1 Type schemas

We saw last time that we could describe type inference by writing typing rules that introduce explicit *type variables* T to solve for:

$$\frac{}{\Gamma, x\!:\!\tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash b : B}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \quad \tau_0 = \tau_1 \to T}{\Gamma \vdash e_0\ e_1 : T} \qquad \frac{\Gamma, x\!:\!T \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : T \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x\!:\!\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2} \qquad \frac{\Gamma, x\!:\!T_1, y\!:\!T_1 \to T_2 \vdash e : \tau' \quad \tau' = T_2}{\Gamma \vdash \mathbf{rec}\ y.\ \lambda x.\, e : T_1 \to T_2}$$

This simple type inference mechanism does not result in as much *polymorphism*[1] as we would like. For example, consider a program that binds a variable f to the identity function, then applies it to both an **int** and a **bool**:

$$\begin{array}{l}\mathbf{let}\ f = \lambda x.\, x\ \mathbf{in} \\ \quad \mathbf{if}\ (f\ \mathbf{true})\ \mathbf{then}\ (f\ 3)\ \mathbf{else}\ (f\ 4)\end{array} \tag{1}$$

The type system above will find that the function f has some type $T \to T$, which means that it can act as if it had this type for any T. However, when the type checker encounters the application to **true**, it decides $T = \mathbf{bool}$ first and says that the function is of type $\mathbf{bool} \to \mathbf{bool}$. It then gives a unification error when it sees the **int** parameters 3 and 4. We would like f to be polymorphic, having type $\mathbf{bool} \to \mathbf{bool}$ when applied to a **bool** parameter and type $\mathbf{int} \to \mathbf{int}$ when applied to an **int** parameter.

The various versions of ML can do this. The trick is to bind variables like f not to types, but rather to *type schemas*. A type schema σ is a pattern for a type, which can mention type parameters α:

$$\sigma \quad ::= \quad \forall \alpha_1, \ldots, \alpha_n \,.\, \tau \quad {\scriptstyle (n \geq 0)}$$

The idea is that if a variable has a type schema mentioning type parameters $\alpha_1, \ldots, \alpha_n$, it is bound to a term that can act as though it has any type that looks like τ with the parameters α_i replaced by arbitrary types $\tau_1, \ldots, \tau_n$. For example, we give the variable f the type schema $\forall \alpha \,.\, \alpha \to \alpha$, and the type of the K combinator $\lambda xy.\, x$ (a.k.a. *FALSE*) is

$$\forall \alpha . \forall \beta \,.\, \alpha \to \beta \to \alpha.$$

1.1 Inferring type schemas

To incorporate type schemas into the type system, we extend Γ to bind variables to type schemas:

$$\Gamma = x_1\!:\!\sigma_1, \ldots, x_n\!:\!\sigma_n$$

Then the typing rule for variables *instantiates* the variable's type by replacing type parameters α with types. To make this work with type inference, these types are fresh type variables to be solved for:

$$\frac{}{\Gamma, x\!:\!\forall \alpha_1, \ldots, \alpha_n \,.\, \tau \vdash x : \tau\{T_1/\alpha_1, \ldots, T_n/\alpha_n\}}\ (\mathit{instantiation})$$

We extend the typing rule for **let** to correspondingly generate type schemas by generalizing over type parameters that appear only in the type of e_1 (that is, do not appear in Γ):

[1] Greek for "many shapes"

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x\!:\!\forall \alpha_1, \ldots, \alpha_n .\, \tau_1 \vdash e_2 : \tau_2 \quad \alpha_i \notin FTV(\Gamma) \quad ^{i\in 1..n}}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2} \; (generalization)$$

How are the parameters α_i chosen? The algorithm is to type-check e_1 using type variables as above. However, once the type τ_1 is found, and unification is used to solve all equations in the derivation of $\Gamma \vdash e_1 : \tau_1$, any *unsolved* type variables T that are not constrained by appearing elsewhere in the program could be replaced by any type. Therefore, we replace each such type variable in τ_1 with a corresponding type parameter α. While it doesn't in principle hurt to have extra type parameters, the usual approach is to generate a type parameter for each unsolved T that appears in τ_1 but not in Γ.

1.2 Example

Here is a derivation exposing the polymorphic type of K in this system:

$$\frac{\dfrac{\dfrac{x\!:\!\alpha, y\!:\!\beta \vdash x : \alpha}{x\!:\!\alpha \vdash \lambda y.\, x : \beta \to \alpha}}{\vdash \lambda x.\, \lambda y.\, x : \alpha \to \beta \to \alpha} \quad \dfrac{\ldots}{k\!:\!\forall \alpha, \beta .\, \alpha \to \beta \to \alpha \vdash e_2 : \tau_2}}{\vdash \textbf{let } k = \lambda x.\, \lambda y.\, x \textbf{ in } e_2 : \tau_2}$$

The type inference algorithm would proceed by computing a type $T_1 \to T_2 \to T_1$ for the variable k. Because neither T_1 nor T_2 would be mentioned in the typing context, it would replace them with the type variables α and β and give k the type schema $\forall \alpha .\forall \beta .\, \alpha \to \beta \to \alpha$ when type-checking e_2.

1.3 Limitations of let-polymorphism

The type systems of ML and Haskell are based on let-polymorphism. We previously considered **let** $x = e_1$ **in** e_2 to be equivalent to $(\lambda x.\, e_2)\ e_1$, but in SML, the former may be typable in some cases when the latter is not, e.g.:

```
- let val f = fn x => x in if (f true) then (f 3) else (f 4) end;
val it = 3 : int
- (fn f => if (f true) then (f 3) else (f 4)) (fn x => x);
stdIn:17.27-17.32 Error: operator and operand don't agree [literal]
  operator domain: bool
  operand:         int
  in expression:
    f 3
stdIn:17.38-17.43 Error: operator and operand don't agree [literal]
  operator domain: bool
  operand:         int
  in expression:
    f 4
```

2 System F

If we consider type schemas to be regular types, we get the language System F, introduced by Girard in 1971. This lets us pass polymorphic terms uninstantiated to functions.

In the Church-style simply-typed λ-calculus, we annotated binding occurrences of variables with their types. The corresponding version of the polymorphic λ-calculus is called *System F*. Here we explicitly abstract terms with respect to types and explicitly instantiate by applying an abstracted term to a type. We augment the syntax with new terms and types:

$$e \;::=\; \cdots \mid \Lambda \alpha.\, e \mid e[\tau] \qquad\qquad \tau \;::=\; b \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha .\, \tau$$

where b are the base types (e.g., **int** and **bool**). The new terms are *type abstraction* and *type application*, respectively. Operationally, we have

$$(\Lambda\alpha.\, e)[\tau] \quad \longrightarrow \quad e\{\tau/\alpha\}.$$

This just gives the rule for instantiating a type schema. Since these reductions only affects the types, they can be performed at compile time.

The typing rules for these constructs need a notion of well-formed type. We introduce a new environment Δ that maps type variables to their *kinds* (for now, there is only one kind: **type**). So Δ is a partial function with finite domain mapping types to $\{\mathbf{type}\}$. Since the range is only a singleton, all Δ does for right now is to specify a set of types, namely $\mathrm{dom}(\Delta)$ (it will get more complicated later). As before, we use the notation $\Delta,\, \alpha : \mathbf{type}$ for the partial function $\Delta[\mathbf{type}/\alpha]$. For now, we just abbreviate this by $\Delta,\, \alpha$.

We have two classes of type judgments:

$$\Delta \vdash \tau : \mathbf{type} \qquad\qquad \Delta;\, \Gamma \vdash e : \tau$$

For now, we just abbreviate the former by $\Delta \vdash \tau$. These judgments just determine when τ is well-formed under the assumptions Δ. The typing rules for this class of judgments are:

$$\Delta,\, \alpha \vdash \alpha \qquad\quad \Delta \vdash b \qquad\quad \frac{\Delta \vdash \sigma \quad \Delta \vdash \tau}{\Delta \vdash \sigma \to \tau} \qquad\quad \frac{\Delta,\, \alpha \vdash \tau}{\Delta \vdash \forall\alpha\,.\tau}$$

Right now, all these rules do is use Δ to keep track of free type variables. One can show that $\Delta \vdash \tau$ iff $FV(\tau) \subseteq \mathrm{dom}(\Delta)$.

The typing rules for the second class of judgments are:

$$\frac{\Delta \vdash \tau}{\Delta;\, \Gamma,\, x : \tau \vdash x : \tau} \qquad \frac{\Delta;\, \Gamma \vdash e_0 : \sigma \to \tau \quad \Delta;\, \Gamma \vdash e_1 : \sigma}{\Delta;\, \Gamma \vdash (e_0\ e_1) : \tau} \qquad \frac{\Delta;\, \Gamma,\, x : \sigma \vdash e : \tau \quad \Delta \vdash \sigma}{\Delta;\, \Gamma \vdash (\lambda x : \sigma.\, e) : \sigma \to \tau}$$

$$\frac{\Delta;\, \Gamma \vdash e : \forall\alpha\,.\tau \quad \Delta \vdash \sigma}{\Delta;\, \Gamma \vdash (e\ \sigma) : \tau\{\sigma/\alpha\}} \qquad \frac{\Delta,\, \alpha;\, \Gamma \vdash e : \tau \quad \alpha \notin FV(\Gamma)}{\Delta;\, \Gamma \vdash (\Lambda\alpha.\, e) : \forall\alpha\,.\tau}$$

One can show that if $\Delta;\, \Gamma \vdash e : \tau$ is derivable, then τ and all types occurring in annotations in e are well-formed. In particular, $\vdash e : \tau$ only if e is a closed term and τ is a closed type, and all type annotations in e are closed types.