t[T] | \lambda X.t $$

The first three term constructs are as for simply typed or untyped lambda calculus. The constructs $t[T]$ and $\lambda X.t$ are for *type instantiation* and *type abstraction*, respectively.

2 Informal Examples

Polymorphic identity. We can compute a single type, rather than a type scheme, for a System F term implementing the polymorphic identity function. The typing is:

$$ \lambda X.\lambda x : X.x : \forall X.X \to X $$

The idea in System F is that our annotated terms can abstract over types (with $\lambda X.t$) and then instantiate a type abstraction (with $t[T]$). The type for a type abstraction $\lambda X.t$ is $\forall X.T$, where $t$ has type $T$ in a context where $X$ is declared.

$\lambda x.(x x)$. The term $\lambda x.(x x)$ is not simply typable, but we can give an annotated System F term corresponding to it which is typable. This example demonstrates also the use of instantiation. The typing is

$$ \lambda x : \forall X.X. (x[(\forall X.X) \to (\forall X.X)] x) : (\forall X.X) \to (\forall X.X) $$

Let us consider this example in more detail. The term in question first takes in $x$ of type $\forall X.X$. Such an $x$ is a very powerful term, since for any type $T$, we have $x[T] : T$. So this term can, via its instantiations, take on any type $T$ we wish. So the term in question instantiates $x$ at the type $\forall X.X \to (\forall X.X)$. So the instantiated $x$ now has the type of a function taking an input of type $\forall X.X$ and returning an output of the same type. So we can apply the instantiated $x$ to $x$ itself. The type of the application is then $\forall X.X$, which completes the explanation of the typing of this term.

Note that typing prevents us from applying $\lambda x : \forall X.X. (x[(\forall X.X) \to (\forall X.X)] x)$ to itself. This is good, since we know that applying $\lambda x.(x x)$ to itself diverges.

3 Type Computation Rules

The following rules inductively define the type computation relation for System F. We use contexts $\Gamma$ that declare the type variables $X$, to keep track of their scoping:

$$ \Gamma ::= \cdot | \Gamma, x : T | \Gamma, X : type $$
As for simple typing, we will write $\Gamma(x) = T$ to mean that the result of looking up the type for term variable $x$ in context $\Gamma$ is $T$ (i.e., the function $\Gamma$ returns type $T$ for $x$). We will here assume that $\lambda$-bound variables are tacitly renamed to ensure that the typing context always has at most one declaration for any variable (either term variable $x$ or type variable $X$). This will ensure we do not confuse scopes of term or type variables with the same names.

\[ \Gamma(x) = T \]
\[ \Gamma \vdash x : T \]
\[ \Gamma \vdash t_1 : T_2 \rightarrow T_1 \quad \Gamma \vdash t_2 : T_2 \quad \Gamma, x : T_1 \vdash t : T_2 \quad \Gamma \vdash \lambda x : T_1. t : T_1 \rightarrow T_2 \]
\[ \Gamma, X : type \vdash t : T \quad \Gamma \vdash \lambda X. t : \forall X. T' \]
\[ \Gamma \vdash t : \forall X. T' \quad \Gamma \vdash t[P] : [T/X]T' \]

In the typing rule for instantiation, we must substitute the type $T$ into the body of the $\forall$-type. This is what allows us, for example, to give $x[(\forall X.X) \rightarrow (\forall X.X)]$ the type $(\forall X.X) \rightarrow (\forall X.X)$ if $x$ has type $\forall X.X$. (The body $T'$ in this case is just $X$.)

4 Reduction Semantics

We will use a reduction semantics for System F that works directly on our annotated terms:

\[ C[[\lambda x : T. t] t'] \rightsquigarrow C[[t'/x]t] \]
\[ C[[\lambda X. t][T]] \rightsquigarrow C[[T/X]t] \]

Here, the contexts $C$ allow reduction anywhere:

\[ C ::= * \mid (C t) \mid (t C) \mid \lambda x : T. C \mid \lambda X. C \mid C[T] \]

5 Metatheory

Theorem 1 (Type Preservation) If $\Gamma \vdash t : T$ (in System F) and $t \rightsquigarrow t'$, then $\Gamma \vdash t' : T$.

Theorem 2 (Strong Normalization) If $\Gamma \vdash t : T$ (in System F), then $t \in SN$.

The proof is based on reducibility, as for simple types, but it requires a major innovation, due to Jean-Yves Girard, to define reducibility for universal types $\forall X. t$. See the book “Proofs and Types”, by Girard et al., for a very nice presentation of this complex argument.

6 Church Encodings

One of the amazing things about System F is that we can express quite interesting algorithms (for example, sorting of lists) as typable System F terms. Since every typable term in System F is (hereditarily) strongly normalizing, this means that we can prove totality of functions, for example, just by encoding them in System F. From recursion theory, we know that not all total functions can be encoded in System F, since there is no recursive language consisting of all and only the total functions. But still, System F is remarkably expressive, as we will now see. To emphasize: all the functions we write below are guaranteed to terminate on all inputs, just in virtue of the fact that they type check in System F.

For this to work, we need to use the Church encoding, instead of the Scott encoding. In the Scott encoding (which we saw previously), data are encoded by their own case-statements. In the Church encoding, data are encoded by their own iterators. These slogans will hopefully be clarified as we consider examples.
6.1 Unary numbers

Here are the Church encodings for 0 and the successor function in System F. We can actually define the type of natural numbers.

\[
\text{nat} := \forall X. (X \rightarrow X) \rightarrow X \rightarrow X
\]
\[
0 := \lambda X. \lambda s : X \rightarrow X. \lambda z : X. z
\]
\[
S := \lambda n : \text{nat}. \lambda X. \lambda s : X \rightarrow X. \lambda z : X. (s (n[X] s z))
\]

You can confirm that we then have these typings:

\[
0 : \text{nat}
\]
\[
S : \text{nat} \rightarrow \text{nat}
\]

**Addition.** The System F term for addition is:

\[
\text{plus} := \lambda n : \text{nat}. \lambda m : \text{nat}. (n[\text{nat}] S m)
\]

This term will iterate the successor function \(n\) times, starting from \(m\). This will indeed produce \((S (S \cdots m))\), with \(n\) calls to successor.

6.2 Polymorphic Lists

\[
\langle \text{list } A \rangle := \forall X. (A \rightarrow X \rightarrow X) \rightarrow X \rightarrow X
\]
\[
\text{nil} := \lambda A. \lambda X. \lambda c : A \rightarrow X \rightarrow X. \lambda n : X. n
\]
\[
\text{cons} := \lambda A. \lambda a : A. \lambda l : \langle \text{list } A \rangle.
\]
\[
\lambda X. \lambda c : A \rightarrow X \rightarrow X. \lambda n : X. (c a (l[X] c n))
\]

You can confirm that we then have these typings:

\[
\text{nil} : \forall A. \langle \text{list } A \rangle
\]
\[
\text{cons} : \forall A. A \rightarrow \langle \text{list } A \rangle \rightarrow \langle \text{list } A \rangle
\]