Day 17.

1. Parametric Polymorphism

Again, we’ll use abstraction to expose a weakness in the type systems we’ve been studying. Consider the following term and derivation:

\[
\begin{align*}
\{ a \mapsto \text{Int} \} & \vdash a : \text{Int} \to \text{Int} \\
\emptyset & \vdash \lambda a . a : \text{Int} \to \text{Int} \\
\emptyset & \vdash (\lambda a . a) (\lambda a . a) : \text{Int} \to \text{Int} \\
\emptyset & \vdash 1 : \text{Int}
\end{align*}
\]

Fine and good—we use \(\lambda a . a\) at two different types, but that’s fine. But now suppose we want to abstract over that function:

\[
\begin{align*}
\{ a \mapsto \text{Int} \} & \vdash a : \text{Int} \\
\emptyset & \vdash \lambda a . a : \text{Int} \to \text{Int} \\
\Gamma & \vdash f : (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int}) \\
\Gamma & \vdash f f : \text{Int} \to \text{Int} \\
\emptyset & \vdash \text{let } f = \lambda a . a \text{ in } f f 1 : \text{Int}
\end{align*}
\]

where \(\Gamma = \{ f \mapsto \text{Int} \to \text{Int} \}\).

- The problem is that we now need to assign a single type to \(f\)… but, as in the previous derivation, we use \(f\) in two different ways
- If we’d initially given \(f\) the type \((\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int})\), the same problem would appear in the other hypotheses.

Our solution: rather than giving \(f\) a single type, capture the family of types that \(f\) can take on.

2. Types and Type Schemes

Syntax:

\[
\begin{align*}
A & \ni \alpha \\
T & \ni t ::= \text{Int} \mid t \to t \mid \alpha \\
S & \ni s ::= t \mid \forall \alpha . s
\end{align*}
\]

- Types now include type variables \(\alpha, \beta, \ldots\). Type variables represent arbitrary types; for example, we could drive

\[
\begin{align*}
\{ a \mapsto \alpha \} & \vdash a : \alpha \\
\emptyset & \vdash \lambda a . a : \alpha \to \alpha
\end{align*}
\]
We *cannot* freely replace type variables with types—just like we can’t freely replace term variables with terms. For example, we cannot conclude that \( \{ a \mapsto \alpha \} \vdash a : \text{Int} \).

- Type schemes quantify over type variables: \( \alpha \to \alpha \) denotes a function from an arbitrary type to itself; \( \forall \alpha. \alpha \to \alpha \) denotes a function from any type to itself.
- Type schemes and type are stratified: we can have \( \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha) \) but not \( (\forall \alpha. \alpha \to \alpha) \to (\forall \alpha. \alpha \to \alpha) \).

How do we deal with schemes and type variables? Substitution \( u[t/\alpha]! \):

\[
\begin{align*}
\text{Int}[t/\alpha] &= \text{Int} \\
\beta[t/\alpha] &= \begin{cases} t & \text{if } \alpha = \beta \\
\beta & \text{otherwise} \end{cases} \\
(u_1 \to u_2)[t/\alpha] &= u_1[t/\alpha] \to u_2[t/\alpha] \\
(\forall \beta.s)[t/\alpha] &= \begin{cases} \forall \beta.S & \text{if } \alpha = \beta \\
\forall \beta.s[t/\alpha] & \text{otherwise} \end{cases}
\end{align*}
\]

- This should feel familiar
- Because types and schemes are stratified, we’re really defining two operations, \([-/-]: T \to T \to A \to Y \) and \([-/-]: S \to T \to A \to S \). But:
  - These aren’t even mutually recursive: schemes never appear inside types
  - We’ll never substitute schemes for variables, only types. (What would break if we could substitute schemes for variables?)
  - Why? Short answer: type inference. Longer answer: not really in a course here, but if you’re interested talk to me.

We can continue the familiar development here. The *free variables* of a type are those type variables not bound by an enclosing \( \forall \):

\[
\begin{align*}
fv(\text{Int}) &= \emptyset \\
fv(\alpha) &= \{ \alpha \} \\
fv(t_1 \to t_2) &= fv(t_1) \cup fv(t_2) \\
fv(\forall \alpha.s) &= fv(s) \setminus \{ \alpha \}
\end{align*}
\]

And we can define a notion of renaming-equivalence for types

\[
\begin{align*}
t_1 \equiv_\alpha u_1 & \quad t_2 \equiv_\alpha u_2 \\
t_1 \to t_2 \equiv_\alpha u_1 \to u_2 & \quad \text{Int} \equiv_\alpha \text{Int} \\
\alpha \equiv_\alpha \alpha \\
\forall \alpha.s_1 \equiv_\alpha s_2 & \quad \left( \gamma \text{ fresh for } s_1 \text{ and } s_2 \right)
\end{align*}
\]

- Yup, two different meanings of \( \alpha \). Notation sucks.
- A variable is *fresh for* a type if it appears nowhere in the type. We can define this formally, but it all becomes tedious.