CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Type Inference

(Monomorphic) Type Inference

Our type inference algorithm will produce a type **scheme** for a given expression (in a given environment). Working with type schemes allows us to concentrate on the **structure** of the term being typed (and the corresponding typing derivation), which is the same regardless of the specific types in question.

Recall that the form of a type derivation for a given term is a tree isomorphic to the syntax tree for the term. To infer a type for the term, we start with a derivation tree of the correct form in which

- each λ-binding type scheme annotation is a fresh type variable
- the type scheme of each term is a fresh type variable.

Example $\lambda\texttt{y.y 3}$

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{y}:\alpha_0\} \;\rhd\; \texttt{y}:\alpha_3}\,\text{VAR} \qquad \dfrac{}{\{\texttt{y}:\alpha_0\} \;\rhd\; \texttt{3}:\alpha_4}\,\text{CONST}}{\{\texttt{y}:\alpha_0\} \;\rhd\; \texttt{(y 3)}:\alpha_2}\,\text{APP}}{\emptyset \;\rhd\; (\lambda\texttt{y}:\alpha_0\texttt{.y 3}):\alpha_1}\,\text{ABS}$$

To make this a **valid** derivation, certain relationships must hold between the type variables mentioned in adjacent rules. We can express these in the form of **equations between type schemes**. These equations represent **constraints** on the possible (valid) instantiations of the type variables.

Example continued

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{y}:\alpha_0\} \;\rhd\; \texttt{y}:\alpha_3}\,\text{VAR} \qquad \dfrac{}{\{\texttt{y}:\alpha_0\} \;\rhd\; \texttt{3}:\alpha_4}\,\text{CONST}}{\{\texttt{y}:\alpha_0\} \;\rhd\; \texttt{(y 3)}:\alpha_2}\,\text{APP}}{\emptyset \;\rhd\; (\lambda\texttt{y}:\alpha_0\texttt{.y 3}):\alpha_1}\,\text{ABS}$$

Each use of an axiom or rule produces one constraint equation. In this example, we get the following

ABS rule:	$\alpha_1 = \alpha_0 \to \alpha_2$
APP rule:	$\alpha_3 = \alpha_4 \to \alpha_2$
VAR rule:	$\alpha_3 = \alpha_0$
CONST rule:	$\alpha_4 = \texttt{int}$

To produce a valid type derivation, we must solve all the constraint equations **simultaneously**.

Here's one way to do this. We eliminate α_3 and α_4 using the (CONST) and (VAR) constraints, leaving

$$\alpha_1 = \alpha_0 \to \alpha_2$$
$$\alpha_0 = \texttt{int} \to \alpha_2$$

Substituting the second equation into the first, we get:

$$\alpha_1 = (\texttt{int} \to \alpha_2) \to \alpha_2$$

Example continued again

Instantiating the original typing tree with this solution, we have the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{y}:\texttt{int} \to \alpha_2\} \;\rhd\; \texttt{y}:\texttt{int} \to \alpha_2}\,\text{VAR} \qquad \dfrac{}{\{\texttt{y}:\texttt{int} \to \alpha_2\} \;\rhd\; \texttt{3}:\texttt{int}}\,\text{CONST}}{\{\texttt{y}:\texttt{int} \to \alpha_2\} \;\rhd\; \texttt{(y 3)}:\alpha_2}\,\text{APP}}{\emptyset \;\rhd\; (\lambda\texttt{y}:\texttt{int} \to \alpha_2\texttt{.y 3}):(\texttt{int} \to \alpha_2) \to \alpha_2}\,\text{ABS}$$

It is easy to see that every **instance** of

$$\emptyset \;\rhd\; (\lambda\texttt{y}:\texttt{int} \to \alpha_2\texttt{.y 3}):(\texttt{int} \to \alpha_2) \to \alpha_2$$

i.e., every assertion resulting from instantiation of the remaining type variable (α_2), will also be derivable. Moreover, since this scheme was constrained as minimally as possible, **every** derivable scheme must be an instance of it.

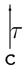
Such a type scheme is called **principal**. The approach outlined here always generates a principal type scheme, (if the given term is typable).

The inference algorithm consists of two steps:

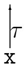
- The term's structure is analyzed to derive a set of constraint equations. This can be described simply in terms of the syntax tree.
- The equations are solved simultaneously. This is an instance of the **unification** problem.

Constraints

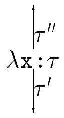
(CONST) $\texttt{E} \rhd \texttt{c}:\tau$ $(\texttt{TypeOf(c)} = \tau)$



(VAR) $\texttt{E} \rhd \texttt{x}:\tau$ $(\texttt{E(x)} = \tau)$

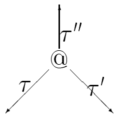


(ABS) $\dfrac{\texttt{E + \{x:}\tau\texttt{\}} \ \rhd\ \texttt{M:}\tau'}{\texttt{E} \ \rhd\ (\lambda\texttt{x:}\tau\texttt{.M}):\tau''}$



$\tau'' = \tau \to \tau'$

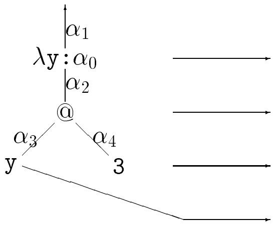
(APP) $\dfrac{\texttt{E} \ \rhd\ \texttt{M:}\tau \qquad \texttt{E} \ \rhd\ \texttt{N:}\tau'}{\texttt{E} \ \rhd\ \texttt{(MN)}:\tau''}$



$\tau = \tau' \to \tau''$

Given a complete expression, we patch the patterns together to get constraint equations.

Example



$\alpha_1 = \alpha_0 \to \alpha_2$

$\alpha_3 = \alpha_4 \to \alpha_2$

$\alpha_4 = \texttt{int}$

$\alpha_3 = \alpha_0$

Constraint Equations

We now have a collection of equations between type schemes

$\tau_{11} = \tau_{12}$
$\tau_{21} = \tau_{22}$
...
$\tau_{n1} = \tau_{n2}$

to be solved simultaneously.

This is a special case of the general problem called **unification**. A solution to a unification problem is a **substitution** from variables to terms that makes all the equations true. Such a substitution is called a **unifier**.

Example: the set of equations

$\alpha_1 = \alpha_0 \to \alpha_2$
$\alpha_3 = \alpha_4 \to \alpha_2$
$\alpha_4 = \texttt{int}$
$\alpha_3 = \alpha_0$

has the unifier

$$\begin{aligned} S = \{\ & \alpha_1 \mapsto (\texttt{int} \to \alpha_2) \to \alpha_2, \\ & \alpha_0 = \alpha_3 \mapsto \texttt{int} \to \alpha_2, \\ & \alpha_4 \mapsto \texttt{int}\} \end{aligned}$$

Moreover, S is a **most general unifier**, meaning that every unifying substitution S' can be obtained from S by a further substitution.

Unification

In general, the unification problem is as follows:

Given a set of equations between terms built from variables and constructors of (fixed) arity ≥ 0, either:

- compute a most general unifier that solves the equations; or
- report that the equations have no solution.

(It can be proved that if the set of equations have any unifier, they have a most general unifier.)

Unification is a widely applicable technique; e.g., it is the fundamental computational process in logic programming. There are several efficient algorithms known for computing unifiers.

For type inference, we specialize the problem as follows:

- the variables are type variables;
- there is one constructor $(\to)$ of arity 2;
- each base type is treated as a constructor of arity 0.

If there is a most general unifier for the equations, it yields a **principal** schematic typing for the original λ-expression. If there is no solution, the original expression is not typable.

A Unification Algorithm

The essence of unification is the recursive traversal of two terms to be equated ("unified").

We denote a substitution by S, and extend its domain from variables to arbitrary terms in the obvious way. Substitutions can be **composed**; we write $S_1 \circ S_2$ to mean the result of applying S_2 and then S_1.

Function $\mathcal{U}$ takes an existing substitution and two terms, and returns an extension of the original substitution that unifies the terms (if possible):

$$\begin{array}{l} \mathcal{U}(S, t_1, t_2) = \\ \quad \text{case } (S(t_1), S(t_2)) \text{ of} \\ \quad\quad (v_1, v_2) : \{v_1 \mapsto v_2\} \circ S \\ \quad\quad (v_1, t_2) : \text{if } v_1 \text{ occurs in } t_2 \text{ then error ``No Unifier''} \\ \quad\quad\quad\quad \text{else } \{v_1 \mapsto t_2\} \circ S \\ \quad\quad (t_1, v_2) : \text{if } v_2 \text{ occurs in } t_1 \text{ then error ``No Unifier''} \\ \quad\quad\quad\quad \text{else } \{v_2 \mapsto t_1\} \circ S \\ \quad\quad (C_1(t_{11}, t_{12}, \ldots, t_{1m}), C_2(t_{21}, t_{22}, \ldots, t_{2n})) : \\ \quad\quad\quad\quad \text{if } C_1 \neq C_2 \\ \quad\quad\quad\quad\quad \text{or } m \neq n \text{ then error ``No Unifier''} \\ \quad\quad\quad\quad \text{else } \mathcal{U}(\ldots\ \mathcal{U}(\ \mathcal{U}(S, t_{11}, t_{21}), t_{12}, t_{22}) \ldots, t_{1m}, t_{2m}) \end{array}$$

To find the m.g.u. of a set of equations

$t_{11} = t_{21}, t_{12} = t_{22}, \ldots, t_{1n} = t_{2n}$

we calculate $\mathcal{U}(\ldots \mathcal{U}(\mathcal{U}(\emptyset, t_{11}, t_{21}), t_{12}, t_{22}) \ldots, t_{1n}, t_{2n})$

Example Again

Use the unification algorithm to solve the set of constraints generated from λ`x.x 3`. To make the role and arity of the constructors clearer, we rewrite the infix $\rightarrow$ constructor as the prefix arity-2 constructor $C_\rightarrow$, and the base type `int` as the arity-0 constructor $C_{\texttt{int}}$.

$\alpha_1 = C_\rightarrow(\alpha_0, \alpha_2)$
$\alpha_3 = C_\rightarrow(\alpha_4, \alpha_2)$
$\alpha_4 = C_{\texttt{int}}$
$\alpha_3 = \alpha_0$

We calculate the m.g.u. by calling $\mathcal{U}$ on each equation in turn, passing the result substitution of each call as the initial substitution for the next:

$S_1 = \mathcal{U}(\emptyset, \alpha_1, C_\rightarrow(\alpha_0, \alpha_2))$
$S_2 = \mathcal{U}(S_1, \alpha_3, C_\rightarrow(\alpha_4, \alpha_2))$
$S_3 = \mathcal{U}(S_2, \alpha_4, C_{\texttt{int}})$
m.g.u. $= S_4 = \mathcal{U}(S_3, \alpha_3, \alpha_0)$

(Notice that the result substitution grows steadily; the algorithm never backtracks. An efficient implementation can therefore maintain a single substitution in a global variable rather than threading it through each call to $\mathcal{U}$.)

Example continued

To pursue the calculation:

$$
\begin{aligned}
S_1 &= \{\ \alpha_1 \mapsto C_\rightarrow(\alpha_0, \alpha_2)\} \\
S_2 &= \{\ \alpha_3 \mapsto C_\rightarrow(\alpha_4, \alpha_2)\} \circ S_1 \\
&= \{\ \alpha_1 \mapsto C_\rightarrow(\alpha_0, \alpha_2), \\
&\qquad \alpha_3 \mapsto C_\rightarrow(\alpha_4, \alpha_2)\} \\
S_3 &= \{\ \alpha_4 \mapsto C_{\texttt{int}}\} \circ S_2 \\
&= \{\ \alpha_1 \mapsto C_\rightarrow(\alpha_0, \alpha_2), \\
&\qquad \alpha_3 \mapsto C_\rightarrow(C_{\texttt{int}}, \alpha_2), \\
&\qquad \alpha_4 \mapsto C_{\texttt{int}}\} \\
S_4 &= \{\ \alpha_0 \mapsto C_\rightarrow(C_{\texttt{int}}, \alpha_2)\} \circ S_3 \\
&= \{\ \alpha_1 \mapsto C_\rightarrow(C_\rightarrow(C_{\texttt{int}}, \alpha_2), \alpha_2), \\
&\qquad \alpha_3 \mapsto C_\rightarrow(C_{\texttt{int}}, \alpha_2), \\
&\qquad \alpha_4 \mapsto C_{\texttt{int}}\} \\
&\qquad \alpha_0 \mapsto C_\rightarrow(C_{\texttt{int}}, \alpha_2)\}
\end{aligned}
$$

This is the same m.g.u. we derived before by ad-hoc substitution.

Note that the compositions performed in the calculation of S_1 and S_2 are trivial, whereas those in S_3 and S_4 are less so.

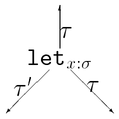
It is crucial to remember that the substitution argument to $\mathcal{U}$ must be applied to both the term arguments **before** performing the case dispatch. This is illustrated in the calculation of the new substitution component of S_4.

Polymorphic Type Inference

The same unification-based approach works for Hindley-Milner style λ-calculus with polymorphic `let`-bindings.

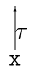
The only changes are:

$$
\texttt{(LET)}\ \frac{\texttt{E} \rhd \texttt{M}:\tau' \qquad \texttt{E + }\{\texttt{x:Clos(}\tau'\texttt{,E)}\} \rhd \texttt{N}:\tau}{\texttt{E} \rhd \texttt{(let x = M in N)}:\tau}
$$



$\sigma = Clos(\tau', E)$

$\texttt{(VAR)}\quad \texttt{E} \rhd \texttt{x}:\tau \quad (\texttt{E(x)} = \sigma < \tau)$



$\tau < \sigma$

To handle the revised `VAR` constraints, we simply introduce new type variables to represent instantiations for the bound types in the scheme, and resolve them by unification.

Before applying the `Clos` rule for `LET`, we must **first** resolve (by unification) all the typing constraints in `M`. Otherwise, we can't determine which variables are free in the environment.