1 Recap

Last lecture we saw how to unify types.

$$\begin{aligned}
\mathsf{Unify}(\varnothing) &\triangleq I \\
\mathsf{Unify}(\alpha = \alpha, E) &\triangleq \mathsf{Unify}(E) \\
\mathsf{Unify}(\alpha = \tau, E) &\triangleq \{\tau/\alpha\} \cdot \mathsf{Unify}(E\{\tau/\alpha\}), \quad \alpha \notin FV(\tau) \\
\mathsf{Unify}(\sigma_1 \to \tau_1 = \sigma_2 \to \tau_2, E) &\triangleq \mathsf{Unify}(\sigma_1 = \sigma_2, \tau_1 = \tau_2, E)
\end{aligned}$$

where I is the identity substitution $\alpha \mapsto \alpha$. Substitutions are applied from left to right, so the composition $S\,T$ means: do S first, then do T.

2 Polymorphic λ-Calculus

Suppose we have base types int and 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 int and also to a bool.

$$\mathsf{let}\ f = \lambda x.\,x\ \mathsf{in}\ \mathsf{if}\ (f\ \mathsf{true})\ \mathsf{then}\ (f\ 3)\ \mathsf{else}\ (f\ 4) \tag{1}$$

The type checker encounters the bool first and says that the function is of type bool $\to$ bool, then gives an error when it sees the int parameter, whereas we really want it to be interpreted as type bool $\to$ bool when applied to a bool parameter and int $\to$ int when applied to an int parameter.

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

$$\tau \ ::=\ \mathsf{int} \ \mid\ \mathsf{bool} \ \mid\ \alpha \ \mid\ \sigma \to \tau \ \mid\ \forall \alpha.\,\tau \tag{2}$$

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

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

and the type of the K combinator $\lambda xy.\,x$ will be

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

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 int $\to$ int or bool $\to$ bool depending on context.

The resulting language is called the *polymorphic λ-calculus*. 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

$$\Gamma \vdash n : \mathsf{int} \quad \text{(and similarly for other constants)} \qquad\qquad \Gamma,\, x:\tau \vdash x:\tau$$

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

we add the following two new rules for polymorphic types:

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

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

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

The generalization rule includes the side condition $\alpha \notin FV(\Gamma)$. The idea here is that the type judgment $\Gamma \vdash e : \tau$ must hold without any assumptions involving α; if so, then we can conclude that α could have been any type σ, 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.

$$\cfrac{\cfrac{\cfrac{\cfrac{x:\alpha,\, y:\beta \vdash x:\alpha}{x:\alpha \vdash \lambda y.\, x : \beta \to \alpha}}{\vdash \lambda x.\, \lambda y.\, x : \alpha \to \beta \to \alpha}}{\vdash \lambda x.\, \lambda y.\, x : \forall \beta.\, \alpha \to \beta \to \alpha}}{\vdash \lambda x.\, \lambda y.\, x : \forall \alpha.\forall \beta.\, \alpha \to \beta \to \alpha}$$

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:

$$\cfrac{\cfrac{\cfrac{\cfrac{x:\forall\alpha.\alpha \vdash x:\forall\alpha.\alpha}{x:\forall\alpha.\alpha \vdash x:\alpha\to\beta} \quad \cfrac{x:\forall\alpha.\alpha \vdash x:\forall\alpha.\alpha}{x:\forall\alpha.\alpha \vdash x:\alpha}}{x:\forall\alpha.\alpha \vdash xx:\beta}}{\vdash \lambda x.\, xx : (\forall\alpha.\alpha) \to \beta}}{\vdash \lambda x.\, xx : \forall\beta.\, (\forall\alpha.\alpha) \to \beta}$$

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 $\forall\beta.\, (\forall\alpha.\alpha\to\alpha) \to (\beta\to\beta)$, which *is* meaningful.

$$
\dfrac{\dfrac{\dfrac{x:\forall\alpha.\alpha\to\alpha\vdash x:\forall\alpha.\alpha\to\alpha}{x:\forall\alpha.\alpha\to\alpha\vdash x:(\beta\to\beta)\to(\beta\to\beta)} \quad \dfrac{x:\forall\alpha.\alpha\to\alpha\vdash x:\forall\alpha.\alpha\to\alpha}{x:\forall\alpha.\alpha\to\alpha\vdash x:\beta\to\beta}}{\dfrac{x:\forall\alpha.\alpha\to\alpha\vdash xx:\beta\to\beta}{\vdash \lambda x.\,xx:(\forall\alpha.\alpha\to\alpha)\to(\beta\to\beta)}}}{\vdash \lambda x.\,xx:\forall\beta.(\forall\alpha.\alpha\to\alpha)\to(\beta\to\beta)}
$$

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 λ-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 $\forall\alpha_1\ldots\forall\alpha_n.\tau$, where τ is quantifier-free:

$$
\begin{array}{ll}
\text{quantifier-free terms} & \tau \ ::= \ \mathsf{int} \ \mid \ \mathsf{bool} \ \mid \ \alpha \ \mid \ \tau_1 \to \tau_2 \\
\text{polymorphic terms} & \pi \ ::= \ \tau \ \mid \ \forall\alpha.\pi
\end{array}
$$

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:

$$
e \ ::= \ \cdots \ \mid \ \mathsf{let}\ x = e_1 \ \mathsf{in}\ e_2
$$

and replace the generalization rule with the let rule

$$
\frac{\Gamma\vdash d:\sigma \qquad \Gamma,\, x:\forall\alpha_1\ldots\forall\alpha_n.\sigma\vdash e:\tau}{\Gamma\vdash \mathsf{let}\ x = d\ \mathsf{in}\ e:\tau}\ (\{\alpha_1,\ldots,\alpha_n\} = FV(\sigma) - FV(\Gamma))
$$

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

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

```
# let f = fun x -> x in if (f true) then (f 3) else (f 4);;
- : int = 3
# (fun f -> if (f true) then (f 3) else (f 4)) (fun x -> x);;
Error: This expression has type int but an expression was expected of type
     bool
```

In theory, let-polymorphism can cause the type checker to run in exponential time, but in practice this is not a problem.

6 System F

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

$$(\Lambda\alpha.\, e)\,\tau \;\to\; e\{\tau/\alpha\}. \tag{3}$$

This just gives the rule for instantiating a type schema. Since these reductions only affect 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, namely type, so Δ is a partial function with finite domain mapping type variables to $\{\mathsf{type}\}$. Since the range is only a singleton, all Δ does for now is to specify a set of types, namely $\mathsf{dom}\,\Delta$ (it will get more complicated later). As before, we use the notation $\Delta,\, \alpha : \mathsf{type}$ for the partial function $\Delta[\mathsf{type}/\alpha]$. For now, we just abbreviate this by $\Delta,\, \alpha$.

The type system has two classes of judgments:

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

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:

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

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

The typing rules for the second class of judgments are:

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

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

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.

For example, the polymorphic identity function is $\Lambda\alpha.\, \lambda x : \alpha.\, x$, which has polymorphic type $\forall\alpha.\alpha \to \alpha$ according to the following proof:

$$\cfrac{\cfrac{\cfrac{\alpha \vdash \alpha}{\alpha;\, x : \alpha \vdash x : \alpha} \qquad \alpha \vdash \alpha}{\alpha;\, \vdash (\lambda x : \alpha.\, x) : \alpha \to \alpha}}{\vdash (\Lambda\alpha.\, \lambda x : \alpha.\, x) : \forall\alpha.\alpha \to \alpha}$$

To apply this function to a value of a particular type, one must explicitly instantiate the type using (3):

$$((\Lambda\alpha.\, \lambda x : \alpha.\, x)\ \mathsf{int})\ 3 \;\to\; (\lambda x : \mathsf{int}.\, x)\ 3 \;\to\; 3.$$