Logical reasons for types

Types help us rule out certain programs that don't seem to make sense.

Can we apply a function to itself `f f`?

Sometimes, it might be sensible:
`fn x => x` or `fn x => y`.

But in general it looks very suspicious.

This sort of self-application can lead to inconsistencies in formal logics.

Russell's paradox considers $\{x \mid x \notin x\}$,

To avoid this, Russell introduced a system of types.

Type theory is an *alternative* to set theory as a foundation for mathematics.

There are interesting links between type theory and programming.

Programming reasons for types

- More efficient code, and more effective use of space.
- ‘Sanity check’ for programs, catching a lot of programming errors before execution.
- Documentation.
- Data hiding and modularization.

At the same time, some programmers find them an irksome restriction. How can we achieve the best balance?

Different typing methods

We can distinguish between

- Strong typing, as in Modula-3, where types must match up exactly.
- Weak typing, as in C, where greater latitude is allowed.

and also between

- Static typing, as in FORTRAN, which happens at compilation
- Dynamic typing, as in LISP, which happens during execution.

ML is statically and strongly typed.

At the same time, *polymorphism* gives great flexibility.

Type Expressions

A type expression is:

- One of the basic types: `unit`, `bool`, `int`, `real`, `string`.
- A type variable α.
- $\sigma \rightarrow \tau$
- $\sigma * \tau$
- σ `list`

Where σ and τ are any type expressions.

Polymorphism

Some functions can have various different types — *polymorphism*. We distinguish between:

- True ('parametric') polymorphism, where all the possible types for an expression are instances of some schematic type.

  `fn x => x`

- Ad hoc polymorphism, or *overloading*, where this isn't so.

  $+$

Overloading is limited to a few special cases.

Except for overloading, the ML type inference system can infer a type for every expression.

Type variables

If an expression has a type involving a variable α then it can also be given any type that results from consistently replacing α by another type expression.

A type σ is more general than τ, $(\sigma \leq \tau)$, when we can substitute types for type variables in σ and get τ. For example:

$$
\begin{array}{rcl}
\alpha & \leq & \texttt{bool} \\
\beta & \leq & \alpha \\
(\alpha \to \alpha) & \leq & (\texttt{int} \to \texttt{int}) \\
(\alpha \to \alpha) & \not\leq & (\texttt{int} \to \texttt{bool}) \\
(\alpha \to \beta) & \leq & (\beta \to \beta) \\
(\alpha \to \beta) & \not\leq & \alpha
\end{array}
$$

Most general types

Every expression in ML that has a type has a most general type.

(Hindley/Milner)

There is an algorithm for finding the most general type of any expression, even if it contains no type information at all.

ML implementations use this algorithm.

Thus, except for overloading, it is never necessary to write down a type.

This makes the ML type system much less onerous than some others.

For every pair of type expressions σ and τ, if they can be unified, there is a *most general unifier*, up to re-naming of type variables.

ML type inference (1)

Example:

```
fn a => (fn f => fn x => f x) (fn x => x)
```

Attach distinct type variables to distinct variables in the expression.

fn (a:α) => (fn (f:β) => fn (x:γ) =>
(f:β) (x:γ)) (fn (x:δ) => (x:δ))

Note:

- previously defined constants get their assigned type.
- distinct instances of polymorphic constants get assigned types with distinct type variables.
- different bindings of the same variable are really different variables.

ML type inference (2)

$f\ x$ can only be well-typed if $f : \sigma \to \tau$ and $x : \sigma$ for some σ and τ.

$(f\ x) : \tau$.

`fn (x:`β`) => E:`γ has type $\beta \to \gamma$.

Using these facts, we can find relations among the type variables.

Essentially, we get a series of simultaneous equations, and use them to eliminate some unknowns.

The remaining unknowns, if any, parametrize the final polymorphic type.

If the types can't be matched up, or some type variable has to be equal to some composite type containing itself, then typechecking fails.

It is as a form of term *unification*.

Unification

To solve a list of equations:

$$t_1 = s_1, \ldots, t_n = s_n$$

- if t_1 is a variable, replace all occurrences of it in the rest of the list by s_1;
- otherwise, if s_1 is a variable, replace all occurrences of it in the rest of the list by t_1;
- otherwise, if they are distinct constants, then fail;
- otherwise, if they have distinct head constructors ($\rightarrow, *,$ `list`) then fail;
- otherwise, add to the list the list of equations of their corresponding parts;

Repeat until all equations are processed.

ML type inference (3)

For `(f:`β`) (x:`γ`)` to be well-typed,

For some ϵ
$\beta = \gamma \to \epsilon$.

`(fn f => fn x => f x):`$(\gamma \to \epsilon) \to (\gamma \to \epsilon)$

and this is applied to

`(fn x => x):`$\delta \to \delta$

So $(\gamma \to \epsilon) = (\delta \to \delta)$ and
$\gamma = \delta$ and $\epsilon = \delta$

So, the whole expression has type $\alpha \to (\delta \to \delta)$.

It doesn't matter how we name the type variables now, so we can call it $\alpha \to (\beta \to \beta)$.

Let polymorphism

Let allows us to have local bindings

`let val v = E in E' end`.

This is like `(fn v => E') E`, but typechecking this way leads to a problem,

If `v` is bound to something polymorphic, multiple instances must be allowed to have different types.

```
let val I = fn x => x
 in if I true then I 1 else 0
end;
```

One way to type-check this is by substitution of the bound variable.

There are more efficient ways.

Type preservation

Typechecking ML is static, completed before evaluation.

Evaluation of well-typed expressions cannot give rise to ill-typed expressions.

type preservation.

The main step in evaluation is :

```
(fn x => t[x]) u
  -> t[u]
```

`x` and `u` must have the same types at the outset, so this preserves typeability.

The reverse is *not* true,

```
(fn a => fn b => b) (fn x => x x)
```

Pathologies of typechecking

```
fn a => let fun pair x y = fn z => z x y
            val x1 = fn y => pair y y
            val x2 = fn y => x1(x1 y)
            val x3 = fn y => x2(x2 y)
            val x4 = fn y => x3(x3 y)
            val x5 = fn y => x4(x4 y)
         in x5 (fn z => z)
        end;
```

The type of this expression takes about a minute to calculate, and when printed out takes 50,000 lines.