Day 18.

1. Types and Type Schemes

Intuitively, we want to think about type schemes as denoting families of (or sets of) types. The type scheme $\forall\alpha.\alpha \to \alpha$ denotes the set $\{\texttt{Int} \to \texttt{Int}, \texttt{Bool} \to \texttt{Bool}, (\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int}), \dots\}$, whereas the scheme $\forall\alpha.\alpha$ denotes the set of all types. We can make this idea formal, as follows. We call the set of types denoted by a scheme s the *instances* of s, $\lfloor s \rfloor$, defined as follows:

$$\lfloor t \rfloor = \{t\} \qquad \lfloor \forall\alpha.s \rfloor = \bigcup_{t \in \mathcal{T}} \lfloor s[t/\alpha] \rfloor$$

So, for example:

$$\begin{aligned} \lfloor \forall\alpha.\alpha \rfloor &= \mathcal{T} \\ \lfloor \alpha \rfloor &= \alpha \\ \lfloor \forall\alpha.\forall\beta.\alpha \to \beta \rfloor &= \{t_1 \to t_2 \mid t_1 \in \mathcal{T}, t_2 \in \mathcal{T}\} \end{aligned}$$

This idea of instances aligns perfectly with our idea of renaming-equivalence:

Theorem 18.1. *$s_1 \equiv_\alpha s_2$ iff $\lfloor s_1 \rfloor = \lfloor s_2 \rfloor$.*

We can also use instances to order type schemes: we say a type scheme s_1 is *more general* than s_2, $s_1 \geq s_2$, iff $\lfloor s_1 \rfloor \supseteq \lfloor s_2 \rfloor$. For example,

$$\forall\alpha.\alpha \geq \forall\alpha.\alpha \to \alpha \geq \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha \times \beta \geq \forall\alpha.\alpha \times \texttt{Bool} \to \alpha \times \texttt{Bool} \geq \texttt{Int} \times \texttt{Bool} \to \texttt{Int} \times \texttt{Bool}.$$

We can relate generality to substitution, although with a bit of hair.

Theorem 18.2. *$\forall\overline{\alpha}.t_1 \geq \forall\overline{\beta}.t_2$ iff there are types $u_1 \dots u_n$ such that $t_1[\overline{u}/\overline{\alpha}] = t_2$.*

2. Polymorphism in Terms

Let's start with our standard functional expression language, and add polymorphism.

$$\mathcal{E} \ni e ::= z \mid e \odot e \mid x \mid \lambda x.e \mid e\ e$$

- We don't get any new expression forms for polymorphism—the idea is that $\lambda a.a$ should *implicitly* be polymorphic.

- This is following the *Curry-style* presentation that we've used almost all of the semester. Of course, there are also presentations of polymorphism, which makes polymorphism *explicit*; the most common of these is the *Girard-Reynolds* calculus, also called System F.

We have all the typing rules we're familiar with, and two new ones to introduce and eliminate polymorphism.

$$\frac{}{\Gamma \vdash z : \texttt{Int}} \quad \frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 \odot e_2}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \quad \frac{\Gamma[x \mapsto t_1] \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2} \quad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}$$

$$\frac{\Gamma \vdash e : \forall \alpha.s}{\Gamma \vdash e : s[t/\alpha]} \quad \frac{\Gamma \vdash e : s}{\Gamma \vdash e : \forall \alpha.s}$$

- Intuition: type variables stand in for arbitrary types, so if something produces a result of arbitrary type, it can produce a result of any type we like.
- Quantifiers here are making that assumption *explicit* in types, even though it remains *implicit* in terms.
- Functions restricted to work on types, *not* type schemes, to preserve the stratification of types and schemes.

Does this work as we want? We can derive a polymorphic type for the identity function, as we'd expect.

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

But we can also derive some types that we might not want! For example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{a \mapsto \alpha\} \vdash a : \alpha}}{\{a \mapsto \alpha\} \vdash a : \forall \alpha.\alpha}\ (*)}{\{a \mapsto \alpha\} \vdash a : \beta}}{\emptyset \vdash \lambda a.a : \alpha \to \beta}}{\emptyset \vdash \lambda a.a : \forall \beta.\alpha \to \beta}}{\emptyset \vdash \lambda a.a : \forall \alpha.\forall \beta.\alpha \to \beta}$$

What's gone wrong? Our intuition about type variables wasn't quite correct.

- Variables in the type of a expression may capture dependence of the term on its environment
- Variables only truly represent arbitrary types if they don't appear in the environment. (Called the *eigenvariable* condition.)

We can update our quantification rules to capture the eigenvariable condition:

$$\frac{\Gamma \vdash e : \forall \alpha.S}{\Gamma \vdash e : s[t/\alpha]} \quad \frac{\Gamma \vdash e : s}{\Gamma \vdash e : \forall \alpha.s}\ (\alpha \notin fv(\Gamma))$$

where $fv(\Gamma)$ is defined in terms of fv on schemes in an entirely boring fashion. Now the bogus generalization step (marked with an $*$) in the derivation above isn't valid—because α is in $fv(\{a \mapsto \alpha\})$.

We have one additional problem: because we've restricted functions we don't have any way to bind polymorphic values. We'll add a new language feature to introduce polymorphism... or actually, reuse an existing language feature.

$$\mathcal{E} \ni e ::= z \mid e \odot e \mid x \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e$$

with the typing rule

$$\frac{\Gamma \vdash e_1 : s \quad \Gamma[x \mapsto s] \vdash e_2 : t}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t}$$

This is *not* the way we've been using `let` so far! We can't interpret $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$ as $(\lambda x.t_2)\ t_1$. In the latter, x would have to have a monomorphic type, where our goal here is to make it polymorphic. Now, we can return to our starting example:

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

where $\Gamma = \{f \mapsto \forall\alpha.\alpha \to \alpha\}$.

3. Principal Types and Type Inference

We'll finish with a brief discussion of why Hindley-Milner polymorphism is restricted the way it is. One of the key properties of the Hindley-Milner system is that it has principal types.

Theorem 18.3. *If $\Gamma \vdash e : s_1$ and $\Gamma \vdash e : s_2$, then there is some s_0 such that $s_0 \geq S_1$, $s_0 \geq s_2$ and $\Gamma \vdash e : s_0$.*

Intuitively, this means that every typeable term has a most general type, and that if we can figure out that most general type it encompasses all the other types at which that term could be used. The practical consequence is that type inference can be useful. So long as the compiler has some way of computing a term's most general type, then there's no danger that the type the compiler computes isn't as good as the one you might intend.

I'm not going to prove principality for H-M, although I'm happy to talk about how you might outside of class.

Now suppose we relax the Hindley-Milner constraints just a little—we'll allow quantifiers to be nested in types. This gives a much simpler system—no more stratification of types and schemes, and no more special treatment of `let`. And, it's more expressive. But we can show that this system can't have principal types either. Consider these two typings:

$$\emptyset \vdash \lambda f.\lambda a.\lambda b.(f\ a, f\ b) : \forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha \to \alpha \to (\beta, \beta)$$
$$\emptyset \vdash \lambda f.\lambda a.\lambda b.(f\ a, f\ b) : \forall\alpha.\forall\beta.(\forall\gamma.\gamma \to \gamma) \to \alpha \to \beta \to (\alpha, \beta)$$

These types are fundamentally incomparable: the top one lets you pick a function and arguments that match, while the bottom one lets you pick different types for a and b. There's no more-general type that encompasses both of these (take my word for it). So, such a system, as expressive as it might be, can't have a well-defined type inference algorithm that never makes the wrong guess.