Type inference

Lecture 14 Thursday, March 14, 2019

1 Type inference

In the simply typed lambda calculus, we must explicitly state the type of function arguments: $\lambda x\!:\!\tau.\, e$. This explicitness makes it possible to type check functions.

$$\frac{\Gamma, x\!:\!\tau \vdash e\!:\!\tau'}{\Gamma \vdash \lambda x\!:\!\tau.\, e\!:\!\tau \to \tau'}$$

Suppose we didn't want to provide type annotations for function arguments. Consider the typing rule for functions without type annotations.

$$\frac{\Gamma, x\!:\!\tau \vdash e\!:\!\tau'}{\Gamma \vdash \lambda x.\, e\!:\!\tau \to \tau'}$$

The type-checking algorithm would need to guess or somehow know what type τ to put into the type context.

Can we still type check our program without these type annotations? For the simply typed lambda calculus (and many of the extensions we have considered so far), the answer is yes: we can *infer* or *reconstruct* the types of a program.

Let's consider an example to see how this type inference could work.

$$\lambda a.\, \lambda b.\, \lambda c.\, \mathsf{if}\ a\ (b+1)\ \mathsf{then}\ b\ \mathsf{else}\ c$$

Since the variable b is used in an addition, the type of b must be **int**. The variable a must be some kind of function, since it is applied to the expression $b+1$. Since a has a function type, the type of the expression $b+1$ (i.e., **int**) must be a's argument type. Moreover, the result of the function application ($a\ (b+1)$) is used as the test of a conditional, so it better be the case that the result type of a is also **bool**. So the type of a should be **int** $\to$ **bool**. Both branches of a conditional should return values of the same type, so the type of c must be the same as the type of b, namely **int**.

We can write the expression with the reconstructed types:

$$\lambda a\!:\!\textbf{int} \to \textbf{bool}.\, \lambda b\!:\!\textbf{int}.\, \lambda c\!:\!\textbf{int}.\, \mathsf{if}\ a\ (b+1)\ \mathsf{then}\ b\ \mathsf{else}\ c$$

1.1 Constraint-based typing

We now present an algorithm that, when given a typing context Γ and an expression e, produces a set of *constraints*—equations between types (including type variables)—that must be satisfied in order for e to be well-typed in Γ.

We first introduce *type variables*, which are just placeholders for types. We use X and Y to range over type variables.

The language we will consider is the lambda calculus with integer constants and addition. We assume that all function definitions contain a type annotation for the argument, but this type may simply be a type variable X.

$$\begin{aligned} e &::= x \mid \lambda x\!:\!\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \\ \tau &::= \textbf{int} \mid X \mid \tau_1 \to \tau_2 \end{aligned}$$

To formally define type inference, we introduce a new typing relation:

$$\Gamma \vdash e : \tau \triangleright C$$

Intuitively, if $\Gamma \vdash e : \tau \triangleright C$, then expression e has type τ provided that every constraint in the set C is satisfied. Constraints are of the form $\tau_1 \equiv \tau_2$.

We define the judgment $\Gamma \vdash e : \tau \triangleright C$ with inference rules and axioms. When read from bottom to top, these inference rules provide a procedure that, given Γ and e, calculates τ and C such that $\Gamma \vdash e : \tau \triangleright C$.

$$\text{CT-VAR}\ \frac{}{\Gamma \vdash x : \tau \triangleright \emptyset}\ x : \tau \in \Gamma \qquad\qquad \text{CT-INT}\ \frac{}{\Gamma \vdash n : \textsf{int} \triangleright \emptyset}$$

$$\text{CT-ADD}\ \frac{\Gamma \vdash e_1 : \tau_1 \triangleright C_1 \qquad \Gamma \vdash e_2 : \tau_2 \triangleright C_2}{\Gamma \vdash e_1 + e_2 : \textsf{int} \triangleright C_1 \cup C_2 \cup \{\tau_1 \equiv \textsf{int}, \tau_2 \equiv \textsf{int}\}}$$

$$\text{CT-ABS}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \triangleright C}{\Gamma \vdash \lambda x : \tau_1 .\, e : \tau_1 \to \tau_2 \triangleright C} \qquad \text{CT-APP}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \triangleright C_1 \quad \Gamma \vdash e_2 : \tau_2 \triangleright C_2 \\ C' = C_1 \cup C_2 \cup \{\tau_1 \equiv \tau_2 \to X\}\end{array}}{\Gamma \vdash e_1\ e_2 : X \triangleright C'}\ X \text{ is fresh}$$

Note that we must be careful with the choice of fresh type variables. We have omitted some of the technical details that ensure the fresh type variables in the rule CT-APP are chosen appropriately.

Let's see an example of using this typing judgment to produce a set of constraints. Consider the program $\lambda a : X.\, \lambda b : Y.\, 2 + (a\ (b + 3))$.

$$\dfrac{\dfrac{\dfrac{}{a : X, b : Y \vdash 2 : \textsf{int} \triangleright \emptyset} \quad \dfrac{\dfrac{\dfrac{}{a : X, b : Y \vdash a : X \triangleright \emptyset} \quad \dfrac{\dfrac{}{a : X, b : Y \vdash b : Y \triangleright \emptyset} \quad \dfrac{}{a : X, b : Y \vdash 3 : \textsf{int} \triangleright \emptyset}}{a : X, b : Y \vdash b + 3 : \textsf{int} \triangleright \{Y \equiv \textsf{int}, \textsf{int} \equiv \textsf{int}\}}}{a : X, b : Y \vdash a\ (b + 3) : Z \triangleright \{X \equiv \textsf{int} \to Z, Y \equiv \textsf{int}, \textsf{int} \equiv \textsf{int}\}}}{a : X, b : Y \vdash 2 + (a\ (b + 3)) : \textsf{int} \triangleright \{Z \equiv \textsf{int}, X \equiv \textsf{int} \to Z, Y \equiv \textsf{int}, \textsf{int} \equiv \textsf{int}\}}}{a : X \vdash \lambda b : Y.\, 2 + (a\ (b + 3)) : Y \to \textsf{int} \triangleright \{Z \equiv \textsf{int}, X \equiv \textsf{int} \to Z, Y \equiv \textsf{int}, \textsf{int} \equiv \textsf{int}\}}}{\vdash \lambda a : X.\, \lambda b : Y.\, 2 + (a\ (b + 3)) : X \to Y \to \textsf{int} \triangleright \{Z \equiv \textsf{int}, X \equiv \textsf{int} \to Z, Y \equiv \textsf{int}, \textsf{int} \equiv \textsf{int}\}}$$

The typing derivation means that expression $\lambda a : X.\, \lambda b : Y.\, 2 + (a\ (b + 3)) : X \to (Y \to \textsf{int})$ has type $X \to Y \to \textsf{int}$ provided that we can satisfy the constraints $Z \equiv \textsf{int}$, $X \equiv \textsf{int} \to Z$, $Y \equiv \textsf{int}$, and $\textsf{int} \equiv \textsf{int}$.

1.2 Unification

So what does it mean for a set of constraints to be satisfied? To answer this question, we define *type substitutions* (or just *substitutions*, when it's clear from context).

1.2.1 Type substitution

A type substitution is a finite map from type variables to types. For example, we write $[X \mapsto \textsf{int}, Y \mapsto \textsf{int} \to \textsf{int}]$ for the substitution that maps type variable X to **int**, and type variable Y to $\textsf{int} \to \textsf{int}$.

Note that the same variable could occur in both the domain and range of a substitution. In that case, the intention is that all substitutions are performed simultaneously. For example the substitution $[X \mapsto \textsf{int}, Y \mapsto \textsf{int} \to X]$ maps Y to $\textsf{int} \to \textsf{X}$ (not to $\textsf{int} \to \textsf{int}$).

More formally, we define substitution of type variables as follows.

$$\sigma(X) = \begin{cases} \tau & \text{if } X \mapsto \tau \in \sigma \\ X & \text{if } X \text{ not in the domain of } \sigma \end{cases}$$
$$\sigma(\textbf{int}) = \textbf{int}$$
$$\sigma(\tau \to \tau') = \sigma(\tau) \to \sigma(\tau')$$

Note that we don't need to worry about avoiding variable capture, since there are no constructs in the language that bind type variables. (We'll soon see more sophisticated typing constructs that introduce binders for type variables.)

We can extend substitution to constraints, and sets of constraints in the obvious way:

$$\sigma(\tau_1 \equiv \tau_2) = \sigma(\tau_1) \equiv \sigma(\tau_2)$$
$$\sigma(C) = \{\sigma(c) \mid c \in C\}$$

Given two substitutions σ and σ', we write $\sigma \circ \sigma'$ for the composition of the substitutions: $\sigma \circ \sigma'(\tau) = \sigma(\sigma'(\tau))$.

1.2.2 Unification

Constraints are of the form $\tau \equiv \tau'$. We say that a substitution σ *unifies* constraint $\tau \equiv \tau'$ if $\sigma(\tau)$ is the same as $\sigma(\tau')$. We say that substitution σ *satisfies* (or *unifies*) set of constraints C if σ unifies every constraint in C.

For example, the substitution $\sigma = [X \mapsto \textbf{int}, Y \mapsto \textbf{int} \to \textbf{int}]$ unifies the constraint $X \to (X \to \textbf{int}) \equiv \textbf{int} \to Y$, since

$$\sigma(X \to (X \to \textbf{int})) \quad = \quad \textbf{int} \to (\textbf{int} \to \textbf{int}) \quad = \quad \sigma(\textbf{int} \to Y)$$

So to solve a set of constraints C, we need to find a substitution that unifies C. More specifically, suppose that $\Gamma \vdash e : \tau \triangleright C$. Expression e is typable if and only if there is a substitution σ that satisfies C, and moreover, the type of e is $\sigma(\tau)$. If there are no substitutions that satisfy C, then we know that e is not typable.

1.2.3 Unification algorithm

To calculate solutions to constraint sets, we use the idea, due to Hindley and Milner, of using *unification* to check that the set of solutions is non-empty, and to find a "best" solution (from which all other solutions can be easily generated).

The algorithm for unification is defined as follows.

$$\begin{aligned}
\mathit{unify}(\emptyset) &= [] \qquad \text{(the empty substitution)} \\
\mathit{unify}(\{\tau \equiv \tau'\} \cup C) &= \text{if } \tau = \tau' \text{ then} \\
&\qquad \mathit{unify}(C) \\
&\quad \text{else if } \tau = X \text{ and } X \text{ not a free variable of } \tau' \text{ then} \\
&\qquad \text{let } \sigma = [X \mapsto \tau'] \text{ in} \\
&\qquad \mathit{unify}(\sigma(C)) \circ \sigma \\
&\quad \text{else if } \tau' = X \text{ and } X \text{ not a free variable of } \tau \text{ then} \\
&\qquad \text{let } \sigma = [X \mapsto \tau] \text{ in} \\
&\qquad \mathit{unify}(\sigma(C)) \circ \sigma \\
&\quad \text{else if } \tau = \tau_o \to \tau_1 \text{ and } \tau' = \tau'_o \to \tau'_1 \text{ then} \\
&\qquad \mathit{unify}(C \cup \{\tau_0 \equiv \tau'_0, \tau_1 \equiv \tau'_1\}) \\
&\quad \text{else} \\
&\qquad \mathit{fail}
\end{aligned}$$

The check that X is not a free variable of the other type ensures that the algorithm doesn't produce a cyclic substitution (e.g., $X \mapsto (X \to X)$), which doesn't make sense with the finite types that we currently have.

The unification algorithm always terminates. (How would you go about proving this?) Moreover, it produces a solution if and only if a solution exists. The solution found is the most general solution, in the sense that if $\sigma = \mathit{unify}(C)$ and σ' is a solution to C, then there is some σ'' such that $\sigma' = \sigma'' \circ \sigma$.