Day 18.

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:

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

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:

\[
\begin{align*}
\Gamma & \vdash f : (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int}) \\
\Gamma & \vdash f f : \text{Int} \to \text{Int} \\
\Gamma & \vdash 1 : \text{Int}
\end{align*}
\]

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:

\[
\begin{align*}
\mathcal{A} & \ni \alpha \\
\mathcal{V} & \ni T ::= \text{Int} | T \to T | \alpha \\
\mathcal{S} & \ni S ::= T | \forall \alpha. S
\end{align*}
\]

- 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 $U[T/\alpha]$!

- This should feel familiar
- Because types and schemes are stratified, we’re really defining two operations, $[-/-] : Y \to Y \to A \to Y$ and $[-/-] : S \to Y \to A \to S$. 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 $\forall$:

$$fv(\text{Int}) = \emptyset \quad \text{fv}(T_1 \to T_2) = \text{fv}(T_1) \cup \text{fv}(T_2)$$

$$fv(\alpha) = \{\alpha\} \quad \text{fv}(\forall \alpha. S) = \text{fv}(S) \setminus \{\alpha\}$$

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

$$T_1 \equiv_\alpha U_1 \quad T_2 \equiv_\alpha U_2$$
$$T_1 \to T_2 \equiv_\alpha U_1 \to U_2$$
$$\text{Int} \equiv_\alpha \text{Int}$$
$$\alpha \equiv_\alpha \alpha$$

$$S_1[\gamma/\alpha] \equiv_\alpha S_2[\gamma/\beta] \quad (\gamma \text{ fresh for } S_1 \text{ and } S_2)$$

- 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.