Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 6

Details of ML

Topics covered:

- Interaction with ML; loading files
- Basic datatypes and operators
- Infixes and concrete syntax
- Further examples
- Defining new types

Interaction with ML

We have just been typing things into ML and thinking about the results. However one doesn't write real programs in this way.

Typically, one writes the expressions and declarations in a file. To try them out as you go, these can be inserted in the ML window using cut and paste.

You can cut and paste using X-windows and similar systems, or an editor like Emacs with multiple buffers.

For larger programs, it's convenient simply to load them from file into the ML session. This can be done using the `include` command in ML.

Loading from files (1)

Suppose that the file `myprog.ml` contains the following lines:

```
let pythag x y z =
  x * x + y * y = z * z;;

pythag 3 4 5;;

pythag 5 12 13;;

pythag 1 2 3;;
```

Loading from files (2)

Using `include` on this file results in the following:

```
#include "myprog.ml";;
pythag : int -> int -> int -> bool = <fun>
- : bool = true
- : bool = true
- : bool = false
- : unit = ()
```

ML responds just as if the file had been typed in at the top level.

Normally, the inclusion process terminates as soon as an error is encountered. The error message is accompanied by a line number.

The final line is the return from the `include` command itself. It has type `string -> unit`.

It is permissible to nest inclusions.

Comments

Comments are written between `(*` and `*)`, e.g.

```
(* ---------------------------------- *)
(* Is (x,y,z) is a Pythagorean triple? *)
(* ---------------------------------- *)

let pythag x y z =
    x * x + y * y = z * z;;

(*comments*) pythag (*can*) 3 (*go*) 4
(*almost*) 5 (*anywhere*)
(* and (* can (* be (* nested *)
quite *) arbitrarily *) *);;
```

The comments are ignored by ML, but are useful for people reading the code.

Basic datatypes

The main primitive types are:

- 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, written for example `6` and `-11`.
- 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 other types and type constructors later.

Infixes

Many built-in constants like + have infix status. They also have a notion of precedence, so expressions associate more or less as expected.

For example, `x < 2 * y + z` is parsed as `< x (+ (* 2 y) z)`.

You can define your own infixes using the `#infix` directive:

```
#let o f g = fun x -> f(g x);;
o : ('a -> 'b) -> ('c -> 'a) ->
    'c -> 'b = <fun>
##infix "o";;
#let add2 = successor o successor;;
add2 : int -> int = <fun>
#add2 0;;
- : int = 2
```

Basic infix operators

Operator	Meaning
`mod`	Modulus (remainder)
`*`	Multiplication
`/`	Truncating division
`+`	Addition
`-`	Subtraction
`^`	String concatenation
`=`	Equality
`<>`	Inequality
`<`	Less than
`<=`	Less than or equal
`>`	Greater than
`>=`	Greater than or equal
`&`	Boolean 'and'
`or`	Boolean 'or'

Notes on operators

Note that the logical operators `&` and `or` also have a non-eager evaluation strategy. You can regard them as synonyms:

$$
\begin{aligned}
p \;\&\; q &\stackrel{\triangle}{=} \text{if } p \text{ then } q \text{ else false} \\
p \text{ or } q &\stackrel{\triangle}{=} \text{if } p \text{ then true else } q
\end{aligned}
$$

There are also two built-in symbolic operations on the basic types, `-`, which performs numerical negation, and `not` which performs logical negation.

The logical `not` operator has special prefix status, so that `not not p` is parsed as `not (not p)`. This reverses the usual left-associativity convention for functions.

Examples (1)

We define by recursion a function that takes a positive integer n and a function f and returns f^n, i.e. $f \circ \cdots \circ f$ (n times):

```
#let rec funpow n f x =
    if n = 0 then x
    else funpow (n - 1) f (f x);;
funpow : int -> ('a -> 'a) ->
         'a -> 'a = <fun>
```

In fact, `funpow` takes a machine integer `n` to the corresponding Church numeral. We can easily define an inverse function that goes the other way:

```
#let defrock n = n (fun x -> x + 1) 0;;
defrock : ((int -> int) -> int -> 'a) ->
          'a = <fun>
#defrock(funpow 32);;
- : int = 32
```

Examples (2)

Recall the arithmetic operators on Church numerals. We can test them:

```
  #let add m n f x = m f (n f x)
   and mul m n f x = m (n f) x
   and exp m n f x = n m f x;;
   and test bop x y =
     defrock (bop (funpow x) (funpow y));;
...
  #test add 2 10;;
  - : int = 12
  #test mul 2 10;;
  - : int = 20
  #test exp 2 10;;
  - : int = 1024
```

They seem to work. Of course they're not very efficient!

Defining new types

ML has facilities for defining new types and type constructors.

In fact, the constructors can be recursive, i.e. not only take existing types as arguments but the new type itself.

We will look at a simple example first: the disjoint sum type.

```
#type ('a,'b)sum = inl of 'a
                 | inr of 'b;;
Type sum defined.
```

This creates a new type `sum` and two new *constructors*:

```
#inl;;
- : 'a -> ('a, 'b) sum = <fun>
#inr;;
- : 'a -> ('b, 'a) sum = <fun>
```

Properties of type constructors

All type constructors arising in this way have three key properties:

1. They are exhaustive, i.e. every element of the new type is obtainable either by `inl x` for some `x` or `inr y` for some `y`.

2. They are injective, i.e. an equality test `inl x = inl y` is true if and only if `x = y`, and similarly for `inr`.

3. They are distinct, i.e. their ranges are disjoint. More concretely this means in the above example that `inl x = inr y` is false whatever `x` and `y` might be.

Because of these properties, we can define functions by *pattern matching*.

Pattern matching

We perform pattern matching by using more general expressions called *varstructs* as the arguments to lambdas. We can also consider several different cases. For example:

```
#fun (inl n) -> n > 6
   | (inr b) -> b;;
- : (int, bool) sum -> bool = <fun>
```

This function has the property, naturally enough, that when applied to `inl n` it returns `n > 6` and when applied to `inr b` it returns `b`. Why is this valid?

It is permissible to recover `n` from `inl n` because the constructors are injective. The cases cannot clash because the constructors are distinct. Finally, the function is defined everywhere because the constructors are exhaustive.

Non-exhaustive matching

In fact, we can define partial functions that don't cover every case:

```
#fun (inr b) -> b;;
Toplevel input:
>fun (inr b) -> b;;
>^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
- : ('a, 'b) sum -> 'b = <fun>
```

The compiler warns us of this fact. If we try to use the function on an argument not of the form `inl x`, then it will not work:

```
#(...) (inl 3);;
Uncaught exception: Match_failure
```

General matching

We can perform matching even in other situations, when the matches might not be mutually exclusive. In this case, the first possible match is taken.

```
#(fun 0 -> true | n -> false) 0;;
- : bool = true
#(fun 0 -> true | n -> false) 1;;
- : bool = false
```

However, in general, constants need special constructor status, or they will be treated just as variables for binding:

```
#(fun true -> 1 | false -> 0) (4 < 3);;
- : int = 0
#let true_1 = true and false_1 = false in
(fun true_1 -> 1 | false_1 -> 0) (4 < 3);;
- : int = 1
```

Other matching constructs

Instead of using `fun` with various patterns, and applying the function to an expression, there is an alternative syntax that uses the expression directly:

$$\text{match } E \text{ with } pattern_1\texttt{->}E_1 \mid \cdots \mid pattern_n\texttt{->}E_n$$

The simplest alternative of all is to use

$$\text{let } pattern = expression$$

but in this case only a single pattern is allowed. For example:

```
#let (inl x) = inl 3;;
Toplevel input:
>let (inl x) = inl 3;;
>^^^^^^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
x : int = 3
```

Recursive types

As we have said, types can be defined recursively, i.e. in terms of themselves.

```
#type ('a)list = Nil
               | Cons of 'a * ('a)list;;
Type list defined.
#Nil;;
- : 'a list = Nil
#Cons;;
- : 'a * 'a list -> 'a list = <fun>
```

We imagine `Nil` as the empty list and `Cons` as a function that adds a new element on the front of a list. The lists $[]$, $[1]$, $[1;2]$ and $[1;2;3]$ are written:

```
Nil;;
Cons(1,Nil);;
Cons(1,Cons(2,Nil));;
Cons(1,Cons(2,Cons(3,Nil)));;
```

Lists

Actually, this type is already built in. The empty list is written `[]` and the recursive constructor `::`, has infix status. Thus, the above lists are actually written:

```
#[];;
- : 'a list = []
#1::[];;
- : int list = [1]
#1::2::[];;
- : int list = [1; 2]
#1::2::3::[];;
- : int list = [1; 2; 3]
```

The version that is printed can also be used for input:

```
#[1;2;3;4;5] = 1::2::3::4::5::[];;
- : bool = true
```

Pattern matching over lists

We can now define functions by pattern matching in the usual way. For example, we can define functions to take the head and tail of a list:

```
#let hd (h::t) = h;;
Toplevel input:
>let hd (h::t) = h;;
>    ^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
hd : 'a list -> 'a = <fun>
#let tl (h::t) = t;;
Toplevel input:
>let tl (h::t) = t;;
>    ^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
tl : 'a list -> 'a list = <fun>
```

ML warns us that they will fail when applied to an empty list.

Recursive functions over lists

It is possible to mix pattern matching and recursion. This is natural since the type itself is defined recursively. For example, here is a function to return the length of a list:

```
#let rec length =
   fun [] -> 0
     | (h::t) -> 1 + length t;;
length : 'a list -> int = <fun>
#length [];;
- : int = 0
#length [5;3;1];;
- : int = 3
```

Alternatively, this can be written in terms of our earlier 'destructor' functions `hd` and `tl`. This style of function definition is more usual in many languages, notably LISP, but the direct use of pattern matching is often more elegant.

The subtlety of recursive types

Consider the following:

```
#type ('a)embedding =
   K of ('a)embedding->'a;;
```

This looks suspicious because it embeds the function space $A \to B$ inside A. In fact it only embeds the *computable* functions. We must be careful with semantics, and also types:

```
#let Y h =
  let g (K x) z = h (x (K x)) z in
  g (K g);;
Y : (('a -> 'b) -> 'a -> 'b) ->
    'a -> 'b = <fun>
#let fact = Y (fun f n ->
   if n = 0 then 1 else n * f(n - 1));;
fact : int -> int = <fun>
#fact 6;;
- : int = 720
```