CS578
Programming Language Semantics
Spring'24 Lecture Notes
Type Inference

Type Inference

Type inference is the problem of producing a well-typed term from an **untyped term**. Having a type inference mechanism for a language can be very useful, since it saves the programmer from having to declare the types of all identifiers (though such declarations are still permitted).

Type inference is **not** possible for every language and typing system, but there **is** a type inference algorithm relating the untyped and simply-typed λ-calculi.

Formally, we define a function **erase**, which takes a typed term, removes all the type annotations from the λ-bindings, and returns the resulting untyped term.

An untyped term M is **typable** if there exists a (simply-)typed term M', a type τ and an environment E (with FV(M) $\subseteq$ Dom(E)) such that

$E \vdash M' : \tau$ and $M = \text{erase}(M')$

A (typable) untyped term may be the erasure of more than one typed term. Example: λ`x.x` is the erasure of both λ`x:int.x` and λ`x:bool.x`.

Type Schemes

To make the type inference algorithm precise, it is helpful to extend the language of type expressions to include **type variables**, denoted by $\{\alpha, \beta, \gamma, \ldots\}$.

The resulting grammar for **type schemes** is:

$$\tau ::= b \mid \alpha \mid \tau_1 \rightarrow \tau_2 \qquad (b \in B)$$

They are called schemes because each expression can be thought of as a schematic representation of a whole family of (ordinary) types, produced by **instantiating** the type variables with ordinary types.

Examples

α
$\alpha \rightarrow \beta$
$\alpha \rightarrow$ `(int` $\rightarrow \alpha$`)`
`bool`

An ordinary type is just a type scheme with no type variables. They are also called **ground types** or **monotypes**.

We modify our typing system so that typing rules, environments, λ-binding annotations, assertions, etc., all refer to type **schemes** where they used to refer to ground types. All this requires is a suitable change in the interpretation of the metavariables τ.

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 λ`y.y 3`

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

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

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

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:

$$\dfrac{\dfrac{\dfrac{}{\{\texttt{y:int} \to \alpha_2\} \vdash \texttt{y:int} \to \alpha_2}\,VAR \quad \dfrac{}{\{\texttt{y:int} \to \alpha_2\} \vdash \texttt{3:int}}\,CONST}{\{\texttt{y:int} \to \alpha_2\} \vdash \texttt{(y 3)}:\alpha_2}\,APP}{\emptyset \vdash \texttt{(}\lambda\texttt{y:int} \to \alpha_2\texttt{.y 3)}:(\texttt{int} \to \alpha_2) \to \alpha_2}\,ABS$$

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

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

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).

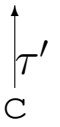
The inference algorithm consists of two steps:

• 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.

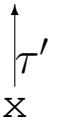
Constraints

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



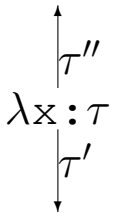
$\tau = \tau'$

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



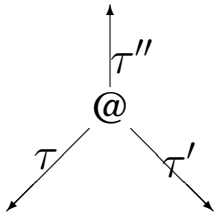
$\tau = \tau'$

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



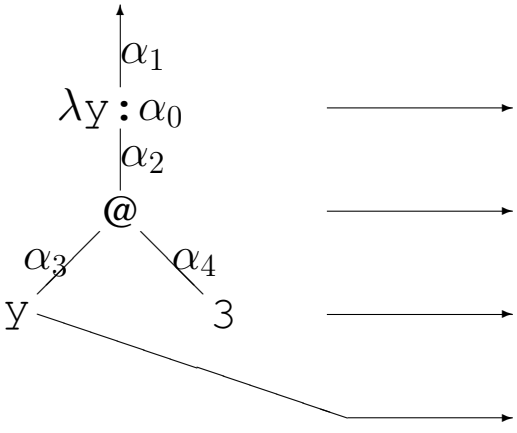
$\tau'' = \tau \to \tau'$

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



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

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



$\alpha_1 = \alpha_0 \to \alpha_2$

$\alpha_3 = \alpha_4 \to \alpha_2$

$\alpha_4 = \texttt{int}$

$\alpha_3 = \alpha_0$

Constraint Equations

We now have a collection of equations between type schemes

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

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

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

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 a fundamental computational process in logic programming languages such as Prolog. 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 ($\rightarrow$) 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):

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

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:

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

(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:

$$
\begin{array}{rl}
S_1 & = \{\ \alpha_1 \mapsto C_{\rightarrow}(\alpha_0, \alpha_2)\} \\
S_2 & = \{\ \alpha_3 \mapsto C_{\rightarrow}(\alpha_4, \alpha_2)\} \circ S_1 \\
& = \{\ \alpha_1 \mapsto C_{\rightarrow}(\alpha_0, \alpha_2), \\
& \quad\ \alpha_3 \mapsto C_{\rightarrow}(\alpha_4, \alpha_2)\} \\
S_3 & = \{\ \alpha_4 \mapsto C_{\mathtt{int}}\} \circ S_2 \\
& = \{\ \alpha_1 \mapsto C_{\rightarrow}(\alpha_0, \alpha_2), \\
& \quad\ \alpha_3 \mapsto C_{\rightarrow}(C_{\mathtt{int}}, \alpha_2), \\
& \quad\ \alpha_4 \mapsto C_{\mathtt{int}}\} \\
S_4 & = \{\ \alpha_0 \mapsto C_{\rightarrow}(C_{\mathtt{int}}, \alpha_2)\} \circ S_3 \\
& = \{\ \alpha_1 \mapsto C_{\rightarrow}(C_{\rightarrow}(C_{\mathtt{int}}, \alpha_2), \alpha_2), \\
& \quad\ \alpha_3 \mapsto C_{\rightarrow}(C_{\mathtt{int}}, \alpha_2), \\
& \quad\ \alpha_4 \mapsto C_{\mathtt{int}}, \\
& \quad\ \alpha_0 \mapsto C_{\rightarrow}(C_{\mathtt{int}}, \alpha_2)\}
\end{array}
$$

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.

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) decideable, sound, and has an (efficient decideable) 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.,

$\emptyset \vdash$ `λx.(x,x)` `:` $\alpha \to (\alpha \times \alpha)$

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:

```
(λf.(f 3, f true))(λx.(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

```
(λf:∀α.α → (α × α).(f [int] 3, f [bool] true))
                   (Λα.λx:α.(x,x))
```

```
(λa:int.λb:bool.λf:∀α.α → α.(f [int] a,f [bool] b))
                    1 true (Λα.λx:α.x)
```

In general, expressions are:

M	$:=$	c	Built-in constants
	$\mid$	v	Variable names
	$\mid$	$(M_1 M_2)$	Function applications
	$\mid$	$(\lambda v : \sigma . M)$	Function abstractions
	$\mid$	$(\Lambda \alpha . M)$	Type abstractions
	$\mid$	$(M[\sigma])$	Type applications

where polymorphic type schemes are:

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

ML Polymorphism

It is perfectly possible to give (sound) typing rules for the 2nd-order polymorphic calculus. We'll investigate this in the next chapter. But meanwhile we want a type inference mechanism, so that the user doesn't have to write down these complex types.

Unfortunately, this calculus has no principal types, and typability of untyped terms is actually undecidable.

So nearly all FL's use a more restrictive style of polymorphism (called **ML-polymorphism**, `let`**-style polymorphism**, **Hindley-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:

M	$:=$	c	Built-in constants
	$\mid$	v	Variable names
	$\mid$	$(M_1 M_2)$	Function applications
	$\mid$	$(\lambda v.M)$	Function abstractions
	$\mid$	$(\texttt{let } v \texttt{ = } M_1 \texttt{ in } M_2)$	`let` abstractions

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 `(`λ`x.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 `let`-bound expressions; that is, for typing purposes the sequence

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

is equivalent to

```
let a = exp1 in let 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

$\forall\alpha.\forall\beta.\alpha \to \gamma \to \beta > \texttt{int} \to \gamma \to \texttt{bool}$
with $S = \{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{bool}\}$
$\forall\alpha.\forall\beta.\alpha \to \gamma \to \beta > \texttt{int} \to \gamma \to (\delta \to \delta)$
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.

$$\frac{\texttt{Typeof(c)} > \tau}{E \vdash \texttt{c:}\tau}\, CONST$$

$$\frac{E(x) > \tau}{E \vdash \texttt{x:}\tau}\, VAR$$

$$\frac{E + \{\texttt{x} : \tau_1\} \vdash M : \tau_2}{E \vdash (\lambda \texttt{x} : \tau_1 . M) : \tau_1 \rightarrow \tau_2}\, ABS$$

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

$$\frac{E \vdash M : \tau \quad E + \{\texttt{x} : \text{Clos}(\tau, E)\} \vdash N : \tau_1}{E \vdash \texttt{let x = } M \texttt{ in } N : \tau_1}\, LET$$

Here $\text{Clos}(\tau, E)$ 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

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

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 = λx.(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.

$$\frac{E \vdash M_1 : \tau_1 \quad E \vdash M_2 : \tau_2}{E \vdash (M_1, M_2) : \tau_1 \times \tau_2}\, PAIR$$

The derivation is then as follows:

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

where:

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

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, 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.

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:

$$\frac{E + \{\texttt{f} : \tau_2 \to \tau_1, \texttt{x} : \tau_2\} \vdash M : \tau_1 \quad E + \{\texttt{f} : \text{Clos}(\tau_2 \to \tau_1, E)\} \vdash N : \tau}{E \vdash \texttt{letrec f = } \lambda\texttt{x.}M \texttt{ in } N : \tau}$$

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

$\forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list`.

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

```
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`:

$$\begin{array}{l}\{\ \texttt{ref} : \forall\alpha.\alpha \to \alpha\ \texttt{ref},\\ \quad := : \forall\alpha.\alpha\ \ \texttt{ref} \to \alpha \to \texttt{unit},\\ \quad ! : \forall\alpha.\alpha\ \ \texttt{ref} \to \alpha\}\end{array}$$

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 that 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.