1 Polymorphism in OCaml

In languages like OCaml, programmers don’t have to annotate their programs with ∀X. τ or e [τ]. 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 the System F type

```
∀A. (A → A) → A → A
```

We can also write

```
double (fun x → x+1) 7
```

and OCaml will infer that the polymorphic function double is instantiated at 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 cannot 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-polyomorphism (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 `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: λx:τ.e. These annotations are used in the typing rule for functions.

```
Γ, x : τ ⊢ e : τ' 

Γ ⊢ λx:τ.e : τ → τ'
```

Suppose that we didn’t want to provide type annotations for function arguments. We would need to guess a $\tau$ 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.

\[
\lambda a. \lambda b. \lambda c. \text{if } a \ (b + 1) \ \text{then } b \ \text{else } c
\]

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:

\[
\lambda a : \text{int} \rightarrow \text{bool}. \lambda b : \text{int}. \lambda c : \text{int}. \text{if } a \ (b + 1) \ \text{then } b \ \text{else } c
\]

### 2.1 Constraint-based typing

We now present an algorithm that, given a typing context $\Gamma$ 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 $\Gamma$. 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 $X$.

\[
e ::= x \mid \lambda x : \tau. e \mid e_1 \ e_2 \mid n \mid e_1 + e_2
\]

\[
\tau ::= \text{int} \mid X \mid \tau_1 \rightarrow \tau_2
\]

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

\[
\Gamma \vdash e : \tau | C
\]

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

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

\[
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau | \emptyset} \quad \quad \quad \frac{}{\Gamma \vdash n : \text{int} | \emptyset}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 | C_1 \quad \Gamma \vdash e_2 : \tau_2 | C_2}{\Gamma \vdash e_1 + e_2 : \text{int} | C_1 \cup C_2 \cup \{\tau_1 = \text{int}, \tau_2 = \text{int}\}}
\]

2
CT-Abs: \[
\frac{
\Gamma, x : \tau_1 \vdash e : \tau_2 \mid C
}{
\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \rightarrow \tau_2 \mid C
}\]

CT-App: \[
\frac{
\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad C' = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \rightarrow X\}
}{
\Gamma \vdash e_1 e_2 : X \mid C'
}\]

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 \( \langle X \mapsto \text{int}, Y \mapsto \text{int} \rightarrow \text{int} \rangle \) for the substitution that maps type variable \( X \) to \( \text{int} \), and type variable \( Y \) to \( \text{int} \rightarrow \text{int} \). 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 \( \langle X \mapsto \text{int}, Y \mapsto (\text{int} \rightarrow \text{int}) \rangle \) maps \( Y \) to \( \text{int} \rightarrow X \).

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

\[
\sigma(X) = \begin{cases} 
\tau & \text{if } X \mapsto \tau \in \sigma \\
X & \text{if } X \text{ not in the domain of } \sigma
\end{cases}
\]

\[
\sigma(\text{int}) = \text{int}
\]

\[
\sigma(\tau \rightarrow \tau') = \sigma(\tau) \rightarrow \sigma(\tau')
\]

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 \( \sigma \) and \( \sigma' \), 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 \( \sigma \) unifies constraint \( \tau = \tau' \) if \( \sigma(\tau) = \sigma(\tau') \). We say that substitution \( \sigma \) satisfies (or unifies) set of constraints \( C \) if \( \sigma \) unifies every constraint in \( C \).

For example, the substitution \( \sigma = [X \mapsto \text{int}, Y \mapsto (\text{int} \rightarrow \text{int})] \) unifies the constraint

\[
X \rightarrow (X \rightarrow \text{int}) = \text{int} \rightarrow Y
\]

since

\[
\sigma(X \rightarrow (X \rightarrow \text{int})) = \text{int} \rightarrow (\text{int} \rightarrow \text{int}) = \sigma(\text{int} \rightarrow Y)
\]

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 \( (\Gamma, e, \tau, C) \) is a pair \( \sigma, \tau' \) such that \( \sigma \) 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:

\[
\text{unify} (\emptyset) = [] \quad \text{(the empty substitution)}
\]

\[
\text{unify} (\{ \tau = \tau' \} \cup C') = \begin{cases} 
\text{unify}(C') & \text{if } \tau = \tau' \\
\text{unify}(C' \{ \tau' / X \}) \circ [X \mapsto \tau'] & \text{if } \tau = X \text{ and } X \text{ not a free variable of } \tau' \\
\text{unify}(C' \{ \tau / X \}) \circ [X \mapsto \tau] & \text{if } \tau' = X \text{ and } X \text{ not a free variable of } \tau \\
\text{unify}(C' \cup \{ \tau_0 = \tau'_0, \tau_1 = \tau'_1 \}) & \text{if } \tau = \tau_0 \rightarrow \tau_1 \text{ and } \tau' = \tau'_0 \rightarrow \tau'_1 \\
\text{fail} & \text{else}
\end{cases}
\]

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., \( X \mapsto (X \rightarrow X) \)), 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 \( \sigma = \text{unify}(C) \) and \( \sigma' \) is a solution to \( C \), then there is some \( \sigma'' \) such that \( \sigma' = (\sigma'' \circ \sigma) \).