We saw how to use inductive and coinductive types last recitation. We can also add polymorphic
types to our type system, which leads us to System $\mathbf{F}$. With polymorphic types, we can have
functions that work the same regardless of the types of some parts of the expression.\footnote{These notes are derived from 15-312 Spring 2017 course notes by Jake Zimmerman.}

## 1 System $\mathbf{F}$

Inductive and coinductive types expand the expressivity of $\mathbf{T}$ considerably. The power of type
operators allows us to generically manipulate data of heterogeneous types, building new types
from old ones. But to write truly “generic” programs, we want truly polymorphic expressions—
functions that operate on containers of some arbitrary type, for example. To gain this power,
we add \textit{parametric polymorphism} to the language, which results in System $\mathbf{F}$, introduced by
Girard (1972) and Reynolds (1974).

\[
\begin{align*}
\text{Typ} & \quad \tau ::= t & \text{type variable} \\
& \quad \tau_1 \to \tau_2 & \text{function} \\
& \quad \forall (t. \tau) & \text{universal type}
\end{align*}
\]

\[
\begin{align*}
\text{Exp} & \quad e ::= x & \text{variable} \\
& \quad \lambda (x : \tau) e & \text{abstraction} \\
& \quad e_1(e_2) & \text{application} \\
& \quad \Lambda(t) e & \text{type abstraction} \\
& \quad e[\tau] & \text{type application}
\end{align*}
\]

Take stock of what we’ve added since last time, and what we’ve removed. The familiar \textbf{type}
\textbf{variables} are now baked into the language, along with the \textbf{universal type}. We also have
a new form of lambda expression, one that works over type variables rather than expression
variables.

What’s missing? Nearly every other construct we’ve come to know and love! As will be the case
repeatedly in the course, our tools such as products, sums, and inductive types are subsumed
by the new polymorphic types. The result is an extremely simple System $\mathbf{F}$ that is actually
even more powerful.
1.1 Statics

Now that types have variables, we need to decide which type abt’s are considered valid. We introduce the following judgment:

\[ \Delta \vdash \tau \text{ type} \]

meaning that in the type context \( \Delta \), \( \tau \) is a valid type. The type context \( \Delta \) contains the type variables that we have seen so far.

We also attach the type context to the typing judgment, which now looks like:

\[ \Delta, \Gamma \vdash e : \tau \]

To define what types are valid, we essentially just want to state that closed types (ones with no free variables) are valid, and open types are invalid. These rules express that fact:

\[
\begin{align*}
\Delta, \tau \text{ type } & \vdash \tau \text{ type} \\
\Delta \vdash \tau_1 \text{ type } \quad \Delta \vdash \tau_2 \text{ type } & \quad \Delta \vdash \tau_1 \rightarrow \tau_2 \text{ type} \\
\Delta \vdash \tau \text{ type } & \quad \Delta, t \text{ type } \vdash \tau \text{ type} \\
\end{align*}
\]

And now we may define the typing judgment. The cases for variable, lambda, and application are as they were in System T; we simply carry \( \Delta \) along for the ride. There are two interesting new rules:

\[
\begin{align*}
\Delta, t \text{ type } \quad \Gamma \vdash e : \tau & \quad \Delta, \Gamma \vdash \Lambda(t) e : \forall t. \tau \\
\Delta \vdash e : \forall t. \tau & \quad \Delta, \Gamma \vdash e[\tau] : [\tau/t] \tau' \\
\end{align*}
\]

Type lambdas are the introduction of universal types, and type applications are their elimination. The type application rule saying that if some expression \( e \) is valid for all choices of \( t \), then it will also be valid when the actual type \( \tau \) is substituted for \( t \) (provided that \( \tau \) is a valid type).

This is very similar to polymorphic types in ML, where types may contain type variables. Be aware that ML usually leaves the type lambda implicit. That is, the ML type

\[
('a \rightarrow 'b) \rightarrow 'c
\]

is actually

\[
\forall \alpha. \forall \beta. \forall \gamma. (\alpha \rightarrow \beta) \rightarrow \gamma
\]

\( \alpha \) in System F. Observe that ML implicitly places the type lambdas at the front of the type. As we will soon see, this is an important distinction between ML and System F. ML cannot directly express a type like

\[
\forall \alpha. \alpha \rightarrow \forall \beta. \beta
\]

which System F easily can do.

ML also does not explicitly apply types. Consider the polymorphic identity function in F:

\[
\text{id} \triangleq \forall \alpha. \lambda(x : \alpha) x
\]

This function is truly polymorphic, as we can apply \( \text{id}[\text{nat}] \) to get the identity function on naturals, \( \text{id}[\text{nat} \rightarrow \text{nat}] \) to get the identity function on functions from naturals to naturals, etc. However, in ML, the type checker automatically applies the appropriate type argument to its type abstractions. \( \text{id} 0 \) and \( \text{id} \ (\text{fn} \ (x : \text{nat}) \Rightarrow x) \) implicitly involve the specialization of the function \( \text{id} \).
1.2 Dynamics

System F also has a remarkably simple dynamics. The rules for lambda and application remain the same as in lazy/eager System T, and we need only introduce the rules for type lambda and type application.

\[
\begin{align*}
\Lambda(t) e & \rightarrow e' \\
\text{val} & \mapsto e \\
\[\tau\] & \mapsto [\tau/e'] \\
\Lambda(t)[\tau] & \mapsto [\tau/t][e]
\end{align*}
\]

That’s it! Type functions are values, type applications are eager, and they eventually substitute a type for a variable in a type abstraction.

Examples:

\[
\Lambda(\alpha) \lambda(x:\alpha)x \text{ is the polymorphic identity function}
\]

\[
\Lambda(\alpha) \Lambda(\beta) \lambda(f:\alpha\to\beta)\lambda(x:\alpha)f(x) \text{ is the polymorphic applicator function}
\]

1.3 Church Encodings

System F is great, but we’re still pining for all our old types like natural numbers, lists, etc. But what if I told you that universal types could replace all of them? As it turns out, we can construct products, sums, inductive types, etc. in System F, using a scheme called Church encodings.

How would we express \texttt{nat} in System F?

\[
\texttt{nat} \equiv \forall t.t \to (t \to t) \to t
\]

It may be difficult at first glance to see why this polymorphic type expresses everything we need for \texttt{nat}. Using this definition of \texttt{nat}, how would we write \texttt{z}, \texttt{s}, and \texttt{rec} - all the stuff we used to have for \texttt{nat} in System T?

\[
\begin{align*}
\texttt{z} & \equiv \Lambda(t) \lambda(b:t)\lambda(s:t\to t)b \\
\texttt{s} & \equiv \lambda(x:\texttt{nat})\Lambda(t)\lambda(b:t)\lambda(s:t\to t)s(x[t](b)(s)) \\
\texttt{iter}\{\tau\}(e_1,x,e_2,e) & \equiv e[\tau](e_1)(\lambda(x:\tau)e_2)
\end{align*}
\]

As you can see in the definitions for \texttt{z}, \texttt{s}, and \texttt{rec}, the first polymorphic term taken in to the function represents the zero base case, and the second polymorphic term represents the successor case. We only need a way for us to define zero and the successor for us to be able to construct a natural number, and so we only need these two arguments before the polymorphic function gives us something of type \texttt{nat}.

In the Church encoding, the number is its own recursor! That is a powerful idea. A number is only as meaningful as the ability to count with it, and so it is fitting that numbers be represented using their recursor.

How would we express sum and product types in System F?

\[
\begin{align*}
\tau_1 + \tau_2 & \equiv \forall t.(\tau_1 \to t) \to (\tau_2 \to t) \to t \\
\tau_1 \times \tau_2 & \equiv \forall t.(\tau_1 \to \tau_2 \to t) \to t
\end{align*}
\]
For sum types, we simply need something of type $\tau_1$ or something of type $\tau_2$ to be able to get something of type $\tau_1 + \tau_2$. By a similar argument, we need something of type $\tau_1$ and something of type $\tau_2$ to be able to construct a term of type $\tau_1 \times \tau_2$.

\[
1 \cdot e \triangleq \Lambda(t) \lambda (l : \tau_1 \rightarrow t) \lambda (r : \tau_2 \rightarrow t) l(e)
\]
\[
r \cdot e \triangleq \Lambda(t) \lambda (l : \tau_1 \rightarrow t) \lambda (r : \tau_2 \rightarrow t) r(e)
\]
\[
\langle e_1, e_2 \rangle \triangleq \Lambda(t) \lambda (h : \tau_1 \rightarrow \tau_2 \rightarrow t) h(e_1)(e_2)
\]
\[
e \cdot 1 \triangleq e[\tau_1][\lambda (l : \tau_1) \lambda (r : \tau_2) l]
\]
\[
e \cdot r \triangleq e[\tau_2][\lambda (l : \tau_1) \lambda (r : \tau_2) r]
\]

Each function argument tells us how to interact with the sum or product type internally. As an exercise, try to define case in the encoding.

We can also now create polymorphic data structures, which you’ve seen in SML in 15-150:

\[
\alpha \text{ list} \triangleq \forall \alpha. \mu(t.1 + (\alpha \times t))
\]
\[
\alpha \text{ stream} \triangleq \forall \alpha. \nu(t.1 + (\alpha \times t))
\]

Note that the thing inside of the $\forall$ is a type operator! However, $\alpha. \mu(t.1 + (\alpha \times t))$ and $\alpha. \nu(t.1 + (\alpha \times t))$ are not polynomial type operators since they contain inductive and coinductive types. We can still change our map\{$t.\tau$\} to work with these type operators, though, and you’ll see how to do this in Assignment 3.