Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 8

Imperative features of ML

Topics covered:

- Output
- Sequencing commands
- Exceptions
- References
- Arrays
- Imperative types

ML's imperative features

ML is not a pure functional programming language. Now at last we will discuss its imperative features.

Whether a certain feature is really *imperative* is partly a matter of taste. We group together here some features that may not be found in pure languages.

The main reasons for having these features is that they can make programs (a) more efficient and/or (b) easier to write.

In any case, it's hard to imagine doing certain things like I/O in ML in a purely functional way.

Output

Input-output and other kinds of interaction with the environment seem essentially imperative.

From a certain point of view, we can imagine the input and output as potentially infinite streams and handle them in a purely functional style — this is done in some lazy languages like Miranda or Haskell. It's not very convenient in ML.

ML has various special functions whose evaluation causes a side-effect of interacting with the environment. We will show one, the `print` function which prints a string:

```
- print;
> val it = fn : string -> unit
- print "hello";
hello> val it = () : unit
- print "goodbye\n";
goodbye
> val it = () : unit
```

Sequencing

Now that we have some expressions whose evaluation causes a side-effect, we care even more about evaluation order. Since we know the rules, we can predict for example:

```
- let val x = print "first "
   in print "second\n"
  end;
first second
```

However, ML also provides a sequencing operation `;` as do most imperative languages like Modula-3. In `e1 ; e2`, expression `e1` is evaluated and the result discarded, then `e2` is evaluated and is the value of the whole expression:

```
- (print "first "; print "second\n");
first second
> val it = () : unit
- (1;2);
> val it = 2 : int
```

Exceptions (1)

ML's errors, e.g. matching failures and division by zero, are all signalled by propagating *exceptions*:

```
- 1 div 0;
! Uncaught exception:
! Div
```

In all these cases the compiler complains about an 'uncaught exception'. As the error message suggests, it is possible to 'catch' them.

There is a type `exn` of *exceptions*, which is effectively a recursive type. Unlike with ordinary types, one can add new constructors for the type `exn` at any point in the program via an exception declaration, e.g.

```
- exception Died;
> exn Died = Died : exn
- exception Failed of string;
> exn Failed = fn : string -> exn
```

Exceptions (2)

One can explicitly generate an exception using the `raise` construct.

```
raise <exception>
```

For example, we might invent our own exception to cover the case of taking the head of an empty list:

```
> exn Head_of_empty = Head_of_empty : exn
- fun hd [] = raise Head_of_empty
    | hd (h::t) = h;
> val hd = fn : 'a list -> 'a
- hd(tl [1]);
! Uncaught exception:
! Head_of_empty
```

Exceptions (3)

One can *catch* exceptions using `<expr> handle <patterns>`, where the patterns to match exceptions are just as for ordinary recursive types.

```
- fun headstring sl =
   hd sl
   handle Head_of_empty => ""
          | Failed s =>
                "Failure because "^s;
> val headstring =
     fn : string list -> string
- headstring ["hi","there"];
> val it = "hi" : string
- headstring [];
> val it = "" : string
```

On one view, exceptions are not really imperative features. We can imagine a hidden type of exceptions tacked onto the return type of each function. Anyway, they are often quite useful!

Exceptions (4)

Exceptions can normally be treated just as other ML values. For example, suppose we want to define a function to "trace" other functions:

```
- fun trace name f x =
   (print ("entering "^name^"\n");
    let val y =
      f x handle ex =>
      (print (name^" gave an exception\n");
       raise ex)
     in (print (name^" finished\n"); y)
    end);
> val trace = fn : string ->
                  ('a -> 'b) -> 'a -> 'b
- fun hd' l = trace "hd" hd l;
> val hd' = fn : 'a list -> 'a
- hd' [1,2];
entering hd
hd finished
> val it = 1 : int
```

References (1)

ML does have real assignable variables, and expressions can, as a side-effect, modify the values of these variables.

They are explicitly accessed via *references* (pointers in C parlance) and the pointers themselves behave more like ordinary ML values.

One sets up a new assignable memory cell with the initial contents `x` by writing `ref x`. This expression returns the corresponding reference, i.e. a pointer to it.

One manipulates the contents via the pointer.

This is quite similar to C: here one often simulates 'variable parameters' and passes back composite objects from functions via explicit use of pointers.

To get at the contents of a `ref`, use the dereferencing (indirection) operator `!`. To modify its value, use `:=`.

References (2)

Here is an example of how we can create and play with a reference cell:

```
- val x = ref 1;
> val x = ref 1 : int ref
- !x;
> val it = 1 : int
- x := 2;
> val it = () : unit
- !x;
> val it = 2 : int
- x := !x + !x;
> val it = () : unit
- x;
> val it = ref 4 : int ref
- !x;
> val it = 4 : int
```

In most respects `ref` behaves like a type constructor, so one can pattern-match against it.

References (3)

References are useful in ML for two different reasons.

First, as you might expect, they allow us to modify the state of the program as we go along, in a more conventional style. If we didn't use references, functions would often have to have additional arguments.

Secondly, they can be used to construct data structures that are *shared* or *cyclic*.

Whichever way you use them, it is sometimes useful to have reference variables inside a function for convenience or efficiency, but use them in such a way that the function as a whole is still a true function, i.e. returns the same value on multiple calls.

References (4)

For example, it's ofen convenient to *memoize* or *cache* the result of a previous function call, so that if we get the same argument again, we can return the stored value instead of recalculating it.

We start by defining a function to find an item in a list of pairs:

```
- exception Not_found;
> exn Not_found = Not_found : exn
- fun assoc a [] = raise Not_found
    | assoc a ((x,y)::rest) =
        if x = a then y
        else assoc a rest;
> val assoc =
    fn : ''a -> (''a * 'b) list -> 'b
```

References (5)

Now we declare an internal reference variable `store` to hold the list of `(x,f(x))` pairs for previously encountered calls.

```
- fun cache f =
      let val store = ref []
       in fn x =>
            assoc x (!store)
            handle Not_found =>
              let val y = f(x)
               in (store := (x,y)::(!store);
                   y)
              end
      end;
> val cache = fn : (''a -> 'b) ->
                   (''a -> 'b)
```

First the cached function sees if it's already got the result stored. If so, it returns it. Otherwise, the underlying function is calculated and a new pair put in the store before the result is returned.

Arrays (1)

As well as individual reference cells, one can use arrays. The appropriate functions for handling arrays need to be made available by:

```
- open Array;
```

An array of size `n`, with each element initialized to `x` is created using the following call

```
- array(n,x);
```

One can then read element `m` of an array `a` using:

```
- sub(a,m);
```

and write value `y` to element `m` of `a` using:

```
- update(a,n,y);
```

The elements are numbered from zero. Thus the elements of an array of size n are $0, \ldots, n-1$.

Arrays (2)

Here is a simple example:

```
- val a = array(5,0);
> val a = <array> : int array
- sub(a,1);
> val it = 0 : int
- update(a,1,7);
> val it = () : unit
- sub(a,1);
> val it = 7 : int
```

All reading and writing is constrained by bounds checking, e.g.

```
- sub(a,5);
! Uncaught exception:
! Subscript
```

Imperative features and types

There are unfortunate interactions between references and let polymorphism.

For example, according to the usual rules, the following should be valid, even though it writes something as an integer and reads it as a boolean:

```
val l = ref [];
l := 1;
hd(!l) = true
```

ML places restrictions on the polymorphic type of expressions involving references to avoid these problems. It won't let you declare something like the above:

```
- val l = ref [];
! Toplevel input:
! val l = ref [];
! ^^^^^^^^^^^^^^
! Value polymorphism: Free type variable
                      at top level
```