

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:

$$\frac{\frac{\frac{\overline{\{a \mapsto \text{Int} \rightarrow \text{Int}\} \vdash a : \text{Int} \rightarrow \text{Int}}}{\emptyset \vdash \lambda a.a : (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})} \quad \frac{\overline{\{a \mapsto \text{Int}\} \vdash a : \text{Int}}}{\emptyset \vdash \lambda a.a : \text{Int} \rightarrow \text{Int}}}{\frac{\emptyset \vdash (\lambda a.a) (\lambda a.a) : \text{Int} \rightarrow \text{Int} \quad \overline{\emptyset \vdash 1 : \text{Int}}}{\emptyset \vdash (\lambda a.a) (\lambda a.a) 1 : \text{Int}}}}$$

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:

$$\frac{\frac{\overline{\{a \mapsto \text{Int}\} \vdash a : \text{Int}}}{\emptyset \vdash \lambda a.a : \text{Int} \rightarrow \text{Int}} \quad \frac{\frac{\Gamma \vdash f : (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int}) \quad \overline{\Gamma \vdash f : \text{Int} \rightarrow \text{Int}}}{\Gamma \vdash f f : \text{Int} \rightarrow \text{Int}} \quad \overline{\Gamma \vdash 1 : \text{Int}}}{\Gamma \vdash f f 1 : \text{Int}}}{\emptyset \vdash \text{let } f = \lambda a.a \text{ in } f f 1 : \text{Int}}$$

where $\Gamma = \{f \mapsto \text{Int} \rightarrow \text{Int}\}$.

- 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 $(\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})$, 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:

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

- Types now include *type variables* α, β, \dots . Type variables represent arbitrary types; for example, we could drive

$$\frac{\overline{\{a \mapsto \alpha\} \vdash a : \alpha}}{\emptyset \vdash \lambda a.a : \alpha \rightarrow \alpha}$$

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

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

How do we deal with schemes and type variables? Substitution $U[T/\alpha]$!

$$\begin{aligned} \mathbf{Int}[T/\alpha] &= \mathbf{Int} & (U_1 \rightarrow U_2)[T/\alpha] &= U_1[T/\alpha] \rightarrow U_2[T/\alpha] \\ \beta[T/\alpha] &= \begin{cases} T & \text{if } \alpha = \beta \\ \beta & \text{otherwise} \end{cases} & (\forall\beta.S)[T/\alpha] &= \begin{cases} \forall\beta.S & \text{if } \alpha = \beta \\ \forall\beta.S[T/\alpha] & \text{otherwise} \end{cases} \end{aligned}$$

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

$$\begin{aligned} fv(\mathbf{Int}) &= \emptyset & fv(T_1 \rightarrow T_2) &= fv(T_1) \cup fv(T_2) \\ fv(\alpha) &= \{\alpha\} & fv(\forall\alpha.S) &= fv(S) \setminus \{\alpha\} \end{aligned}$$

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

$$\begin{aligned} \frac{T_1 \equiv_\alpha U_1 \quad T_2 \equiv_\alpha U_2}{T_1 \rightarrow T_2 \equiv_\alpha U_1 \rightarrow U_2} \quad \frac{}{\mathbf{Int} \equiv_\alpha \mathbf{Int}} \quad \frac{}{\alpha \equiv_\alpha \alpha} \\ \frac{S_1[\gamma/\alpha] \equiv_\alpha S_2[\gamma/\beta]}{\forall\alpha.S_1 \equiv_\alpha \forall\beta.S_2} \quad (\gamma \text{ fresh for } S_1 \text{ and } S_2) \end{aligned}$$

- 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.