Introduction to Functional Programming

John Harrison
University of Cambridge

Lecture 5
ML

Topics covered:

- From lambda calculus to ML
- Eager evaluation and its consequences
- The ML family
- Starting up ML
- Basic interaction; expressions and declarations
- Examples

From λ-calculus to ML

We have already added some new primitive features to λ-calculus for various reasons, e.g.

- The `let` construct as primitive to make the polymorphic type system more powerful.
- The recursion operator in order to regain Turing completeness in a typed setting.

This process continues until we reach ML. Notably, basic datatypes are not implemented as λ-terms.

For example, $2 + 2 \longrightarrow 4$ happens by machine arithmetic, not by manipulating Church numerals. These extra reduction rules are often called δ-*conversions*.

Problem with normal order

We have said that *normal order reduction* ('call by name') is the best reduction strategy, because if any strategy terminates, that one does. Recall the example where this strategy terminates but others don't:

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

However, there is an efficiency problem:

$$(\lambda x.\ x + x + x)\ (10 + 5)$$

A normal order reduction gives

$$(10 + 5) + (10 + 5) + (10 + 5)$$

Then we need to evaluate three different instances of the same expression! If we evaluated the argument $10 + 5$ first, it would be more efficient.

Lazy versus eager

This reevaluation is unacceptable in practice. There are two solutions, which split the world of functional programming:

- **Lazy evaluation** keeps normal order reduction but also uses clever sharing techniques, storing expressions as a graph. In this way, an expression is never evaluated until needed, and never more than once.
- **Eager evaluation** simply evaluates the arguments first ('applicative order reduction'), and the programmer is expected to avoid looping instances.

In some ways lazy evaluation is more elegant and powerful. However it is not easy to implement efficiently, and does not interact well with imperative features.

ML's evaluation strategy

ML implements eager evaluation. However, as we shall see, there are ways of achieving many of the benefits of laziness. The evaluation rules are:

- Constants evaluate to themselves
- Evaluation stops immediately at λ-abstractions and does not look inside them. In particular, there is no η-conversion.
- When evaluating a combination $s\ t$, then *first* both s and t are evaluated. Then, assuming that the evaluated form of s is a lambda abstraction, a toplevel β-conversion is performed and the process is repeated.

It is important to grasp this evaluation strategy in order to understand ML properly.

Examples

$$
\begin{array}{ll}
 & (\lambda x.\ (\lambda y.\ y + y)\ x)(2 + 2) \\
\longrightarrow & (\lambda x.\ (\lambda y.\ y + y)\ x)4 \\
\longrightarrow & (\lambda y.\ y + y)4 \\
\longrightarrow & 4 + 4 \\
\longrightarrow & 8
\end{array}
$$

$$
\begin{array}{ll}
 & ((\lambda f x.\ f\ x)\ (\lambda y.\ y + y))\ (2 + 2) \\
\longrightarrow & ((\lambda f x.\ f\ x)\ (\lambda y.\ y + y))\ 4 \\
\longrightarrow & (\lambda x.\ (\lambda y.\ y + y)\ x)\ 4 \\
\longrightarrow & (\lambda y.\ y + y)\ 4 \\
\longrightarrow & 4 + 4
\end{array}
$$

The conditional

Because of eager evaluation, we need to make the conditional primitive with its own special reduction strategy. If we tried to implement

$$\text{if } b \text{ then } e_1 \text{ else } e_2$$

as the application of a normal ternary operator:

$$\text{COND } b\ e_1\ e_2$$

then most interesting cases would loop, because b, e_1 and e_2 are *all* evaluated before the value of b is used. Consider:

$$\begin{aligned}\text{let rec fact}(n) = \;& \text{if ISZERO } n \text{ then } 1 \\ & \text{else } n * \text{fact}(\text{PRE } n)\end{aligned}$$

If we want $\text{fact}(0)$, we need to evaluate $\text{fact}(\text{PRE } 0)$, $\text{fact}(\text{PRE } (\text{PRE } 0))$, and so on.

New primitives

Because of eager evaluation, we make two changes.

- We make the conditional statement a primitive of the language, and modify the standard evaluation strategy so that *first* the boolean expression is evaluated, and only then is *exactly one* of the two arms evaluated, as appropriate.

- We also slightly modify the reduction rule for the recursion operator. Instead of:

$$Rec\ f \longrightarrow f(Rec\ f)$$

we use

$$Rec\ f \longrightarrow f(\lambda x.\ Rec\ f\ x)$$

This avoids looping because ML will not evaluate inside the lambda.

The ML family

ML is not a single language. There are many descendants of the original ML used as the metalanguage of Edinburgh LCF:

- Standard ML and SML/NJ
- CAML and Objective CAML
- Edinburgh ML and Cambridge ML
- Lazy ML

We will be using CAML Light, a small and efficient implementation of the CAML language.

The features we use are general, and can easily be translated to other dialects, and other functional languages.

Starting up ML

ML is already installed on Thor. In order to use it, put its binary directory on your path:

```
PATH="$PATH:/home/jrh13/caml/bin"
export PATH
```

You can put these two lines near the end of your `.bash_profile`, and it will be set up automatically every time you log on.

Now simply type `camllight` and the system will start up and present its prompt.

```
$ camllight
>       Caml Light version 0.73

#
```

If you want to install CAML Light on your own computer, see the Web page:

```
http://pauillac.inria.fr/caml/
```

Interacting with ML

The simplest way to interact with ML is to use it as a simple calculator.

```
#10 + 5;;
- : int = 15
```

ML not only prints the result, but also infers the *type* of the expression, namely `int`. If ML cannot assign a type to an expression then it is rejected:

```
#1 + true;;
Toplevel input:
>let it = 1 + true;;
>            ^^^^
This expression has type bool,
but is used with type int.
```

ML knows the type of the built-in operator `+`, and that is how it makes its type inference.

Using functions (1)

Since ML is a functional language, expressions are allowed to have function type. The ML syntax for a lambda abstraction $\lambda x.\ t[x]$ is `fun x -> t[x]`. For example we can define the successor function:

```
#fun x -> x + 1;;
- : int -> int = <fun>
```

Again, the type of the expression, this time `int -> int`, is inferred and displayed. However the function itself is not printed; the system merely writes `<fun>`.

Functions are applied by juxtaposition, as in λ-calculus:

```
#(fun x -> x + 1) 4;;
- : int = 5
```

Using functions (2)

ML supports similar syntactic conventions to λ-calculus for curried functions of several arguments:

```
#(fun x -> fun y -> x + y) 1 2;;
- : int = 3
#(fun x y -> x + y) 1 2;;
- : int = 3
#fun x y -> x + y;;
- : int -> int -> int = <fun>
```

The first two lines are treated completely equivalently, as is the fully bracketed version:

```
#((fun x -> (fun y -> x + y)) 1) 2;;
- : int = 3
```

Let bindings

It is not necessary to evaluate an expresion all in one piece. You can use `let` bindings:

```
#let successor = fun x -> x + 1 in
 successor(successor(successor 0));;
- : int = 3
```

When binding functions, one can also use the more elegant sugaring:

```
#let successor x = x + 1 in
 successor(successor(successor 0));;
- : int = 3
```

Function bindings can also be recursive: just add the `rec` keyword:

```
#let rec fact n = if n = 0 then 1
                  else n * fact(n - 1) in
 fact 6;;
- : int = 720
```

Declarations

In fact, the set of bindings can be augmented interactively. Simply omit the `in` and terminate the phrase:

```
#let successor x = x + 1;;
successor : int -> int = <fun>
```

or

```
#let rec fact n = if n = 0 then 1
                  else n * fact(n - 1);;
fact : int -> int = <fun>
```

Thus one can make a series of function declarations. These can then be used later in the session.

```
#successor 6;;
- : int = 7
```

Note: this is *not* the same thing as variable assignment!

Polymorphic functions

We can define polymorphic functions, like the identity operator:

```
#let I = fun x -> x;;
I : 'a -> 'a = <fun>
```

ML prints type variables as `'a`,`'b` etc. These are supposed to be ASCII representations of α, β and so on. We can now use it with different types:

```
#I true;;
- : bool = true
#I 1;;
- : int = 1
#I I I I 12;;
- : int = 12
```

Each instance of `I` in the last expression has a different type, and intutively corresponds to a different function.

Combinators

In fact, let's define all the combinators:

```
#let I x = x;;
I : 'a -> 'a = <fun>
#let K x y = x;;
K : 'a -> 'b -> 'a = <fun>
#let S f g x = (f x) (g x);;
S : ('a -> 'b -> 'c) ->
       ('a -> 'b) -> 'a -> 'c = <fun>
```

Note that the system keeps track of the types for us. Recall that $I = S\ K\ K$; let us try this out in ML:

```
#let I' x = S K K x;;
I' : 'a -> 'a = <fun>
```

It has the right type, and seems to work:

```
#I' 3 = 3;;
- : bool = true
```

Equality of functions

Why don't we just compare the two versions of the identity operator? Here's why:

```
#I' = I;;
Uncaught exception: Invalid_argument
   "equal: functional value"
```

It is in general *forbidden* to compare functions for equality. Why? Aren't functions supposed to be first-class objects in ML?

Yes, but unfortunately, (extensional) function equality is not computable. This is a consequence of *Rice's theorem*, or simply of the *unsolvability of the halting problem*.

Some versions of ML distinguish between functions and other 'equality types' and filter out these cases during typechecking.

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:

```
let pair x y = fun z -> z x y in
let x1 = fun y -> pair y y in
let x2 = fun y -> x1(x1 y) in
let x3 = fun y -> x2(x2 y) in
let x4 = fun y -> x3(x3 y) in
let x5 = fun y -> x4(x4 y) in
x5(fun z -> z);;
```

The type of this expression takes about 10 seconds to calculate, and occupies over 4000 lines on an 80-column terminal.

Don't try this at home.