CS 4110 – Programming Languages and Logics
Lecture #25: Type Inference

1 Polymorphism in OCaml

In languages lik OCaml, programmers don't have to annotate their programs with $\forall X.\, \tau$ or $e\,[\tau]$. Both are automatically inferred by the compiler, although the programmer can specify types explicitly if desired.

For example, we can write

```
let double f x = f (f x)
```

and Ocaml will figure out that the type is

```
('a → 'a) → 'a → 'a
```

which is roughly equivalent to

$$\forall A.\, (A \to A) \to A \to A$$

We can also write

```
double (fun x → x+1) 7
```

and Ocaml will infer that the polymorphic function `double` is instantiated on the type `int`.

The polymorphism in ML is not, however, exactly like the polymorphism in System F. ML restricts what types a type variable may be instantiated with. Specifically, type variables can not be instantiated with polymorphic types. Also, polymorphic types are not allowed to appear on the left-hand side of arrows—i.e., a polymorphic type cannot be the type of a function argument. This form of polymorphism is known as *let-polymorphism* (due to the special role played by `let` in ML), or *prenex polymorphism*. These restrictions ensure that *type inference* is decidable.

An example of a term that is typable in System F but not typable in ML is the self-application expression $\lambda x.\, x\, x$. Try typing

```
fun x → x x
```

in the top-level loop of Ocaml, and see what happens...

2 Type Inference

In the simply-typed lambda calculus, we explicitly annotate the type of function arguments: $\lambda x\!:\!\tau.\, e$. These annotations are used in the typing rule for functions.

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

Suppose that we didn't want to provide type annotations for function arguments. We would need to guess a 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 had better be the case that the result type of a is also **bool**. So the type of a should be **int** $\rightarrow$ **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:

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

2.1 Constraint-based typing

We now present an algorithm that, 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 introduce *type variables*, which are just placeholders for types. We let metavariables X and Y 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.

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

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

$$\Gamma \vdash e\!:\!\tau \mid C$$

Intuitively, if $\Gamma \vdash e : \tau \mid C$, then expression e has type τ provided that every constraint in the set C is satisfied.

We define the judgment $\Gamma \vdash e\!:\!\tau \mid C$ 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 $\Gamma \vdash e\!:\!\tau \mid C$.

$$\textsc{CT-Var}\ \frac{x\!:\!\tau \in \Gamma}{\Gamma \vdash x\!:\!\tau \mid \emptyset} \qquad\qquad \textsc{CT-Int}\ \frac{}{\Gamma \vdash n\!:\!\textbf{int} \mid \emptyset}$$

$$\textsc{CT-Add}\ \frac{\Gamma \vdash e_1\!:\!\tau_1 \mid C_1 \quad \Gamma \vdash e_2\!:\!\tau_2 \mid C_2}{\Gamma \vdash e_1 + e_2\!:\!\textbf{int} \mid C_1 \cup C_2 \cup \{\tau_1 = \textbf{int}, \tau_2 = \textbf{int}\}}$$

$$\textsc{CT-Abs}\ \frac{\Gamma, x\!:\!\tau_1 \vdash e\!:\!\tau_2 \mid C}{\Gamma \vdash \lambda x\!:\!\tau_1.\, e\!:\!\tau_1 \rightarrow \tau_2 \mid C}$$

$$\text{CT-APP}\ \frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad C' = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to X\}}{\Gamma \vdash e_1\ e_2 : X \mid C'}\ X \text{ fresh}$$

Note that we must be careful with the choice of type variables—in particular, the type variable in the rule CT-APP must be chosen appropriately.

2.2 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). A type substitution is a finite map from type variables to types. For example, we write $[X \mapsto \textbf{int}, Y \mapsto \textbf{int} \to \textbf{int}]$ for the substitution that maps type variable X to **int**, and type variable Y to $\textbf{int} \to \textbf{int}$. Note that the same variable may occur in both the domain and range of a substitution. In that case, the intention is that the substitutions are performed simultaneously. For example the substitution $[X \mapsto \textbf{int}, Y \mapsto (\textbf{int} \to X)]$ maps Y to $\textbf{int} \to X$.

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. If we had polymorphic types $\forall X.\ \tau$ from the polymorphic lambda calculus, we would need to be concerned with this.

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

2.2.1 Unification

Constraints are of the form $\tau = \tau'$. We say that a substitution σ *unifies* constraint $\tau = \tau'$ if $\sigma(\tau) = \sigma(\tau')$. 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

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

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 $\Gamma \vdash e : \tau \mid C$; a solution for (Γ, e, τ, C) is a pair $\sigma, \tau')$ such that σ satisfies C and $\sigma(\tau) = \tau'$. If there are no substitutions that satisfy C, then we know that e is not typeable.

2.2.2 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 unification algorithm is defined as follows:

```
$unify(\emptyset) = []$        (the empty substitution)
$unify(\{\tau = \tau'\} \cup C') =$ if $\tau = \tau'$ then
        $unify(C')$
    else if $\tau = X$ and $X$ not a free variable of $\tau'$ then
        $unify(C'\{\tau'/X\}) \circ [X \mapsto \tau']$
    else if $\tau' = X$ and $X$ not a free variable of $\tau$ then
        $unify(C'\{\tau/X\}) \circ [X \mapsto \tau]$
    else if $\tau = \tau_o \to \tau_1$ and $\tau' = \tau_o' \to \tau_1'$ then
        $unify(C' \cup \{\tau_0 = \tau_0', \tau_1 = \tau_1'\})$
    else
        *fail*
```

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 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 $\sigma = unify(C)$ and σ' is a solution to C, then there is some σ'' such that $\sigma' = (\sigma'' \circ \sigma)$.