1 Recap

In the last lecture, we saw how to unify types.

\[
\begin{align*}
\text{Unify}(\emptyset) & \triangleq I \\
\text{Unify}(\alpha = \alpha, E) & \triangleq \text{Unify}(E) \\
\text{Unify}(\alpha = \tau, E) & \triangleq \{\tau/\alpha\} \cdot \text{Unify}(E{\tau/\alpha}), \ \alpha \notin FV(\tau) \\
\text{Unify}(\sigma_1 \rightarrow \tau_1 = \sigma_2 \rightarrow \tau_2, E) & \triangleq \text{Unify}(\sigma_1 = \sigma_2, \tau_1 = \tau_2, E)
\end{align*}
\]

where \(I\) is the identity substitution and substitutions are applied from left to right, so the composition \(ST\) means: do \(S\) first, then \(T\).

2 Polymorphic \(\lambda\)-Calculus

Suppose we have base types \(\text{Int}\) and \(\text{Bool}\). The problem with the simple type inference mechanism that we have presented is that we do not have quite as much polymorphism\(^1\) as we would like. For example, consider a program that binds a variable to the identity function, then applies it to an \(\text{Int}\) and also to a \(\text{Bool}\).

\[
\text{let } f = x : x \in \\
\text{if } (f \text{ true}) \text{ then } (f \ 3) \text{ else } (f \ 4)
\]

(1)

The type checker encounters the \(\text{Bool}\) first and says that the function is of type \(\text{Bool} \rightarrow \text{Bool}\), then gives an error when it sees the \(\text{Int}\) parameter, whereas we really want it to be interpreted as type \(\text{Bool} \rightarrow \text{Bool}\) when applied to a \(\text{Bool}\) parameter and \(\text{Int} \rightarrow \text{Int}\) when applied to an \(\text{Int}\) parameter.

We can handle this by introducing a new type constructor that quantifies over types.

\[
\tau \ ::= \ \text{Int} \mid \text{Bool} \mid \alpha \mid \sigma \rightarrow \tau \mid \forall \alpha. \tau
\]

(2)

The type \(\forall \alpha. \tau\) can be viewed as a polymorphic type or type schema, a pattern with type variables that can be instantiated to obtain actual types. For example, the polymorphic type of the identity function will be the type schema

\[
\forall \alpha. \alpha \rightarrow \alpha
\]

and the type of the \(K\) combinator \(\lambda x y. x\) will be

\[
\forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha.
\]

There will be rules that allow us to delay the instantiation of the type variables until the function is applied. Thus we can interpret the identity function as \(\text{Int} \rightarrow \text{Int}\) or \(\text{Bool} \rightarrow \text{Bool}\) depending on context.

The resulting language is called the polymorphic \(\lambda\)-calculus or System F. In this new language, the terms and evaluation rules are the same, but the types are defined by (2). All the terms that were previously well-typed will still be well-typed, but there will be more well-typed terms than before; for example, (1).

\(^1\)Greek for “many forms”
3 Typing Rules

In addition to the old typing rules

\[ \frac{\Gamma \vdash n : \text{Int} \quad \text{(and similarly for other constants)}}{\Gamma, x : \tau \vdash x : \tau} \]

we add the following two new rules for polymorphic types:

\[ \frac{\Gamma \vdash e : \tau \quad \alpha \notin \text{FV}(\Gamma)}{\Gamma \vdash e : \forall \alpha. \tau} \quad \frac{\Gamma \vdash e : \tau \quad \alpha \notin \text{FV}(\Gamma)}{\Gamma \vdash e : \forall \alpha. \tau} \]

These are called the generalization rule and the instantiation rule, respectively.

The notation \( \tau\{\sigma/\alpha\} \) refers to the safe substitution of the type \( \sigma \) for the type variable \( \alpha \) in \( \tau \). Here the binding operator \( \forall \alpha \) binds the type variable \( \alpha \) in the same way that \( \lambda x \) binds the variable \( x \) in \( \lambda \)-terms, and the notions of scope, free and bound variables are the same. In particular, one can \( \alpha \)-convert type variables as necessary to avoid the capture of free type variables when performing substitutions.

The premise of the generalization rule includes the proviso \( \alpha \notin \text{FV}(\Gamma) \). The idea here is that the type judgment \( \Gamma \vdash e : \tau \) must hold without any assumptions involving \( \alpha \); if so, then we can conclude that \( \alpha \) could have been any type \( \sigma \), and the type judgment \( \Gamma \vdash e : \tau\{\sigma/\alpha\} \) would also hold.

4 Examples

Here is a derivation of the polymorphic type of \( K \) in this system.

\[ \frac{x : \alpha, y : \beta \vdash x : \alpha \quad x : \alpha \vdash \lambda y. x : \beta \to \alpha} {\vdash \lambda x. \lambda y. x : \alpha \to \beta \to \alpha} \]

Starting from \( x : \alpha, y : \beta \vdash x : \alpha \), two applications of the abstraction rule yield \( \vdash \lambda x. \lambda y. x : \alpha \to \beta \to \alpha \), then two applications of the generalization rule yield \( \vdash \lambda x. \lambda y. x : \forall \alpha. \forall \beta. \alpha \to \beta \to \alpha \).

Some terms are typable in this system that were not typable before. For example, the term \( \lambda x. xx \) is typable:

\[ \frac{x : \forall \alpha. \alpha \vdash x : \forall \alpha. \alpha \quad x : \forall \alpha. \alpha \vdash x : \forall \alpha. \alpha} {\vdash \lambda x. xx : \forall \alpha. \alpha \to \beta} \]

Unfortunately, this type is not too meaningful, because nothing has type \( \forall \alpha. \alpha \). This type is said to be uninhabited, and we give it a name: Void. However, by a similar argument, we can show that \( \lambda x. xx \) also has type \( \forall \beta. (\forall \alpha. \alpha \to \beta) \to (\beta \to \beta) \), which is meaningful.
Although $\lambda x. xx$ is typable, the paradoxical combinator $\Omega = (\lambda x. xx) (\lambda x. xx)$ is not, and neither is the $Y$ combinator. This is because the language is still strongly normalizing. This means that the polymorphic $\lambda$-calculus is not Turing complete, that is, it cannot simulate arbitrary Turing machines.

Worse, types inference is undecidable, so the programmer must sometimes provide types.

5 Let-Polymorphism

We can regain decidability of type inference by placing some restrictions on the use of the type quantifier $\forall \alpha$. Specifically, we will only allow it at the top level; that is, we will only allow polymorphic type expressions of the form $\forall \alpha_1 \cdots \forall \alpha_n, \tau$, where $\tau$ is quantifier-free:

\[
\begin{align*}
\text{quantifier-free terms} & \quad \tau ::= \text{Int} \mid \text{Bool} \mid \alpha \mid \tau_1 \to \tau_2 \\
\text{polymorphic terms} & \quad \pi ::= \tau \mid \forall \alpha. \pi
\end{align*}
\]

We will also modify our rules so that it can only be introduced in the context of a let statement. Thus we will modify our definition of terms to include a let statement:

\[
e ::= \cdots \mid \text{let } x = e_1 \text{ in } e_2
\]

and replace the generalization rule with the let rule

\[
\begin{array}{c}
\Gamma \vdash d: \sigma \\
\Gamma, x: \forall \alpha_1 \cdots \forall \alpha_n. \sigma \vdash e: \tau \\
\{\alpha_1, \ldots, \alpha_n\} = FV(\sigma) - FV(\Gamma)
\end{array}
\]

\[
\Gamma \vdash \text{let } x = d \text{ in } e: \tau
\]

So type schemas are only used to type let expressions. For this reason, this approach is called let-polymorphism.

6 Let-Polymorphism and ML

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

\[
\begin{array}{l}
\# \text{ let } f = \text{fun } x \to x \text{ in if (f true) then (f 3) else (f 4)};; \\
- : \text{int} = 3
\end{array}
\]

\[
\begin{array}{l}
\# (\text{fun } f \to \text{if (f true) then (f 3) else (f 4)}) (\text{fun x -> x});; \\
\text{Error: This expression has type int but an expression was expected of type bool}
\end{array}
\]

In theory, let-polymorphism can cause the type checker to run in exponential time, but in practice this is not a problem.
In the Church-style simply-typed $\lambda$-calculus, we annotated binding occurrences of variables with their types. The corresponding version of the polymorphic $\lambda$-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:

$$
e ::= \cdots | \Lambda \alpha. e | e \tau \quad \tau ::= b | \tau_1 \rightarrow \tau_2 | \alpha | \forall \alpha. \tau$$

where $b$ are the base types (e.g., `Int` and `Bool`). The new terms are type abstraction and type application, respectively. Operationally, we have

$$(\Lambda \alpha. e) \tau \rightarrow e[\tau/\alpha].$$

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 $\Delta$ that maps type variables to their kinds (for now, there is only one kind: type). So $\Delta$ is a partial function with finite domain mapping types to $\{\text{type}\}$. Since the range is only a singleton, all $\Delta$ does for right now is to specify a set of types, namely $\text{dom}(\Delta)$ (it will get more complicated later). As before, we use the notation $\Delta; \alpha : \text{type}$ for the partial function $\Delta[\text{type}/\alpha]$. For now, we just abbreviate this by $\Delta, \alpha$.

The type system has two classes of judgments:

$$\Delta \vdash \tau : \text{type} \quad \Delta; \Gamma \vdash e : \tau$$

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

$$\begin{align*}
\Delta, \alpha \vdash \alpha & \quad \Delta \vdash b & \quad \Delta \vdash \sigma & \Delta \vdash \tau & \quad \Delta, \alpha \vdash \tau \\
\Delta \vdash \sigma \rightarrow \tau & \quad \Delta \vdash \forall \alpha. \tau
\end{align*}$$

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

The typing rules for the second class of judgments are:

$$\begin{align*}
\Delta \vdash \tau & \quad \Delta; \Gamma \vdash \sigma \rightarrow \tau & \quad \Delta; \Gamma \vdash e_1 : \sigma & \quad \Delta; \Gamma \vdash e_0 : \sigma & \quad \Delta \vdash \sigma \\
\Delta; \Gamma \vdash (e_0 e_1) : \tau & \quad \Delta; \Gamma \vdash (\lambda x : \sigma. e) : \sigma \rightarrow \tau \\
\Delta; \Gamma \vdash e : \forall \alpha. \tau & \quad \Delta \vdash \sigma & \quad \Delta, \alpha; \Gamma \vdash e : \tau & \quad \alpha \notin \text{FV}(\Gamma) \\
\Delta; \Gamma \vdash (e \sigma) : \tau[\sigma/\alpha] & \quad \Delta, \alpha; \Gamma \vdash (\lambda \alpha. e) : \forall \alpha. \tau
\end{align*}$$

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