Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 8

Effective ML

Topics covered:

- Using standard combinators
- Tail recursion and accumulators
- Forcing evaluation
- Minimizing consing
- Exceptions
- References and arrays

Using standard combinators

It often turns out that a few *combinators* are very useful: we can implement practically anything by plugging them together.

This is particularly true because we can have higher order functions.

For example, the `itlist` function:

$$\begin{aligned}&\text{itlist } f\ [x_1;\ x_2;\ \ldots\ ;x_n]\ b\\ &= f\ x_1\ (f\ x_2\ (f\ x_3\ (\cdots(f\ x_n\ b))))\end{aligned}$$

can often be used to avoid explicit recursive definitions over lists. We define it as:

```
#let rec itlist f =
    fun [] b -> b
      | (h::t) b -> f h (itlist f t b);;
itlist : ('a -> 'b -> 'b) ->
          'a list -> 'b -> 'b = <fun>
```

Itlist examples (1)

For example, here is a function to add up all the elements of a list:

```
#let sum l =
    itlist (fun x sum -> x + sum) l 0;;
sum : int list -> int = <fun>
#sum [1;2;3;4;5];;
- : int = 15
#sum [];;
- : int = 0
#sum [1;1;1;1];;
- : int = 4
```

An alternative coding is simply:

```
#let sum l = itlist (prefix +) l 0;;
```

If we want to multiply the elements instead, we change it to:

```
#let prod l = itlist (prefix *) l 1;;
```

Itlist examples (2)

Here is a filtering function:

```
#let filter p l =
   itlist (fun x s -> if p x then x::s
                     else s) l [];;
```

and here are some logical operations over lists:

```
#let forall p l =
   itlist (fun h a -> p(h) & a) l true;;
#let exists p l =
   itlist (fun h a -> p(h) or a) l false;;
```

and some old favourites:

```
#let length l =
   itlist (fun x s -> s + 1) l 0;;
#let append l m =
   itlist (fun h t -> h::t) l m;;
#let map f l =
   itlist (fun x s -> (f x)::s) l [];;
```

Itlist examples (3)

We can implement set operations quite directly using these combinators as building blocks.

```
#let mem x l =
   exists (fun y -> y = x) l;;
#let insert x l =
  if mem x l then l else x::l;;
#let union l1 l2 =
   itlist insert l1 l2;;
#let setify l =
   union l [];;
#let Union l =
   itlist union l [];;
#let intersect l1 l2 =
   filter (fun x -> mem x l2) l1;;
#let subtract l1 l2 =
   filter (fun x -> not mem x l2) l1;;
#let subset l1 l2 =
   forall (fun t -> mem t l2) l1;;
```

Storing local variables

Recall our definition of the factorial:

```
#let rec fact n = if n = 0 then 1
                  else n * fact(n - 1);;
```

A call to `fact 6` causes another call to `fact 5` (and beyond), but the computer needs to save the old value `6` in order to do the final multiplication.

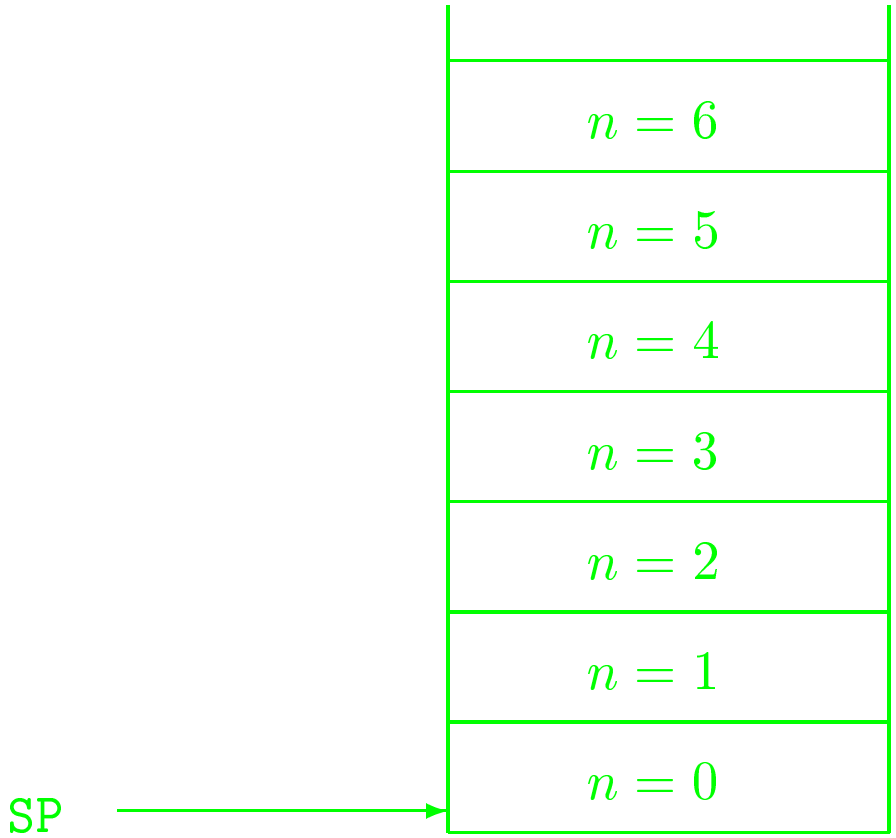
Therefore, the local variables of a function, in this case `n`, cannot be stored in a fixed place, because each instance of the function needs its own copy.

Instead each instance of the function is allocated a frame on a *stack*.

This technique is similar in ML to most other languages that support recursion, including C.

The stack

Here is an imaginary snapshot of the stack during the evaluation of the innermost call of `fact`:



Note that we use about n words of storage when there are n nested recursive calls. In many situations this is very wasteful.

A tail recursive version

Now, by contrast, consider the following implementation of the factorial function:

```
#let rec tfact x n =
   if n = 0 then x
   else tfact (x * n) (n - 1);;
tfact : int -> int -> int = <fun>
#let fact n = tfact 1 n;;
fact : int -> int = <fun>
#fact 6;;
- : int = 720
```

The recursive call is the whole expression; it does not occur as a proper subexpression of some other expression involving values of variables.

Such a call is said to be a *tail call* because it is the very last thing the calling function does.

A function where all recursive calls are tail calls is said to be *tail recursive*.

Why tail recursion?

If a function is tail recursive, the compiler is clever enough to realize that the same area of storage can be used for the local variables of each instance.

We avoid using all that stack.

The additional argument `x` of the `tfact` function is called an *accumulator*, because it accumulates the result as the recursive calls stack up, and is then returned at the end.

Working in this way, rather than modifying the return value on the way back up, is a common way of making functions tail recursive.

In a sense, a tail recursive implementation using accumulators corresponds to an iterative version using assignments and while-loops.

The only difference is that we pass the state around explicitly.

Forcing evaluation (1)

Recall that ML does not evaluate inside lambdas. Therefore it sometimes pays to pull expressions outside a lambda when they do not depend on the value of the bound variable.

For example, we might code the efficient factorial by making `tfact` local:

```
#let fact n =
    let rec tfact x n =
      if n = 0 then x
      else tfact (x * n) (n - 1) in
    tfact 1 n;;
```

However the binding of the recursive function to `tfact` does not get evaluated until `fact` sees its arguments.

Moreover, it gets reevaluated each time even though it doesn't depend on `n`.

Forcing evaluation (2)

A better coding is as follows:

```
#let fact =
    let rec tfact x n =
      if n = 0 then x
      else tfact (x * n) (n - 1) in
    tfact 1;;
```

This is about 20% faster when called on the argument `6`. In cases where the subexpression involves much more evaluation, the difference can be spectacular.

Most compilers do not do such optimizations automatically.

However it falls under the general heading of *partial evaluation*, a big research field.

Minimizing consing (1)

The space used by type constructors ('cons cells') is not allocated and deallocated in such as straightforward way as stack space.

In general, it is difficult to work out when a certain cons cell is in use, and when it is available for recycling. For example in:

```
let l = 1::[] in tl l;;
```

the cons cell can be reused immediately. However if `l` is passed to other functions, it is impossible to decide at compile-time when the cons cell is no longer needed.

Therefore, space in functional languages has to be reclaimed by analyzing memory periodically and deciding which bits are needed. The remainder is then reclaimed. This process is known as *garbage collection*.

Minimizing consing (2)

We can often make programs more space and time efficient by reducing consing. One simple trick is to reduce the usage of `append`. By looking at the definition:

```
#let rec append l1 l2 =
   match l1 with
     [] -> l2
   | (h::t) -> h::(append t l2);;
```

we can see that it generates n cons cells where n is the length of the first argument. For example, our implementation of reversal:

```
#let rec rev =
   fun [] -> []
     | (h::t) -> append (rev t) [h];;
```

is very inefficient, generating about $n^2/2$ cons cells, where n is the length of the list.

Minimizing consing (3)

A far better version is:

```
#let rev =
    let rec reverse acc =
      fun [] -> acc
        | (h::t) -> reverse (h::acc) t in
    reverse [];;
```

This only generates n cons cells, and has the additional merit of being tail recursive, so we save stack space.

One can also avoid consing in pattern-matching by using `as`, e.g. instead of rebuilding a cons cell:

```
fun [] -> []
  | (h::t) -> if h < 0 then t else h::t;;
```

using

```
fun [] -> []
  | (h::t as l) -> if h < 0 then t else l;;
```

Exceptions (1)

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

```
#1 / 0;;
Uncaught exception: Division_by_zero
```

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;;
Exception Died defined.
#exception Failed of string;;
Exception Failed defined.
```

Exceptions (2)

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

```
#raise (Failed "I don't know why");;
Uncaught exception:
  Failed "I don't know why"
```

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

```
#exception Head_of_empty;;
Exception Head_of_empty defined.
#let hd = fun [] -> raise Head_of_empty
            | (h::t) -> h;;
hd : 'a list -> 'a = <fun>
```

Exceptions (3)

One can *catch* exceptions using `try ...with` followed by a series of patterns to match exceptions, e.g.

```
#let headstring sl =
    try hd sl
    with Head_of_empty -> ""
       | Failed s -> "Failure because "^s;;
headstring : string list -> string = <fun>
#headstring ["hi"; "there"];;
- : string = "hi"
#headstring [];;
- : 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!

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:

```
#let x = ref 1;;
x : int ref = ref 1
#!x;;
- : int = 1
#x := 2;;
- : unit = ()
#!x;;
- : int = 2
#x := !x + !x;;
- : unit = ()
#x;;
- : int ref = ref 4
#!x;;
- : int = 4
```

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

Arrays (1)

As well as individual reference cells, one can use arrays, or *vectors* in the usual CAML parlance.

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

```
#make_vect n x;;
```

One can then read element `m` of a vector `v` using:

```
#vect_item v m;;
```

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

```
#vect_assign v m y;;
```

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

Arrays (2)

Here is a simple example:

```
#let v = make_vect 5 0;;
v : int vect = [|0; 0; 0; 0; 0|]
#vect_item v 1;;
- : int = 0
#vect_assign v 1 10;;
- : unit = ()
#v;;
- : int vect = [|0; 10; 0; 0; 0|]
#vect_item v 1;;
- : int = 10
```

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

```
#vect_item v 5;;
Uncaught exception:
  Invalid_argument "vect_item"
```

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:

```
#let l = ref [];;
l : '_a list ref = ref []
#l := [1];;
#hd(!l) = true;;
```

ML places restrictions on the polymorphic type of expressions involving references to avoid these problems. The type is not universal in the usual sense, merely as yet undetermined. The first use fixes the type.