Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 4

Types

Topics covered:

- Why types? Answers from logic and programming
- Simply typed lambda calculus
- Polymorphism
- Let polymorphism
- Most general types and Milner's algorithm
- Strong normalization; recursion.

Logical reasons for types

Just what do the ‘terms’ in λ-calculus really *mean*? How can it make sense to apply a function to itself as in $f\ f$?

It makes some sense for functions like the identity $\lambda x.\ x$ or constant functions $\lambda x.\ y$. But in general it looks very suspicious.

Perhaps this is why the attempt to extend the system to include set theory hit Russell’s paradox.

This is why Church developed typed lambda calculus, which is successfully used, even to this day, as a foundation for mathematics.

Type theory is now seen as an *alternative* to set theory as a foundation. There are interesting links between type theory and programming (Martin-Löf and others).

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.

The stock of types

For now, we assume that we have some set *Ptys* of type constants, e.g. `int` and `bool`, and we can build up new types from them using only the function type constructor, e.g.

$$\mathrm{int} \to \mathrm{int}$$

and

$$(\mathrm{int} \to \mathrm{bool}) \to \mathrm{int} \to \mathrm{bool}$$

(We assume that the function arrow is normally right-associative.)

Formally, this is another inductive definition.

In due course we will introduce more type constructors, e.g. the product type $\alpha \times \beta$, as well as type *variables*, involved in polymorphism. We normally use α and β for type variables.

The idea of typed λ-calculus

Here we simply give every λ-term a *type*, and enforce strong typing.

We can only form an application $s\ t$ if the types match up, i.e. s has function type $\sigma \to \tau$ and t has type σ. In that case, the combination $s\ t$ has type τ.

We use the notation $t : \sigma$ to mean 't has type σ'.

We think of types as sets in which the objects denoted by terms live, and read $t : \sigma$ as $t \in \sigma$. This is the usual mathematical notation in the case of function spaces.

However as usual, this interpretation is only a heuristic guide, and we develop simply typed λ-calculus as a formal system.

Church typing

This is a way of making the types part of the term structure. It is a system of *explicit* types.

- Every constant $c : \sigma_c$ is associated with a particular type.
- A variable can have any type; identically-named variables with different types are different terms.
- If $s : \sigma \to \tau$ and $t : \sigma$ are terms, then $s\ t : \tau$ is a term.
- If $v : \sigma$ is a variable and $t : \tau$ is a term then $\lambda x.\, t : \sigma \to \tau$ is a term.

However we prefer to use *implicit* typing. In fact, in ML, the user never has to write a type!

Curry typing

Here the terms are exactly the same as before, and there is a separate notion of typing. All typing is *implicit*, and a term may have many types. We use a *context* Γ which gives type assignments for variables and constants. Now:

$$\Gamma \vdash t : \sigma$$

means that 'assuming all the type assignments in Γ hold, then t has type σ'.

We will continue to write simply $t : \sigma$ in cases where the context is empty.

Formal typing rules

The typing relation is defined inductively as follows:

$$\frac{v : \sigma \in \Gamma}{\Gamma \vdash v : \sigma}$$

$$\frac{c : \sigma \in \Gamma}{\Gamma \vdash c : \sigma}$$

$$\frac{\Gamma \vdash s : \sigma \to \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash s\ t : \tau}$$

$$\frac{\Gamma \cup \{v : \sigma\} \vdash t : \tau}{\Gamma \vdash \lambda v.\ t : \sigma \to \tau}$$

An example

Let us see how to assign a type to the identity function $\lambda x.\ x$. By the first rule we have:

$$\{x : \sigma\} \vdash x : \sigma$$

and therefore by the last rule we get as expected

$$\emptyset \vdash \lambda x.\ x : \sigma \to \sigma$$

By convention, we just write $\lambda x.\ x : \sigma \to \sigma$.

Note that the context was crucial to 'link' different occurrences of the same variable. In Church typing this would not have been needed as each variable carries around its type.

Type preservation

The conversion rules are exactly the same as in the untyped case.

A very important property is *type preservation.* This says that the conversion rules from untyped λ-calculus preserve the typability properties.

For example if $\Gamma \vdash t : \sigma$ and $t \underset{\beta}{\longrightarrow} t'$, then also $\Gamma \vdash t' : \sigma$.

Similar properties hold for the other conversions, and by structural induction we find that if $\Gamma \vdash t : \sigma$ and $t \longrightarrow t'$, then also $\Gamma \vdash t' : \sigma$.

This shows that static typing and dynamic execution don't interfere with each other.

Adding new types

The above system still works if we add more primitive types and type constructors.

For example, let us introduce a new binary type constructor $\times$ for product types.

We also introduce the (infix) pairing operator ',', which has type $\sigma \to \tau \to \sigma \times \tau$. Thus:

$$(x : \sigma), (y : \tau) : \sigma \times \tau$$

We can also introduce constants $\mathrm{fst} : \sigma \times \tau \to \sigma$ and $\mathrm{snd} : \sigma \times \tau \to \tau$.

In general, we can have arbitrarily many type constructors, and arbitrary constants whose types involve these constructors. As we shall see, ML lets us define our own type constructors.

Polymorphism

The Curry typing system already gives us a form of *polymorphism*, in that a given term may have different types. We distinguish between

- True ('parametric') polymorphism, where all the possible types bear a structural relationship.
- Ad hoc polymorphism, or *overloading*, where the different types a term may have are different, e.g. the use of $+$ over different number systems.

We have true, parametric, polymorphism. For example, possible types for the identity function are $\sigma \to \sigma$, $\tau \to \tau$, or $(\sigma \to \tau) \to (\sigma \to \tau)$.

Let polymorphism (1)

Our system has an unfortunate restriction. We can use the same expression with different types in the same term, e.g:

$$\text{if } (\lambda x.\ x) \text{ true then } (\lambda x.\ x)\ 1 \text{ else } 0$$

However if we use a local `let` binding, this is no longer the case. Consider:

$$\text{let } I = \lambda x.\ x \text{ in if } I \text{ true then } I\ 1 \text{ else } 0$$

According to our definitions, this is just syntactic sugar for:

$$(\lambda I.\ \text{if } I \text{ true then } I\ 1 \text{ else } 0)\ (\lambda x.\ x)$$

This cannot be typed according to our rules, since the *single* instance of the identity function needs to be given a *single* type.

Let polymorphism (2)

This restriction is unbearable for practical functional programming. We take two measures to avoid it.

First, we make the `let` construct *primitive*, instead of regarding it as syntactic sugar, as before.

Secondly, we add a new typing rule:

$$\frac{\Gamma \vdash s : \sigma \quad \Gamma \vdash t[s/x] : \tau}{\Gamma \vdash \text{let } x = s \text{ in } t : \tau}$$

Now we can type expressions like the above in the way that we would expect.

Most general types (1)

We have said that the different types for a term bear a structural similarity. In fact more is true: there is always a *most general type* and all other types are instances of this type.

We say that a type σ is more general than σ', and write $\sigma \leq \sigma'$, when we can substitute types for type variables in σ and get σ'.

The formal definition of substitution at the type level is straightforward, because there is no notion of binding. We will use the same notation. For example:

$$\alpha \to bool \leq (\tau \to \tau) \to bool$$

because

$$(\alpha \to bool)[(\tau \to \tau)/\alpha] = (\tau \to \tau) \to bool$$

But not conversely.

Most general types (2)

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.

Strong normalization

Recall our examples of terms with no normal form, such as:

$$
\begin{array}{ll}
 & ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\
\longrightarrow & ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\
\longrightarrow & (\cdots)
\end{array}
$$

In typed lambda calculus, this cannot happen — every typable term has a normal form and all reduction sequences terminate in that normal form. This property is known as *strong normalization.*

This sounds good — every program terminates. Unfortunately this means that we can't write all computable functions, or even all the interesting ones. We need to add something.

Recursion

We can't define arbitrary recursive functions any more — if we could then we could create non-terminating functions.

Clearly $Y = \lambda f.\ (\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x))$ isn't well-typed because it applies x to itself.

So we simply add a way of defining arbitrary recursive functions that *is* well-typed.

Introduce a family of polymorphic recursion operators of the form:

$$Rec : ((\alpha \to \beta) \to (\alpha \to \beta)) \to \alpha \to \beta$$

and the extra reduction rule that for any $F : (\sigma \to \tau) \to (\sigma \to \tau)$ we have:

$$Rec\ F \longrightarrow F\ (Rec\ F)$$