Day 19.

1. Explicit Parametric Polymorphism

Our previous type system imposed restrictions on how polymorphism was introduced and used; in particular, we didn't allow functions to take polymorphic arguments, and so had to introduce the `let` construct to manage polymorphism. In this section, we consider relaxing that restriction.

Why did have this restriction in the first place? Well, without it, it becomes difficult to figure out the types of terms. Consider the following:

$$\lambda f.\lambda x.\lambda y.(f\ x, f\ y)$$

If we just allow polymorphsim willy-nilly (i.e., we take the previous system but relax the restrictions on the form of types), we can construct two different types for this term:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \alpha \to \beta} \quad \dfrac{}{\Gamma \vdash x : \alpha}}{\Gamma \vdash f\ x : \beta} \quad \dfrac{\dfrac{}{\Gamma \vdash f : \alpha \to \beta} \quad \dfrac{}{\Gamma \vdash y : \alpha}}{\Gamma \vdash f\ y : \beta}}{\Gamma \vdash (f\ x, f\ y) : \beta \times \beta}}{\{f \mapsto \alpha \to \beta, x \mapsto \alpha\} \vdash \lambda y.(f\ x, f\ y) : \alpha \to \beta \times \beta}}{\{f \mapsto \alpha \to \beta\} \vdash \lambda x.\lambda y.(f\ x, f\ y) : \alpha \to \alpha \to \beta \times \beta}}{\emptyset \vdash \lambda f.\lambda x.\lambda y.(f\ x, f\ y) : (\alpha \to \beta) \to \alpha \to \alpha \to \beta \times \beta}}{\emptyset \vdash \lambda f.\lambda x.\lambda y.(f\ x, f\ y) : \forall\beta.(\alpha \to \beta) \to \alpha \to \alpha \to \beta \times \beta}}{\emptyset \vdash \lambda f.\lambda x.\lambda y.(f\ x, f\ y) : \forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha \to \alpha \to \beta \times \beta}$$

where $\Gamma = \{f \mapsto \alpha \to \beta, x \mapsto \alpha, y \mapsto \alpha\}$, or:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash f : \forall\gamma.\gamma \to \gamma}}{\Gamma' \vdash f : \alpha \to \alpha} \quad \dfrac{}{\Gamma' \vdash x : \alpha}}{\Gamma' \vdash f\ x : \alpha} \quad \dfrac{\dfrac{\dfrac{}{\Gamma' \vdash f : \forall\gamma.\gamma \to \gamma}}{\Gamma' \vdash f : \beta \to \beta} \quad \dfrac{}{\Gamma' \vdash y : \beta}}{\Gamma' \vdash f\ y : \beta}}{\Gamma' \vdash (f\ x, f\ y) : \alpha \times \beta}}{\{f \mapsto \forall\gamma.\gamma \to \gamma, x \mapsto \alpha\} \vdash \lambda y.(f\ x, f\ y) : \beta \to \alpha \times \beta}}{\{f \mapsto \forall\gamma.\gamma \to \gamma\} \vdash \lambda x.\lambda y.(f\ x, f\ y) : \alpha \to \beta \to \alpha \times \beta}}{\emptyset \vdash \lambda f.\lambda x.\lambda y.(f\ x, f\ y) : (\forall\gamma.\gamma \to \gamma) \to \alpha \to \beta \to \alpha \times \beta}}{\emptyset \vdash \lambda f.\lambda x.\lambda y.(f\ x, f\ y) : \forall\beta.(\forall\gamma.\gamma \to \gamma) \to \alpha \to \beta \to \alpha \times \beta}}{\emptyset \vdash \lambda f.\lambda x.\lambda y.(f\ x, f\ y) : \forall\alpha.\forall\beta.(\forall\gamma.\gamma \to \gamma) \to \alpha \to \beta \to \alpha \times \beta}$$

where $\Gamma' = \{f \mapsto \forall\gamma.\gamma \to \gamma, x \mapsto \alpha, y \mapsto \beta\}$.

To distinguish these cases, we'll introduce more explicit treatment of parametric polymorphism. Happily, this will return us to a syntax-directed system, albeit an unpleasantly verbose one.

$$\begin{aligned}
&\mathcal{A} \ni \alpha \qquad \mathcal{X} \ni x \\
&\mathcal{T} \ni t ::= \mathtt{Int} \mid t \to t \mid \alpha \mid \forall \alpha.t \\
&\mathcal{E} \ni e ::= z \mid e \odot e \mid x \mid \lambda x{:}t.e \mid e\ e \mid \Lambda\alpha.e \mid e\ t
\end{aligned}$$

We now have two new terms: $\Lambda\alpha.e$ introduces polymorphsm, and $e\ t$ eliminates it. These both work exactly parallel to functions: $\Lambda\alpha.e$ abstracts over the type α, just like $\lambda x.e$ abstracts over the expression x; $e\ t$ provides t as the argument to e, just as $e_1\ e_2$ provides e_2 as the argument to e_1. We're also going to be more explicit about the type of function arguments, as required by the example above.

Following that intuition, we can develop typing rules. Just as before we had an environment Γ that tracked the meaning of term variables, now we also have an environment Δ that tracks the meaning of type variables. We only have one possible meaning of type variables at this point, though, so Δ can just be a list of variables. We start out by describing when a type itself is well-formed, using a relation $- \vdash - \ \mathsf{type} \subseteq \mathcal{P}(\mathcal{A}) \times \mathcal{T}$.

$$\frac{}{\Delta \vdash \alpha\ \mathsf{type}}\ (\alpha \in \Delta) \qquad \frac{}{\Delta \vdash \mathtt{Int}\ \mathsf{type}} \qquad \frac{\Delta \vdash t_1\ \mathsf{type} \quad \Delta \vdash t_2\ \mathsf{type}}{\Delta \vdash t_1 \to t_2\ \mathsf{type}} \qquad \frac{\Delta, \alpha \vdash t\ \mathsf{type}}{\Delta \vdash \forall\alpha.t\ \mathsf{type}}$$

We can the define the typing relation for terms, $-;- \vdash - : - \subseteq \mathcal{P}(\mathcal{A}) \times (\mathcal{X} \rightharpoonup \mathcal{T}) \times \mathcal{E} \times \mathcal{T}$

$$\frac{}{\Delta;\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Delta;\Gamma \vdash e_1 : \mathtt{Int} \quad \Delta;\Gamma \vdash e_2 : \mathtt{Int}}{\Delta;\Gamma \vdash e_1 \odot e_2 : \mathtt{Int}} \qquad \frac{}{\Delta;\Gamma \vdash z : \mathtt{Int}}$$

$$\frac{\Delta \vdash t_1\ \mathsf{type} \quad \Delta;\Gamma, x : t_1 \vdash e : t_2}{\Delta;\Gamma \vdash \lambda x{:}t_1.e : t_1 \to t_2} \qquad \frac{\Delta;\Gamma \vdash e_1 : t_1 \to t_2 \quad \Delta;\Gamma \vdash e_2 : t_1}{\Delta;\Gamma \vdash e_1\ e_2 : t_2}$$

$$\frac{\Delta, \alpha;\Gamma \vdash e : t}{\Delta;\Gamma \vdash \Lambda\alpha.e : \forall\alpha.t} \qquad \frac{\Delta;\Gamma \vdash e : \forall\alpha.t' \quad \Delta \vdash t\ \mathsf{type}}{\Delta;\Gamma \vdash e\ t : t'[t/\alpha]}$$

Let's consider some example derivations. On the one hand, we can't derive something like $\emptyset;\emptyset \vdash \lambda x{:}\alpha.x :: \alpha \to \alpha$ any more; in the empty context, α isn't a type. But, we can derive the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\alpha\} \vdash \alpha\ \mathsf{type}} \quad \dfrac{}{\{\alpha\};\{x \mapsto \alpha\} \vdash x : \alpha}}{\{\alpha\};\emptyset \vdash \lambda x{:}\alpha.x : \alpha \to \alpha}}{\emptyset;\emptyset \vdash \Lambda\alpha.\lambda x{:}\alpha.x : \forall\alpha.\alpha \to \alpha}$$

and we can use it:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{\alpha\} \vdash \alpha\ \mathsf{type}} \quad \dfrac{}{\{\alpha\};\{x \mapsto \alpha\} \vdash x : \alpha}}{\{\alpha\};\emptyset \vdash \lambda x{:}\alpha.x : \alpha \to \alpha}}{\emptyset;\emptyset \vdash \Lambda\alpha.\lambda x{:}\alpha.x : \forall\alpha.\alpha \to \alpha} \qquad \dfrac{}{\emptyset \vdash \mathtt{Int}\ \mathsf{type}}}{\emptyset;\emptyset \vdash (\Lambda\alpha.\lambda x{:}\alpha.x)\ \mathtt{Int} : \mathtt{Int} \to \mathtt{Int}} \qquad \dfrac{}{\emptyset;\emptyset \vdash 4 : \mathtt{Int}}}{\emptyset;\emptyset \vdash (\Lambda\alpha.\lambda x{:}\alpha.x)\ \mathtt{Int}\ 4 : \mathtt{Int}}$$

We can return to our initial examples and see how they play out differently. Here's one:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta,\gamma \vdash \gamma\ \mathsf{type}}\quad\dfrac{}{\Delta,\gamma \vdash \gamma\ \mathsf{type}}}{\Delta,\gamma \vdash \gamma \to \gamma\ \mathsf{type}}}{\Delta \vdash \forall\gamma.\gamma \to \gamma\ \mathsf{type}}\quad\dfrac{\dfrac{}{\Delta \vdash \alpha\ \mathsf{type}}\quad\dfrac{\dfrac{}{\Delta \vdash \beta\ \mathsf{type}}\quad\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta;\Gamma' \vdash f : \forall\gamma.\gamma \to \gamma}}{\Delta;\Gamma' \vdash f\,\alpha : \alpha \to \alpha}\quad\dfrac{}{\Delta;\Gamma' \vdash x : \alpha}}{\Delta;\Gamma' \vdash f\,\alpha, x : \alpha}\quad\dfrac{\dfrac{\dfrac{}{\Delta;\Gamma' \vdash f : \forall\gamma.\gamma \to \gamma}}{\Delta;\Gamma' \vdash f\,\beta : \beta \to \beta}\quad\dfrac{}{\Delta;\Gamma' \vdash y : \beta}}{\Delta;\Gamma' \vdash f\,\beta\,y : \beta}}{\Delta;\Gamma' \vdash (f\,\alpha\,x, f\,\beta\,y) : \alpha \times \beta}}{\Delta;\{f \mapsto \forall\gamma.\gamma \to \gamma, x \mapsto \alpha\} \vdash \lambda y{:}\beta.(f\,\alpha\,x, f\,\beta\,y) : \beta \to \alpha \times \beta}}{\Delta;\{f \mapsto \forall\gamma.\gamma \to \gamma\} \vdash \lambda x{:}\alpha.\lambda y{:}\beta.(f\,\alpha\,x, f\,\beta\,y) : \alpha \to \beta \to \alpha \times \beta}}{\Delta;\emptyset \vdash \lambda f{:}\forall\gamma.\gamma \to \gamma.\lambda x{:}\alpha.\lambda y{:}\beta.(f\,\alpha\,x, f\,\beta\,y) : (\forall\gamma.\gamma \to \gamma) \to \alpha \to \beta \to \alpha \times \beta}}{\{\alpha\};\emptyset \vdash \Lambda\beta.\lambda f{:}\forall\gamma.\gamma \to \gamma.\lambda x{:}\alpha.\lambda y{:}\beta.(f\,\alpha\,x, f\,\beta\,y) : \forall\beta.(\forall\gamma.\gamma \to \gamma) \to \alpha \to \beta \to \alpha \times \beta}}{\emptyset;\Lambda\alpha.\Lambda\beta.\lambda f{:}\forall\gamma.\gamma \to \gamma.\lambda x{:}\alpha.\lambda y{:}\beta.(f\,\alpha\,x, f\,\beta\,y) \vdash \forall\alpha.\forall\beta.(\forall\gamma.\gamma \to \gamma) \to \alpha \to \beta \to \alpha \times \beta :}$$

where $\Gamma' = \{f \mapsto \forall\gamma.\gamma \to \gamma, x \mapsto \alpha, y \mapsto \beta\}$ and $\Delta = \{\alpha, \beta\}$

2. Explicit Polymorphism and Semantics

For most of the languages we've introduced, we've given syntax, semantics, and then typing. For the language with implicit polymorphism, we left out the semantics step. But, the expressions there were the same as the expressions in the λ-calculi we studied earlier in the semester, so I hoped that we could get away with not repeating their semantics. That is now no longer the case: our expressions include type abstractions and type applications, unlike anything we saw before.

For our first attempt to give a semantics, we can just follow the intuition that type abstractions $\Lambda\alpha.e$ and type applications $e\,t$ are "just like" value abstraction and application. We have to define substitution of types into expressions $-[-/-] \in \mathcal{E} \times \mathcal{T} \times \mathcal{A} \to \mathcal{E}$, but that is a completely unsurprising definition:

$$\begin{aligned}
x[t/\alpha] &= x & z[t/\alpha] &= z \\
(e_1 \odot e_2)[t/\alpha] &= e_1[t/\alpha] \odot e_2[t/\alpha] & (e_1\,e_2)[t/\alpha] &= e_1[t/\alpha]\,e_2[t/\alpha] \\
(\lambda x{:}t'.e)[t/\alpha] &= \lambda x{:}t'[t/\alpha].e[t/\alpha] & (e\,t')[t/\alpha] &= e[t/\alpha]\,t'[t/\alpha] \\
(\Lambda\beta.e)[t/\alpha] &= \begin{cases} \Lambda\beta.e[t/\alpha] & \text{if } \alpha \neq \beta \\ \Lambda\beta.e & \text{otherwise} \end{cases}
\end{aligned}$$

We have to extend our idea of values to incorporate type abstraction as well as value abstraction

$$\mathcal{V} \ni v ::= z \mid \lambda x{:}t.e \mid \Lambda\alpha.e$$

Finally, we can adapt the evaluation relation $\Downarrow \subseteq \mathcal{E} \times \mathcal{V}$:

$$\frac{}{v \Downarrow v} \qquad \frac{e_1 \Downarrow z_1 \quad e_2 \Downarrow z_2}{e_1 \odot e_2 \Downarrow z_1 \odot z_2} \qquad \frac{e_1 \Downarrow \lambda x{:}t.e \quad e_2 \Downarrow w \quad e[w/x] \Downarrow v}{e_1\,e_2 \Downarrow v} \qquad \frac{e \Downarrow \Lambda\alpha.e' \quad e'[t/\alpha] \Downarrow v}{e\,t \Downarrow v}$$

and we can derive the kinds of results we expect:

$$\dfrac{\dfrac{\dfrac{}{\Lambda\alpha.\lambda x{:}\alpha.x \Downarrow \Lambda\alpha.\lambda x{:}\alpha.x}\quad\dfrac{}{\lambda x{:}\mathtt{Int}.x \Downarrow \lambda x{:}\mathtt{Int}.x}}{(\Lambda\alpha.\lambda x{:}\alpha.x)\,\mathtt{Int} \Downarrow \lambda x{:}\mathtt{Int}.x}\quad\dfrac{}{4 \Downarrow 4}\quad\dfrac{}{x[4/x] \Downarrow 4}}{(\Lambda\alpha.\lambda x{:}\alpha.x)\,\mathtt{Int}\,4 \Downarrow 4}$$

However, you might not find this account totally satisfactory. In particular, while substituting values does something—eventually they get added, or multiplied, or applied, or returned—substituting types doesn't seem to do anything. In fact, we can capture this idea formally, using an idea called *erasure*. The idea is to define a function from the language with explicit polymorphism to the untyped λ-calculus. We then show that the evaluation relation for the explicitly polymorphic language is consistent with erasure—i.e., that the erased parts of the terms aren't actually effecting their outcomes.

Here's our erasure function:

$$er(x) = x \qquad er(z) = z$$
$$er(e_1 \odot e_2) = er(e_1) \odot er(e_2) \qquad er(e_1\ e_2) = er(e_1)\ er(e_2)$$
$$er(\lambda x{:}t.e) = \lambda x.er(e) \qquad er(\Lambda\alpha.e) = er(e) \qquad er(e\ t) = er(e)$$

Then we can tie evaluation and erasure together. Unfortunately, we can't tie them together as closely as you might expect: because our erasure function erases type abstractions, it may turn a value into a non-value. However, we could show the following:

Theorem 19.1. *If $e \Downarrow v$ and $er(e) \Downarrow w$ then $er(v) = w$.*

3. Relating Implicit and Explicit Polymorphism

We now have two different accounts of polymorphism: the implicit one, which supports principal types and type inference, and the explicit one, which types more programs but lacks principality or inference. You might wonder how these two systems are related: we know that there are explicitly-polymorphic expressions that have no implicit equivalent, but are there implicitly-polymorphic expressions that have no explicit equivalent? In fact, we can show that there aren't.

The idea is that we'll construct a relation between implicitly typed and explicitly typed expressions. We'll show that each implicitly-polymorphic expression (IPE) is related to *at least one* explicitly-polymorphic expressions. This function is going to be defined by induction on *typing derivations* of IPEs. In fact, we'll define it by augmenting the implicitly polymorphic type system with explicitly polymorphic terms. That is to say, we're going to describe a new judgment $- \vdash - \rightsquigarrow - : - \subseteq (\mathcal{X} \rightharpoonup \mathcal{S}) \times \mathcal{E}_i \times \mathcal{E}_e \times \mathcal{S}$ as follows:

$$\frac{}{\Gamma \vdash x \rightsquigarrow x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash z \rightsquigarrow z : \texttt{Int}} \qquad \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \texttt{Int} \quad \Gamma \vdash e_2 \rightsquigarrow e_2' : \texttt{Int}}{\Gamma \vdash e_1 \odot e_2 \rightsquigarrow e_1' \odot e_2' : \texttt{Int}}$$

$$\frac{\Gamma, x{:}t_1 \vdash e \rightsquigarrow e' : t_2}{\Gamma \vdash \lambda x.e \rightsquigarrow \lambda x{:}t_1.e' : t_1 \to t_2} \qquad \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : t_1 \to t_2 \quad \Gamma \vdash e_2 \rightsquigarrow e_2' : t_1}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : t_2}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : s}{\Gamma \vdash e \rightsquigarrow \Lambda\alpha.e' : \forall\alpha.s}\ (\alpha \notin fv(\Gamma)) \qquad \frac{\Gamma \vdash e \rightsquigarrow e' : \forall\alpha.s}{\Gamma \vdash e \rightsquigarrow e'\ t : s[t/\alpha]}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : s \quad \Gamma \vdash e_2 \rightsquigarrow e_2' : t}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow (\lambda x{:}s.e_2')e_1' : t}$$

Theorem 19.2.

- *If* $\Gamma \vdash e \rightsquigarrow e' : s$, *then* $\Gamma \vdash e : s$
- *If* $\Gamma \vdash e \rightsquigarrow e' : s$, *then* $fv(\Gamma, s); \Gamma \vdash e' : s$.

What is certainly not the case, however, is that for a given term e we have a single e' such that $\Gamma \vdash e \rightsquigarrow e' : s$. For a simple example, consider the expression $\lambda x.\lambda y.x$; we have the following derivations

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{x \mapsto \alpha, y \mapsto \beta\} \vdash x \rightsquigarrow x : \alpha}}{\{x \mapsto \alpha\} \vdash \lambda y.x \rightsquigarrow \lambda y{:}\beta.x : \beta \to \alpha}}{\emptyset \vdash \lambda x.\lambda y.x \rightsquigarrow \lambda x{:}\alpha.\lambda y{:}\beta.x : \alpha \to \beta \to \alpha}}{\emptyset \vdash \lambda x.\lambda y.x \rightsquigarrow \Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\beta.\alpha \to \beta \to \alpha}}{\emptyset \vdash \lambda x.\lambda y.x \rightsquigarrow \Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha}$$

and

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{x \mapsto \alpha, y \mapsto \beta\} \vdash x \rightsquigarrow x : \alpha}}{\{x \mapsto \alpha\} \vdash \lambda y.x \rightsquigarrow \lambda y{:}\beta.x : \beta \to \alpha}}{\emptyset \vdash \lambda x.\lambda y.x \rightsquigarrow \lambda x{:}\alpha.\lambda y{:}\beta.x : \alpha \to \beta \to \alpha}}{\emptyset \vdash \lambda x.\lambda y.x \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\alpha \to \beta \to \alpha}}{\emptyset \vdash \lambda x.\lambda y.x \rightsquigarrow \Lambda\beta.\Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\beta.\forall\alpha.\alpha \to \beta \to \alpha}$$

While we started with the same term, we've ended up with two different EPEs.

Theorem 19.3. *Suppose that* $\Gamma \vdash e \rightsquigarrow e_1 : s_1$ *and* $\Gamma \vdash e \rightsquigarrow e_2 : s_2$. *Then there is some term* f *such that:*

- $\{fv(\Gamma, s)\}; \Gamma \vdash f : s_1 \to s_2$
- $er(f) \equiv_\alpha \lambda x.x$