Day 18.

多态

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:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{a \mapsto \texttt{Int} \to \texttt{Int}\} \vdash a : \texttt{Int} \to \texttt{Int}}}{\emptyset \vdash \lambda a.a : (\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})} \quad \dfrac{\dfrac{}{\{a \mapsto \texttt{Int}\} \vdash a : \texttt{Int}}}{\emptyset \vdash \lambda a.a : \texttt{Int} \to \texttt{Int}}}{\emptyset \vdash (\lambda a.a)\,(\lambda a.a) : \texttt{Int} \to \texttt{Int}} \quad \dfrac{}{\emptyset \vdash 1 : \texttt{Int}}}{\emptyset \vdash (\lambda a.a)\,(\lambda a.a)\,1 : \texttt{Int}}$$

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:

$$\dfrac{\dfrac{\dfrac{}{\{a \mapsto \texttt{Int}\} \vdash a : \texttt{Int}}}{\emptyset \vdash \lambda a.a : \texttt{Int} \to \texttt{Int}} \quad \dfrac{\dfrac{\Gamma \vdash f : (\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int}) \quad \dfrac{}{\Gamma \vdash f : \texttt{Int} \to \texttt{Int}}}{\Gamma \vdash f\,f : \texttt{Int} \to \texttt{Int}} \quad \dfrac{}{\Gamma \vdash 1 : \texttt{Int}}}{\Gamma \vdash f\,f\,1 : \texttt{Int}}}{\emptyset \vdash \texttt{let}\ f = \lambda a.a\ \texttt{in}\ f\,f\,1 : \texttt{Int}}$$

where $\Gamma = \{f \mapsto \texttt{Int} \to \texttt{Int}\}$.

- 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 $(\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})$, 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:

$$\begin{aligned}
\mathcal{A} &\ni \alpha \\
\mathcal{Y} &\ni T ::= \texttt{Int} \mid T \to T \mid \alpha \\
\mathcal{S} &\ni S ::= T \mid \forall \alpha . S
\end{aligned}$$

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

$$\dfrac{\dfrac{}{\{a \mapsto \alpha\} \vdash a : \alpha}}{\emptyset \vdash \lambda a.a : \alpha \to \alpha}$$

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 $\{a \mapsto \alpha\} \vdash a : \texttt{Int}$.

- 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 $U[T/\alpha]$!

$$\texttt{Int}[T/\alpha] = \texttt{Int} \qquad (U_1 \to U_2)[T/\alpha] = U_1[T/\alpha] \to U_2[T/\alpha]$$

$$\beta[T/\alpha] = \begin{cases} T & \text{if } \alpha = \beta \\ \beta & \text{otherwise} \end{cases} \qquad (\forall\beta.S)[T/\alpha] = \begin{cases} \forall\beta.S & \text{if } \alpha = \beta \\ \forall\beta.S[T/\alpha] & \text{otherwise} \end{cases}$$

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

$$fv(\texttt{Int}) = \emptyset \qquad fv(T_1 \to T_2) = fv(T_1) \cup fv(T_2)$$

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

除了alpha

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

$$\frac{T_1 \equiv_\alpha U_1 \quad T_2 \equiv_\alpha U_2}{T_1 \to T_2 \equiv_\alpha U_1 \to U_2} \qquad \frac{}{\texttt{Int} \equiv_\alpha \texttt{Int}} \qquad \frac{}{\alpha \equiv_\alpha \alpha}$$

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

- Yup, two different meanings of α. 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.