Scribe: Ryan Williams and Niranjan Nagarajan Lecturer: Andrew Myers

1 Introduction

In the strictly typed languages that we have studied so far we have used type declarations to specify the types of various expressions. But from a programmer's point of view this requirement is a nuisance and it would be nice if we could automatically infer the intended types of expressions. This process is called type inference and in effect involves constructing a proof tree for the type of an expression. To motivate this idea let us consider an example:

```
let
  square = rec s. fn z. z*z
in
  (fn f. fn x. fn y.
     if (f x y)
       (f (square x) y)
       (f x (f x y)))
```

How do we infer the type of this expression? Well, we first note that z is constrained to be of type *int* as the $*$ operator is applied to it. This forces square to have type $int \rightarrow int$ and x to have type *int*. We can also see that (f x y) needs to be of type *bool* (as it appears as the test in an if-expression) and since y and(f x y) are used in the same positions in applications of f, y needs to have the same type i.e. *bool*. Therefore f must have type $int \rightarrow bool \rightarrow bool$ and the type of the entire expression is $(int \rightarrow bool \rightarrow bool) \rightarrow int \rightarrow bool \rightarrow bool$.

The process that we have discussed above can be formalized. We start off by running the ordinary type-checking algorithm on the given expression. In places where the type of an expression is not known we use new type variables instead. The type-checking algorithm would then place restrictions on the *type variables* in the form of *type equations*. The resulting set of type equations is then $unified$ to determine the type of the expression. An illustration of this process in action is given by the following proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{T2 = int \rightarrow T6}{f{:}T2,\ x{:}T5 \vdash f{:}int \rightarrow T6} \quad \dfrac{}{f{:}T2,\ x{:}T5 \vdash 1{:}int}}{f{:}T2,\ x{:}T5 \vdash (f\ 1){:}T6}}{\dfrac{f{:}T2 \vdash (fn\ x.\ (f\ 1)){:}T1(= T5 \rightarrow T6)}{(fn\ f.\ fn\ x.\ (f\ 1)){:}T2 \rightarrow T1}} \quad \dfrac{\dfrac{}{y{:}T3 \vdash y{:}T4(= T3)}}{(fn\ y.\ y){:}T2(= T3 \rightarrow T4)}}{(fn\ f.\ fn\ x.\ (f\ 1))\ (fn\ y.\ y){:}T1}$$

The equations $T2 = T3 \rightarrow T4,\ T1 = T5 \rightarrow T6,\ T2 = int \rightarrow T6$ and $T4 = T3$ then need to be unified to get T1, the type of the initial expression, and this process will be described in the next section.

2 Unification

In our system of type inference, the types our derivations generate are **type expressions**. The set of type expressions may be defined inductively:

- T_i (a type variable) is a type expression, where i is a natural number.
- If τ_1, t_2 are a type expressions, then $\tau_1 \rightarrow t_2$ is also a type expression.

Throughout this section, τ_1 and τ_2 will denote arbitrary type expressions. A **type equation** will be an expression of the form $\tau_1 = \tau_2$.

After performing a type derivation using our type inference rules above, we obtain a pile of type equations for which we wish to find a solution, i.e. types to substitute for the type variables such that the equations are satisfied.

Given an equation $\tau_1 = t_2$, we wish to match components of the structure of the two expressions in some way (*unifying* them), generating more type equations which will allow us to solve for the unknowns. For example, suppose $\tau_1 = int \to T_1$ and $\tau_2 = T_2 \to (T_3 \to bool)$. The new equations we may infer from the structure of these two expressions are $T_2 = int$ and $T_1 = T_3 \to bool$.

The algorithm we will describe for performing unification is Robinson's (1965). Given a set of type equations E, it returns the weakest substitution of types for type variables that satisfies all the equations of E. "Weakest" means that it contains the fewest number of assignments from type variables to types. Let us formally define what we mean:

Definition. A **substitution** S is a partial mapping from the set of type variables to types, which is only defined for a finite number of type variables. Thus, there are always infinitely many T_i for which $S(T_i)$ is not undefined.

Definition. Let S_1 and S_2 be substitutions. S_1 is **weaker** than S_2 if there exists a substitution S_3 which is not the identity function and $S_2 = S_3 \circ S_1$. Here, $\circ$ acts not as function composition, but as an map "extender". S_3 extends S_1 in the following sense:

$$\begin{aligned} S_3 \circ S_1(T_i) &= S_1(T_i) && \text{if } S_1(T_i) \text{ is defined} \\ &= S_3(T_i) && \text{if } S_3(T_i) \text{ is defined} \\ &= \textit{undefined} && \text{otherwise} \end{aligned}$$

For an example, suppose we wish to unify $E = \{(\tau_0 \to \tau_1) \to \tau_2, \tau_3 \to (\mathsf{bool} \to \mathsf{int})\}$, and find that the substitution $S = [\tau_0 \mapsto \mathsf{bool}, \tau_2 \mapsto (\mathsf{bool} \to \mathsf{int}), \tau_3 \mapsto (\mathsf{bool} \to \tau_1)]$ will satisfy the equations of E. However, $S' = [\tau_2 \mapsto (\mathsf{bool} \to \mathsf{int}), \tau_3 \mapsto (\tau_0 \to \tau_1)]$ is also sufficient for unification (this substitution causes both sides of the equation to be the same syntactically), and S' is weaker than S since $S' = [\tau_0 \mapsto \mathsf{bool}] \circ S$.

Definition. The **weakest substitution** for E will be S such that for all $S' \neq S$ which are sufficient for unifying the equations of E, there exists a non-identity S'', such that $S' = S'' \circ S$.

Let E be a set of type equations. $\mathbf{Unify}(E)$ will return the weakest substitution that satisfies the equations of E. The inductive definition of **Unify** is as follows:

$$\begin{aligned} \mathbf{Unify}(\emptyset) &= \emptyset \\ \mathbf{Unify}(\{T = \tau\} \cup E) &= \mathbf{Unify}(E\{\tau/T\}) \circ [T \mapsto \tau])(^*) \\ \mathbf{Unify}(\{B = B\} \cup E) &= \mathbf{Unify}(E)(^{**}) \\ \mathbf{Unify}(\{B = B'\} \cup E) &= \mathsf{fail} \quad (\mathsf{if} B \neq B') \\ \mathbf{Unify}(\{T = T\} \cup E) &= \mathbf{Unify}(E) \\ \mathbf{Unify}(\{\tau_1 \to \tau_2 = \tau_3 \to \tau_4\} \cup E) &= \mathbf{Unify}(\{\tau_1 = \tau_3, \tau_2 = \tau_4\} \cup E) \end{aligned}$$

(*) where T is a type variable, given that T is not free in the type expression τ.
(**) where B is a base type.

Note that the procedure **Unify** terminates on any E:

For every inductive step of the procedure above, we either

(1) reduce the number of equations for types, or

(2) reduce the equations to cases which will make the number of equations smaller (the last case in the inductive definition above).

3 Type reconstruction

We wish to develop an algorithm $\mathcal{R}$ which will actually perform the type inference for us, using the unification procedure given above. $\mathcal{R}(e, \Gamma, S) = \langle \tau, S' \rangle$ will indicate that when reconstruct the type of e given a type context Γ and substitution S, we infer that e is of type τ with respect to S', a substitution that is stronger than S. More precisely, S' is the weakest substitution that is stronger than S and $S'(\Gamma) \vdash e : S'(\tau)$, which brings us to the definition of substitution on a set of expressions:

Definition. Let S be a substitution defined on the type variables $T_{i_1}, \ldots, T_{i_n}$. For an arbitrary set of expressions A, $S(A) \equiv \{e\{S(T_{i_j})/T_{i_j}\} | e \in A\}$. Informally, $S(A)$ is the set of expressions that result when we feed each $e \in A$ individually into S.

Note that since e and $\vdash$ are not type variables, $S'(\Gamma) \vdash e : S'(\tau) = S'(\Gamma \vdash e : \tau)$; that is, e type-checks with respect to the substitutions of S'.

We shall extend our function **Unify** to also take a substitution as an argument, to give a "context of substitutions" for unifying the set of equations E. We define $\mathbf{Unify}(E, S) \equiv \mathbf{Unify}(S(E)) \circ S$, where (Note that $\mathbf{Unify}(S(E))$ is a substitution, so this definition makes sense.) The inductive definition of $\mathcal{R}$ is as follows:

$$
\begin{aligned}
\mathcal{R}(n, \Gamma, S) &= \langle \mathsf{int},\ S \rangle \\
\mathcal{R}(\#\mathsf{t}, \Gamma, S) &= \langle \mathsf{bool},\ S \rangle \\
\mathcal{R}(\#\mathsf{f}, \Gamma, S) &= \langle \mathsf{bool},\ S \rangle \\
\mathcal{R}(x, \Gamma, S) &= \langle \Gamma(x),\ S \rangle \\
\mathcal{R}(e_1\ e_2, \Gamma, S) &= \mathsf{let}\ \langle T_1,\ S_1 \rangle = \mathcal{R}(e_1, \Gamma, S)\ \mathsf{in}\ \mathsf{let}\ \langle T_2,\ S_2 \rangle = \mathcal{R}(e_2, \Gamma, S_1)\ \mathsf{in}\ \langle T^*,\ \mathbf{Unify}(\{T_2 \to T^* = T_1\}, S_2) \rangle \\
\mathcal{R}(\mathsf{fn}\ x.e, \Gamma, S) &= \mathsf{let}\ \langle T_1,\ S_1 \rangle = \mathcal{R}(e, \Gamma[x \mapsto T^*], S)\ \mathsf{in}\ \langle T^* \to T_1,\ S_1 \rangle
\end{aligned}
$$

where T^* is not used in e, Γ, S. (Note this is a stronger condition than "T^* is a free variable in e, Γ, S.")

Below, we give an example of $\mathcal{R}$ in action:
$\mathcal{R}((\mathsf{fn}\ x.x)\ 1, \emptyset, \emptyset) =$
$\mathsf{let}\ \langle T_1,\ S_1 \rangle = \mathcal{R}((\mathsf{fn}\ x.x), \emptyset, \emptyset)\ \mathsf{in}\ \mathsf{let}\ \langle T_2,\ S_2 \rangle = \mathcal{R}(1, \emptyset, S_1)\ \mathsf{in}\ \langle T^*,\ \mathbf{Unify}(\{T_2 \to T^* = T_1\}, S_2) \rangle$

Noting that $\mathcal{R}((\mathsf{fn}\ x.x), \emptyset, \emptyset) = \mathsf{let}\ \langle T_1,\ S_1 \rangle = \mathcal{R}(x, [x \mapsto T^*], \emptyset)\ \mathsf{in}\ \langle T^{**} \to T_1,\ S_1 \rangle) = \langle T^{**} \to T^{**},\ \emptyset \rangle$, the above is

$= \mathsf{let}\ \langle T_2,\ S_2 \rangle = \mathcal{R}(1, \emptyset, \emptyset)\ \mathsf{in}\ \langle T^*,\ \mathbf{Unify}(\{T_2 \to T^* = T^{**} \to T^{**}\}, \emptyset) \rangle$

Noting that $\mathcal{R}(1, \emptyset, \emptyset) = \langle \mathsf{int},\ \emptyset \rangle$, this is equal to

$= \langle T^*,\ \mathbf{Unify}(\{\mathsf{int} \to T^* = T^{**} \to T^{**}\}, \emptyset) \rangle$
$= \langle T^*,\ \mathbf{Unify}(\{\mathsf{int} = T^{**}, T^* = T^{**}\}, \emptyset) \rangle$
$= \langle T^*,\ \mathbf{Unify}(\{T^* = \mathsf{int}\}, [T^{**} \mapsto \mathsf{int}]) \rangle$
$= \langle T^*,\ [T^* \mapsto \mathsf{int}, T^{**} \mapsto \mathsf{int}] \rangle$.
Thus, the type of the expression $(\mathsf{fn}\ x.x)\ 1$ is T^* with respect to the substitution $[T^* \mapsto \mathsf{int}, T^{**} \mapsto \mathsf{int}]$, which is simply int.

4 Polymorphism

Note that in the last section we got:

$$\mathcal{R}((\mathsf{fn}\ x.x), \emptyset, \emptyset) = \langle T^{**} \to T^{**},\ \emptyset \rangle$$

Here (fn x.x) has a type where the type identifiers have not been resolved or in other words we have a *type schema*. In general the type reconstruction algorithm may not resolve the types fully and we can take

advantage of this feature of the algorithm to get *polymorphism*. Polymorphism which means *many forms* allows us to use the same expression at various points in a program where the expression can have different types at different points. An example of this is id = (fn x.x) which has type $T^{**} \rightarrow T^{**}$, a type schema, which can be instantiated by setting T^{**} to say *int* and using id where a function of type $int \rightarrow int$ is required. Polymorphism is clearly a convenience for the programmer as it avoids the need to create functions for each of the schema instances. The somewhat suprising aspect of Polymorphism is the fact that one can do type inference with polymorphic types as well.

In order to introduce polymorphism formally into our language (and not just as a by-product of the type reconstruction algorithm) we allow variables to have types that are type schemas and we ensure that at every use of the variable we select an instance of the type schema. We then also allow *let* to bind variables to polymorphic terms. In addition we modify our typing rules and the definition of a type context in the following fashion:

$\Gamma \in Var \rightarrow \sigma$
$\sigma ::= \tau \mid \forall T_1, \ldots, T_n.\tau$, where FTV($\tau$) $\subseteq \{T_1, \ldots, T_n\}$
$\Delta = \{T_1, \ldots, T_n\}$ (= set of legal type variables)
$\Delta \vdash \tau$ (= τ is a legal type in δ i.e. it is well formed.)
$\Delta; \Gamma \vdash e : \tau$ (= using type variables in Δ and in the type context Γ, e has type τ)

Our new typing rules then are:

$$\frac{}{\Delta; \Gamma, x:\tau \vdash x:\tau} \qquad \frac{\Delta \vdash \tau^{i \in 1 \ldots n}}{\Delta; \Gamma, x:(\forall T_1, \ldots, T_n.\tau) \vdash x:\tau\{\tau_i / T_i^{\,i \in 1 \ldots n}\}}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Delta; \Gamma \vdash e_2 : \tau \quad \Delta \vdash \tau, \tau'}{\Delta; \Gamma \vdash (e_1\ e_2) : \tau'} \qquad \frac{\Delta; \Gamma, x:\tau \vdash e:\tau' \quad \Delta \vdash \tau, \tau'}{\Delta; \Gamma \vdash (fn\ x.e) : \tau \rightarrow \tau'}$$

$$\frac{\Delta \cup \{T_1, \ldots, T_n\}; \Gamma \vdash e_1 : \tau \quad \Delta; \Gamma, x:\forall T_1, \ldots, T_n.\tau \vdash e_2 : \tau' \quad \Delta \cup \{T_1, \ldots, T_n\} \vdash \tau \quad \Delta \vdash \tau'}{\Delta; \Gamma \vdash (let\ x = e_1\ in\ e_2) : \tau'}$$

Here is an example of the typing rules in action:

$$\frac{\{T_1\}; \emptyset \vdash (fn\ x.x) : T_1 \rightarrow T_1 \qquad \dfrac{\emptyset; id:\forall T_1.T_1 \rightarrow T_1 \vdash id : int \rightarrow int}{\emptyset; id:\forall T_1.T_1 \rightarrow T_1 \vdash id\ 2 : int}}{\emptyset; \emptyset \vdash (let\ id\ =\ (fn\ x.x)\ in\ id\ 2 : int}$$

The type inference and reconstruction algorithm for our new language is the same as before for all cases other than for *let* and for variables. The algorithm given here is due to Milner and it is built into ML. The basic idea here is that if a variable has a type which is a type schema then the schema needs to be instantiated when the variable is used. Also while reconstructing the type of an expression that is bound to a variable in a *let* expression if a type expression with free type variables is obtained then we should map the type of the variable to a type schema with the appropriate free type variables bound in a $\forall$ type. Here are the appropriate rules for the reconstruction algorithm (Note: the rule for let and letrec are similar and so here we give the more general letrec rule):

$$\begin{aligned}\mathcal{W}(x, \Gamma, S) = case\ \Gamma(x)\ of \\ \qquad \tau\ =\ \langle \tau, S \rangle \\ \qquad \mid\ \forall T_1, \ldots T_n.\tau\ = \langle \tau\{T_{fi}/T_i\}, S \rangle\end{aligned}$$

$$\begin{aligned}&\mathcal{W}(letrec\ x = e_1\ in\ e_2, \Gamma, S) = \\ &\quad let\ \Gamma' = \Gamma[x \mapsto T_f]\ in\ let\ \langle T1,\ S1 \rangle\ =\ \mathcal{W}(e_1, \Gamma', S)\ in \\ &\quad let\ S2\ =\ Unify(\{T_f\ = T1\},\ S1)\ in \\ &\quad let\ \Gamma'' = \Gamma[x \mapsto Generic(T1,\ \Gamma,\ S2)]\ in \\ &\quad \mathcal{W}(e_2,\ \Gamma'',\ S2)\end{aligned}$$

$$Generic(\tau,\ \Gamma,\ S) = \forall T_1, \ldots, T_n.S(\tau),\ where\ \{T_1, \ldots, T_n\} = FTV(S(\tau)) - FTV(S(\Gamma))$$