CS611 Lecture 32: Type Inference, ML(predicative) polymorphism 11/12/99

Chaitanya Swamy

1 Introduction

In $\lambda^{\rightarrow}$(and $\lambda^{\rightarrow\times+}$) we annotated the argument x in the expression $\lambda x : \tau.e$ with the type τ. We would like to specify the minimum set of type annotations necessary with which the type can be inferred. For example, in ML the expression – `fn f => fn x => fn y => if (f x y) then (f (x+1) y) else (f x (f x y))` is inferred to have type $(int \rightarrow bool \rightarrow bool) \rightarrow int \rightarrow bool \rightarrow bool$ without any type annotations. Here we could infer `x:`*int* because `(x+1)` is a subexpression and so on.

2 Type Inference and Reconstruction

2.1 Unification

We change our syntax slightly and write λ-expressions as $\lambda x.e$. The typing rule is also changed to be :

$$\frac{\Gamma[x \mapsto \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$

The problem in implementing the typing rule above is that the type checker has to pull out a correct τ from thin air. To avoid this, we use *type variables* to denote the type of expressions whose type we don't yet know. Finally we will end up with a set of equations involving these type variables. The solution will be a set of substitutions for the type variables which satisfies the equations consistently(if the expression is typeable).
A substitution S is **weaker** than S' if there exists another substitution S'' s.t. $S' = S'' \circ S$. The **unification** of two expressions finds the *weakest substitution*(if there exists one) that needs to be done to make the two expressions equal.
For example, the unification of $(t_0 \rightarrow t_1) \rightarrow t_2$ and $t_3 \rightarrow (b \rightarrow c)$ gives $S = [t_2 \mapsto (b \rightarrow c), t_3 \mapsto (t_0 \rightarrow t_1)]$. Another possible substitution is $S' = [t_2 \mapsto (b \rightarrow c), t_0 \mapsto bool, t_3 \mapsto (bool \rightarrow t_1)]$. However S is weaker than S' because $S' = [t_0 \mapsto bool] \circ S$.
We define a function `unify` : (set of equations) $\rightarrow$(set of substitions, S)+fail.

$$\begin{aligned}
\texttt{unify}(\phi) &= \phi \\
\texttt{unify}(E \cup \{t = \tau\}) &= \texttt{unify}(E[\tau/t]) \circ [t \mapsto \tau] \quad \text{if } t \text{ does not occur in } \tau \\
\texttt{unify}(E \cup \{b_1 = b_2\}) &= \begin{cases} \texttt{unify}(E) & \text{if } b_1 = b_2 \text{ where } b_1, b_2 \text{ are base types} \\ \texttt{fail} & \text{otherwise} \end{cases} \\
\texttt{unify}(E \cup \{\tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4\}) &= \texttt{unify}(E \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\})
\end{aligned}$$

2.2 Polymorphism

Another feature of ML is polymorphism. In ML, the expression – `let val f=fn x => x in if (f true) then (f 1) else (f 2)` type-checks and returns 1. The idea is that `f` is a polymorphic function with type $\forall t.t \rightarrow t$. We introduce an ML-like language in which expressions can have such polymorphic types and *type schema* which can have such for-all quantifiers.

$$\begin{array}{llll}
\text{expressions} & : & e & ::= & n \mid x \mid \texttt{fn}\ x\ e \mid e_0\ e_1 \mid \texttt{if}\ e_0\ e_1\ e_2 \mid \texttt{letrec}\ (x\ e_0)\ e_1 \\
\text{base types} & : & B & ::= & int \mid bool \mid unit \\
\text{types} & : & \tau & ::= & B \mid \tau_0 \rightarrow \tau_1 \\
\text{type schema} & : & \sigma & ::= & \tau \mid \forall t.\sigma \\
\text{type context} & : & \Gamma & : & Var \rightarrow \sigma
\end{array}$$

The typing rule for `fn` x e is the same as the rule for $\lambda x.e$. The rules for `if` and `letrec` are as follows :

$$\frac{\Gamma \vdash e_0 : bool \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e_0\ e_1\ e_2 : \tau}$$

$$\frac{[x \mapsto \tau'] \vdash e_0 : \tau' \quad \Gamma[x \mapsto \tau'] \vdash e_1 : \tau}{\Gamma \vdash \texttt{letrec}\ (x\ e_0)\ e_1 : \tau} \quad \text{if } FTV(\tau') = \phi$$

$$\frac{[x \mapsto \tau'] \vdash e_0 : \tau' \quad FTV(\tau') - FTV(\Gamma) = t_1 \ldots t_n \quad \Gamma[x \mapsto \forall t_1 \ldots t_n.\tau'] \vdash e_1 : \tau}{\Gamma \vdash \texttt{letrec}\ (x\ e_0)\ e_1 : \tau}$$

2.3 Type inference/reconstruction algorithm

We give an algorithm $\mathcal{W}$ which given an expression e, type context Γ and a set of substitutions S returns the type τ of the expression e and a set of substitutions S', which need to be done so that e has type τ.

$$\begin{array}{rcl}
\mathcal{W}(n, \Gamma, S) & = & \langle int,\ S\rangle \\
\mathcal{W}(\texttt{true}, \Gamma, S) & = & \langle bool,\ S\rangle \\
\mathcal{W}(\texttt{false}, \Gamma, S) & = & \langle bool,\ S\rangle \\
\mathcal{W}(x, \Gamma, S) & = & \text{case } \Gamma(x) \text{ of} \\
 & & \quad \tau \qquad\qquad => \langle \tau,\ S\rangle \\
 & & \quad \forall t_1 \ldots t_n.\tau \quad => \langle \tau[t_f^1/t_1, t_f^2/t_2, \ldots t_f^n/t_n],\ S\rangle \quad \text{where } t_f^i\text{s are fresh type variables} \\
\mathcal{W}(e_0\ e_1, \Gamma, S) & = & \text{let } \langle \tau_0,\ S'\rangle = \mathcal{W}(e_0, \Gamma, S) \text{ in} \\
 & & \text{let } \langle \tau_1,\ S''\rangle = \mathcal{W}(e_1, \Gamma, S') \text{ in} \\
 & & \text{let } S''' = \texttt{unify}(\{\tau_0 = \tau_1 \to t_f\}, S'') \text{ in } \langle t_f,\ S'''\rangle \\
\text{and } \texttt{unify}(E, S) & = & \texttt{unify}(\underline{S}E) \circ S \quad \text{where the operator } \underline{S} \text{ makes the substitutions given by } S \text{ in } E
\end{array}$$