Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 3

ML's type system

Topics covered:

- Why types?
- Approaches to typing
- Basic types.
- Polymorphism.
- ML typechecking.
- Equality types

Logical reasons for types

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

Is it reasonable to apply a function to itself, as in `f f`? It makes some sense for functions like the identity `fn x => x` or constant functions `fn x => y`. But in general it looks very suspicious.

This sort of self-application can lead to inconsistencies in formal logics designed to provide a foundation for mathematics.

For example, Russell's paradox considers $\{x \mid x \not\in x\}$, the set of all sets that are not members of themselves. To avoid this, Russell introduced a system of types.

Type theory is now seen as an *alternative* to set theory as a foundation. There are interesting links between type theory and programming.

Programming reasons for types

Types were introduced in programming for a mixture of reasons. We can (at least in retrospect) see the following advantages:

- They can help the computer to generate more efficient code, and use space more effectively.
- They serve as a kind of 'sanity check' for programs, catching a lot of programming errors before execution.
- They can serve as documentation for people.
- They can help with 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 (e.g. an argument of type `int` to a function expecting a `float`).

and also between

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

ML is statically and strongly typed. At the same time, a feature called *polymorphism* gives many benefits of weak or dynamic typing.

Basic types

The primitive types in ML include:

- Type `unit`, a 1-element type, whose only element is written `()`.
- Type `bool`, a 2-element type whose elements are written `true` and `false`.
- Type `int`, a subset of the positive and negative integers, e.g. `6` and `~11`.
- Type `real`, floating point numbers, written e.g. `1.0` and `3.1415926`.
- Type `string`, which corresponds to sequences of characters, written `"like this"`.

These can be put together using type constructors, including the function constructor `->` and the Cartesian product constructor `*`.

We will see how to define new types and type constructors later.

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, and all instances of that schema are possible types. For example, `fn x => x` can have any type of the form α`->`α, e.g. `int -> int` or `bool -> bool` but not `int -> real`.

- Ad hoc polymorphism, or *overloading*, where this isn't so. The addition operation in `fn x => 1 + x` and `fn x => 1.0 + x` has types `int * int -> int` and `real * real -> real` respectively, but it can't have type `bool * bool -> bool`.

ML has overloading, but only for a few special cases, and we prefer to ignore it. We'll concentrate on (parametric) polymorphism.

Type variables

In order to express polymorphism, ML allows types to contain *type variables*. These are written `'a`, `'b` etc., ASCII approximations to α and β.

If an expression has a type involving α then it can also be given any type that results from consistently replacing α by another type (which may itself involve type variables).

Let's say that a type σ is more general than τ, and write $\sigma \leq \tau$, 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. This was first proved in a similar context by Hindley, and for the exact setup here by Milner.

What's more, 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. Therefore it is never necessary in ML to write down a type. All typing is implicit.

Thus, the ML type system is much less irksome than in many languages like Modula-3. We never have to specify types explicitly *and* we can often re-use the same code with different types: the compiler will work everything out for us.

ML type inference (1)

Roughly speaking, here's how ML's type inference works. Using this method you can typecheck ML expressions yourself, though it's rather laborious and with some experience it becomes much easier. We'll use as an example:

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

First, attach distinct type variables to distinct variables in the expression, and the appropriate types to previously defined constants, perhaps themselves polymorphic. Different type variables must be used for distinct instances of polymorphic constants.

Note that variables like `x` in `fn x =>` have a limited scope, and outside that scope instances of `x` are really separate variables. We get `fn (a:`α`) => (fn (f:`β`) => fn (x:`γ`) => (f:`β`) (x:`γ`)) (fn (x:`δ`) => (x:`δ`))`.

ML type inference (2)

Now an application of a function to an argument `f x` can only be well-typed if $f : \sigma \to \tau$ and $x : \sigma$ for some σ and τ. In this case, $(f\ x) : \tau$.

An expression `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.

Another way of looking at it is as a case of *unification*.

ML type inference (3)

First, we have an application `(f:`β`) (x:`γ`)`. For this to be well-typed, we must have, for some ϵ that $\beta = \gamma \to \epsilon$. Now

`(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 we must have $(\gamma \to \epsilon) = (\delta \to \delta)$ and so $\gamma = \delta$ and $\epsilon = \delta$, and 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 $\alpha \to (\beta \to \beta)$.

This is the end result of type checking.

Let polymorphism

Recall that we can have local bindings, e.g. using `let val v = E in E' end`. We need to say how to typecheck this.

We can regard it as synonymous with `(fn v => E') E`, but then following the previous rules fails to give one very important property:

If `v` is bound to something polymorphic, we want to allow it to be instantiated to multiple instances inside, e.g.

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

One can typecheck such expressions simply by substituting its definition for the bound variable. This is not very efficient, but is satisfactory in principle. Of course, one must also check that the bound expression is itself well-typed, in case it isn't used in the body.

Type preservation

In ML, the phases of *type checking* and *evaluation* are separate, the former completed 'statically' before evaluation begins.

For this to work, it's essential that evaluation of well-typed expressions cannot give rise to ill-typed expressions, i.e. that (static) typing and (dynamic) evaluation don't interfere with each other. This property is called *type preservation*.

The main step in evaluation is the transition from `(fn x => t[x]) u` to `t[u]`. It's not hard to see, following our rules, that `x` and `u` must have the same types at the outset, so this preserves typeability.

The reverse is *not* true, e.g. `(fn a => fn b => b) (fn x => x x)` is untypeable even though `fn b => b` is typeable.

Pathologies of typechecking

In all our examples, the system very quickly infers a most general type for each expression, and the type is simple. This usually happens in practice, but there are pathological cases, e.g. the following example due to Mairson:

```
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.

Don't try this at home.

Equality types

Sometimes one sees instead of `'a` a type variable `''a`. These sometimes arise when using the built-in equality operation inside an expression.

Equality is not completely polymorphic: one cannot compare functions, or expressions built up from functions.

While this might seem to go against the functional programming philosophy, extensional equality of functions is not computable. The type system is used to reflect this.

```
- (fn x => x) = (fn x => x);
! Toplevel input:
! (fn x => x) = (fn x => x);
!    ^^^^^^^
! Type clash: match rule of type
!   'a -> 'b
! cannot have equality type ''c
```