CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Polytypes

Polymorphism

Polymorphism means using the same code to operate on values of different types at different points in the same program. We are mainly interested in so-called **parametric polymorphism**, which refers to functions that are "oblivious" to the types of their arguments. An example is the function

`pair` $\equiv$ λ`x.(x,x)`

which forms pairs of any type. Obviously, we need some conventions for uniform data representation in order to actually compile such functions.

We distinguish this from **ad-hoc polymorphism**, a.k.a. **overloading**, where the same source code function must have different runtime behavior depending on its argument's types, e.g.,

`double` $\equiv$ λ`x.x + x`

which has different behavior for `x:real` and `x:int`.

We want a type system for polymorphism that is (efficiently) decidable, sound, and has an (efficient decidable) inference algorithm.

Limitations of schemes

We already are part-way there through our use of type schemes.

When we derive a schematic type for `pair`, e.g.,

$\emptyset \ \triangleright\ \lambda\texttt{x.(x,x)}\ :\ \ \alpha \rightarrow (\alpha \times \alpha)$

this implies that the application of `pair` to an expression of **any** type τ will type-check, with α being unified to τ.

Unfortunately, we have no way to express the idea that `pair` may be used with **different** types in different parts of the same expression, because this would require unifying α with two conflicting types.

For example, the following will not type-check:

$(\lambda\texttt{f.(f 3, f true))}(\lambda\texttt{x.(x,x))}$

since α would need to unify with both `int` and `bool`.

What we need is a way to **quantify** type variables over sub-expressions, so they can be **instantiated** to different types at different points in the expression.

2nd-order Polymorphic λ-calculus

One way to express polymorphic programs of this kind is to use an **explicitly-typed** calculus which allows **abstraction** and **application** of type variables as well as ordinary variables.

Examples

$$
\begin{array}{l}
(\lambda \mathtt{f}{:}\forall\alpha.\alpha \to (\alpha \times \alpha).\mathtt{(f\ [int]\ 3,\ f\ [bool]\ true))} \\
\qquad\qquad\qquad (\Lambda\alpha.\lambda \mathtt{x}{:}\alpha.\mathtt{(x,x))}
\end{array}
$$

$$
\begin{array}{l}
(\lambda \mathtt{a{:}int}.\lambda \mathtt{b{:}bool}.\lambda \mathtt{f}{:}\forall\alpha.\alpha \to \alpha.\mathtt{(f\ [int]\ a,f\ [bool]\ b))} \\
\qquad\qquad\qquad \mathtt{1\ true}\ (\Lambda\alpha.\lambda \mathtt{x}{:}\alpha.\mathtt{x})
\end{array}
$$

In general, expressions are:

$$
\begin{array}{rcll}
M & := & c & \text{Built-in constants} \\
 & | & v & \text{Variable names} \\
 & | & (M_1 M_2) & \text{Function applications} \\
 & | & (\lambda v : \sigma.M) & \text{Function abstractions} \\
 & | & (\Lambda\alpha.M) & \text{Type abstractions} \\
 & | & (M[\sigma]) & \text{Type applications}
\end{array}
$$

where polymorphic type schemes are:

$\sigma := b \mid \alpha \mid \sigma_1 \to \sigma_2 \mid \forall\alpha.\sigma$

ML Polymorphism

It is perfectly possible to give (sound) typing rules for the 2nd-order polymoprhic calculus. The problem is that, for the programmer, writing down all the types becomes very tedious very fast. So what we probably want is a type inference mechanism (at least a partial one).

Unfortunately, this calculus has no principal types, and typability of untyped terms has recently been shown to be undecidable.

So nearly all FL's use a more restrictive style of polymorphism (called **ML-polymorphism**, **`let`-style polymorphism**, **Damas-Milner polymorphism**, etc.) for which there **is** an inference algorithm.

The key idea is that we can do inference successfully if we have a visible **definition** for every polymorphic function.

Syntax

We use the following **untyped** syntax for expressions:

$$
\begin{array}{rcll}
M & := & c & \text{Built-in constants} \\
 & | & v & \text{Variable names} \\
 & | & (M_1 M_2) & \text{Function applications} \\
 & | & (\lambda v.M) & \text{Function abstractions} \\
 & | & (\texttt{let } v \texttt{ = } M_1 \texttt{ in } M_2) & \texttt{let}\text{ abstractions}
\end{array}
$$

Note that `let` is now a basic expression form in its own right. `let x = a in b` still has the same **operational** semantics as `(λx.b)a`, but the two expressions will be **different** for typing purposes. Specifically, `let` is used to define expressions that are to be used **polymorphically**.

We would rewrite our first earlier example as:

```
let f = λx.(x,x) in (f 3, f true)
```

The use of `let` is essential; we **can't** write the body here as an ordinary λ-abstraction

```
λf.(f 3, f true)
```

as this will still not type check.

We cannot express the second example at all, however; this style of polymorphism is strictly less powerful.

Top-level Expressions

Note that expressions entered at the "top-level" in CAML are treated like other `let`-bound expressions; that is, for typing purposes the sequence

```
let a = exp1;;
let b = exp2;;
...
```

is equivalent to

```
let a = exp1 in let val b = exp2 in ...
```

In particular, library functions are treated as if they were `let`-bound in the scope of ordinary programs.

ML Polymorphism Type System

We can give rules similar to the ones we introduced earlier for the simply-typed λ-calculus.

In these rules, we use ordinary type schemes just as before

$$\tau ::= b \mid \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \ldots \qquad (b \in B)$$

and also **polymorphic type schemes** of the following form

$$\sigma ::= \tau \mid \forall \alpha . \sigma$$

Note that in these polymorphic schemes any quantifiers always come **first**. For example,

$$\forall \alpha . (\alpha \to (\alpha \times \alpha))$$

is a legitimate polymorphic scheme;

$$(\forall \alpha . \alpha) \to (\forall \alpha . \alpha)$$

is not.

Instantiation

To get an (ordinary) type scheme from a polymorphic scheme $\sigma \equiv \forall\alpha_1\forall\alpha_2 \ldots \forall\alpha_n.\tau$ we instantiate each of the quantified type variables with a type. The resulting scheme will be of the form $\tau' \equiv S(\tau)$, where S is substitution with domain $\{\alpha_1, \ldots, \alpha_n\}$. In this case we write

$\sigma > \tau'$

Examples

$$\forall\alpha.\forall\beta.\alpha \to \gamma \to \beta > \texttt{int} \to \gamma \to \texttt{bool}$$
$$\text{with } S = \{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{bool}\}$$
$$\forall\alpha.\forall\beta.\alpha \to \gamma \to \beta > \texttt{int} \to \gamma \to (\delta \to \delta)$$
$$\text{with } S = \{\alpha \mapsto \texttt{int}, \beta \mapsto (\delta \to \delta)\}$$
$$\forall\alpha.\alpha \to \beta \to \alpha \not> \texttt{int} \to \texttt{bool} \to \texttt{int}$$

Typing Rules

In these rules, environments E map (expression) variables to **polymorphic** type schemes, and `TypeOf` also returns a polymorphic scheme.

$$\frac{}{E \;\triangleright\; \mathtt{c}{:}\tau \;\; (\forall \mathtt{c},\tau \text{ s.t. } \mathtt{Typeof(c)} > \tau)}\; CONST$$

$$\frac{}{E \;\triangleright\; \mathtt{x}{:}\tau \;\; (\mathtt{if\ E(x)} > \tau)}\; VAR$$

$$\frac{E + \{\mathtt{x}{:}\tau_1\} \;\triangleright\; M : \tau_2}{E \;\triangleright\; (\lambda \mathtt{x}{:}\tau_1.M) : \tau_1 \rightarrow \tau_2}\; ABS$$

$$\frac{E \;\triangleright\; M : \tau_1 \rightarrow \tau_2 \quad E \;\triangleright\; N : \tau_1}{E \;\triangleright\; (MN) : \tau_2}\; APP$$

$$\frac{E \;\triangleright\; M : \tau \quad E + \{x : \mathrm{Clos}(\tau, E)\} \;\triangleright\; N : \tau_1}{E \;\triangleright\; \mathtt{let\ x} = M \ \mathtt{in}\ N : \tau_1}\; LET$$

Here $\mathrm{Clos}(\tau, E)$ denotes the "closure" of the type τ formed by abstracting over all the free type variables in τ that are **not** free in E.

Formally

$\mathrm{Clos}(\tau, E) = \forall\alpha_1\forall\alpha_2 \ldots \forall\alpha_n\tau$

where $\{\alpha_1, \alpha_2, \ldots, \alpha_n\} = FV(\tau) - FV(E)$.

Example Derivation

We show a typing derivation for our previous example

`let f =` λ`x.(x,x) in (f 3, f true)`

To handle this example, we need to add a typing rule to deal with the `pair` type constructor.

$$\frac{E \;\triangleright\; M_1 \;:\; \tau_1 \qquad E \;\triangleright\; M_2 \;:\; \tau_2}{E \;\triangleright\; (M_1, M_2) \;:\; \tau_1 \times \tau_2}\; PAIR$$

The derivation is then as follows

$$\dfrac{\dfrac{\dfrac{\dfrac{}{E_1 \triangleright \mathrm{x} : \alpha}\,VAR \quad \dfrac{}{E_1 \triangleright \mathrm{x} : \alpha}\,VAR}{E_1 \triangleright (\mathrm{x},\mathrm{x}) : (\alpha \times \alpha)}\,PAIR \quad \dfrac{\dfrac{\dfrac{}{E_2 \triangleright \mathrm{f} : \mathrm{int} \to (\mathrm{int} \times \mathrm{int})}\,VAR \quad \dfrac{}{E_2 \triangleright 3 : \mathrm{int}}\,CONST}{E_2 \triangleright \mathrm{f}\ 3 : (\mathrm{int} \times \mathrm{int})}\,APP \quad \dfrac{\dfrac{}{E_2 \triangleright \mathrm{f} : \mathrm{bool} \to (\mathrm{bool} \times \mathrm{bool})}\,VAR \quad \dfrac{}{E_2 \triangleright \mathrm{true} : \mathrm{bool}}\,CONST}{E_2 \triangleright \mathrm{f\ true} : (\mathrm{bool} \times \mathrm{bool})}\,APP}{E_2 \triangleright (\mathrm{f}\ 3, \mathrm{f\ true}) : ((\mathrm{int} \times \mathrm{int}) \times (\mathrm{bool} \times \mathrm{bool}))}\,PAIR}{\emptyset \triangleright \lambda \mathrm{x}.(\mathrm{x},\mathrm{x}) : \alpha \to (\alpha \times \alpha)}\,ABS}{\emptyset \triangleright \mathrm{let\ f} = \lambda \mathrm{x}.(\mathrm{x},\mathrm{x})\ \mathrm{in}\ (\mathrm{f}\ 3, \mathrm{f\ true}) : ((\mathrm{int} \times \mathrm{int}) \times (\mathrm{bool} \times \mathrm{bool}))}\,LET$$

where:

$E_1 = \{\mathtt{x} : \alpha\}$
$E_2 = \{\mathtt{f} : \forall\alpha.\alpha \to (\alpha \times \alpha)\}$

The Closure Rule

In the previous example, the Clos rule generalized **all** the type variables in the type derived for the `let`-defining expression. But in general, the rule acts to avoid abstracting over type variables that are really acting as type constants in the current environment.

Consider this example:

λ`x.let f =` λ`y.x in ((f 3) + 1, not(f 3))`

Obviously, this should not be typable, since no `x` can be a legitimate argument to both `+` and `not`. But without the restrictions on closures, for we could derive its type as

$\alpha \rightarrow (\texttt{int} \times \texttt{bool})$.

The problem would arise because we would type the pair expression in the environment:

$\{\texttt{x} : \alpha, \texttt{f} : \forall\alpha\forall\beta.\beta \rightarrow \alpha\}$

But the α in the type scheme for `x` and the α in the type scheme for `f` must be the **same** α, so the polymorphic scheme for `f` should really be quantified **only** over β.

Type Inference

Notice that type abstraction **always** occurs as part of the LET rule and type application occurs as part of the VAR rule. Because of this, polymorphic schemes appear only in the environment; they are never attached directly to expressions. Sometimes a similar system is given with type abstraction and application split out as separate rules; the systems are equivalent.

The advantage of our approach is that, as in the simply-typed case, there is at most one rule applicable to any expression, which can be chosen based completely on its syntactical form. As before, this gives an immediate **type inference** algorithm for this system: use the syntactical structure to generate a schematic derivation and a set of constraints, and use unification to solve the constraints.

In practice, the derivation and solution of the constraints is normally combined into a one-pass algorithm over the structure. For historical reasons, the inference algorithm for ML-style polymorphism is called **Algorithm W**; see text and homework.

The key point about this algorithm is that a polymorphic schematic type of a `let`-bound variable is always deduced **before** the variable's uses are inspected; this is what makes inference feasible.

Efficiency of Algorithm W

Interestingly, this is an example of an algorithm that is efficient in practice but very inefficient (deterministic exponential time complete) in theory. For example, the following nasty example

```
let pair = λx.λy.λz.zxy
 in let x1 = λy.pair y y
    in let x2 = λy.x1(x1(y))
       in ...
              in let xn = λy.xn−1(xn−1(y))
                 in xn(λz.z)
```

produces a principal type of length $2^{2^{cn}}$ (as a string). For $n = 5$ this produces 173 printed pages of output after several hours.

Recursion

As before, we can't define the Y combinator directly in an ML-polymorphism type system, so we need to add explicit support for recursion. Here's a possible definition for a LETREC rule:

$$\frac{E \;\texttt{+}\; \{f : \tau_2 \to \tau_1, x : \tau_2\} \;\triangleright\; M \;:\; \tau_1 \qquad E \;\texttt{+}\; \{f : \mathrm{Clos}(\tau_2 \to \tau_1, E)\} \;\triangleright\; N \;:\; \tau}{E \;\triangleright\; \texttt{letrec f = }\lambda\texttt{x.}M\texttt{ in }N \;:\; \tau}$$

Example

```
letrec map = λf.λx.if null x
    then nil
    else cons(f (hd x),map f (tail x))
in map negate [1,2,3]; map not [true,false]
```

will typecheck correctly as expected, with `map` being given a polymorphic type scheme

$\forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}$.

Note that this typing rule doesn't allow us to define a recursive function that invokes itself in a polymorphic way:

```
letrec f = λn.λx.if n = 0
    then x
    else f (n-1) (x,x)
```

Unfortunately, if we extend the typing system to permit expressions of this kind to be typable, type inference becomes undecidable!

Side-effects

We have to be very careful in extending these typing rules to programs with side-effects. For example, we might naively add the following polymorphic type schemes for ML-style references to `TypeOf`:

$$\{\ \texttt{ref} : \forall\alpha.\alpha \to \alpha\texttt{ref},\\ := : \forall\alpha.\alpha\ \texttt{ref} \to \alpha \to \texttt{unit},\\ ! : \forall\alpha.\alpha\ \texttt{ref} \to \alpha\}$$

where `ref` is a new type constructor.

Unfortunately, under these rules the following expression is typable, though it surely shouldn't be!

```
let r = ref(λx.x) in (r := λx.x+1; (!r) true)
```

Intuitively, the problem is that we infer the type $(\alpha \to \alpha)$ `ref` for `r`'s defining expression, and then **abstract** over α to form a polymorphic scheme for `r`. This abstraction is bogus, since α really refers to the particular type of the value currently held by the reference.

Similar problems arise with other imperative features, such as exceptions. Various solutions to these problems have been proposed and implemented. The cleanest is to put a restriction on the form of `let` definitions, namely the defining expression must be a λ-abstraction, a constant, or a simple variable. In all these cases, the **cbv evaluation** of the expression is essentially a no-op, which turns out to avoid the typing problems.

Overloading

A common complaint about ML is that it doesn't support overloading very well. This is primarily because overloading doesn't mix well with type inference.

Suppose we want to write a function

λ`x.`λ`y.x+y`

In order to determine a type for this function, the inferencer **must** know the type of the `+` operator, e.g., whether it represents integer or real addition. Thus, `+` can't be overloaded.

(Actually, in SML/NJ, `+` **is** overloaded, and the above program is rejected by the typechecker as ambiguous. Arbitrary user functions can't be overloaded, however, for this reason. CAML has no overloading at all.)

Ideally, we might like the type checker to infer that the function takes two "numbers," i.e., either integers or reals – but not, e.g., two booleans. In other words, the function will be treated as polymorphic in an "ad-hoc", rather than "parametric" way.

Haskell incorporates a sytematic mechanism called **type classes** for combining ad-hoc and parametric polymorphism.