Chapter 8

Effective ML

In this chapter, we discuss some of the techniques and tricks that ML programmers can use to make programs more elegant and more efficient. We then go on to discuss some additional *imperative* features that can be used when the purely functional style seems inappropriate.

8.1 Useful combinators

The flexibility of higher order functions often means that one can write some very useful little functions that can be re-used for a variety of related tasks. These are often called *combinators*, and not simply because they can be regarded as lambda terms with no free variables. It often turns out that these functions are so flexible that practically anything can be implemented by plugging them together, rather than, say, explicitly making a recursive definition. In this way they correspond to the original view of combinators as ubiquitous building blocks for mathematical expressions.

For example, a very useful combinator for list operations, often called '`itlist`' or '`fold`', performs the following operation:

$$\texttt{itlist}\ f\ [x_1;\ x_2;\ \ldots\ ; x_n]\ b = f\ x_1\ (f\ x_2\ (f\ x_3\ (\cdots (f\ x_n\ b))))$$

A straightforward definition in ML is:

```
#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>
```

Quite commonly, when defining a recursive function over lists, all one is doing is repeatedly applying some operator in this manner. By using `itlist` with the appropriate argument, one can implement such functions very easily without explicit use of recursion. A typical use is a function to add all the elements of a list of numbers:

```
#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
```

Those especially keen on brevity might prefer to code `sum` as:

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

It is easy to modify this function to form a product rather than a sum:

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

Many useful list operations can be implemented in this way. For example here is a function to filter out only those elements of a list satisfying a predicate:

```
#let filter p l = itlist (fun x s -> if p x then x::s else s) l [];;
filter : ('a -> bool) -> 'a list -> 'a list = <fun>
#filter (fun x -> x mod 2 = 0) [1;6;4;9;5;7;3;2];;
- : int list = [6; 4; 2]
```

Here are functions to find whether either all or some of the elements of a list satisfy a predicate:

```
#let forall p l = itlist (fun h a -> p(h) & a) l true;;
forall : ('a -> bool) -> 'a list -> bool = <fun>
#let exists p l = itlist (fun h a -> p(h) or a) l false;;
exists : ('a -> bool) -> 'a list -> bool = <fun>
#forall (fun x -> x < 3) [1;2];;
- : bool = true
#forall (fun x -> x < 3) [1;2;3];;
- : bool = false
```

and here are alternative versions of old favourites `length`, `append` and `map`:

```
#let length l = itlist (fun x s -> s + 1) l 0;;
length : 'a list -> int = <fun>
#let append l m = itlist (fun h t -> h::t) l m;;
append : 'a list -> 'a list -> 'a list = <fun>
#let map f l = itlist (fun x s -> (f x)::s) l [];;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Some of these functions can themselves become useful combinators, and so on upwards. For example, if we are interested in treating lists as sets, i.e. avoiding duplicate elements, then many of the standard set operations can be expressed very simply in terms of the combinators above:

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

The `setify` function is supposed to turn a list into a set by eliminating any duplicate elements.

8.2 Writing efficient code

Here we accumulate some common tricks of the trade, which can often make ML programs substantially more efficient. In order to justify some of them, we need to sketch in general terms how certain constructs are executed in hardware.

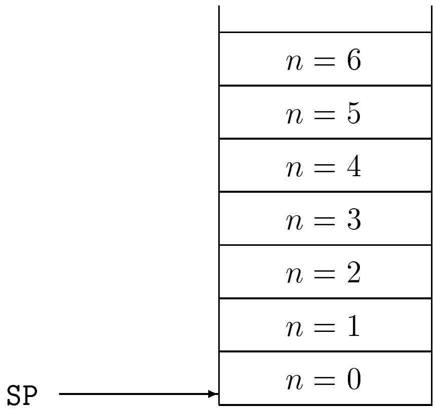
8.2.1 Tail recursion and accumulators

The principal control mechanism in functional programs is recursion. If we are interested in efficient programs, it behoves us to think a little about how recursion is implemented on conventional hardware. In fact, there is not, in this respect at least, much difference between the implementation of ML and many other languages with dynamic variables, such as C.

If functions cannot be called recursively, then we are safe in storing their local variables (which includes the values of arguments) at a fixed place in memory — this is what FORTRAN does. However, this is not possible in general if the function can be called recursively. A call to a function `f` with one set of arguments may include within it a call to `f` with a different set of arguments. The old ones would be overwritten, even if the outer version of f needs to refer to them again after the inner call has finished. For example, consider the factorial function yet again:

```
#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 when this call is finished and the value of `fact 5` is obtained, we still need the original value of `n`, namely 6, in order to do the multiplication yielding the final result. What normally happens in implementations is that each function call is allocated a new frame on a *stack*. Every new function call moves the stack pointer further down[1] the stack, creating space for new variables. When the function call is finished the stack pointer moves up and so the unwanted inner variables are discarded automatically. A diagram may make this clearer:



This is an imagined snapshot of the stack during execution of the innermost recursive call, i.e. `fact 0`. All the local variables for the upper stages are stacked up above, with each instance of the function having its own stack frame, and when the calls are finished the stack pointer `SP` moves back up.

Therefore, our implementation of `fact` requires n stack frames when applied to argument n. 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
```

Although `tfact` is also recursive, 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), and a function where all recursive calls are tail calls is said to be *tail recursive*.

[1]Despite the name, stacks conventionally grow downwards.

What is significant about tail calls? When making a recursive call to `tfact`, there is no need to preserve the old values of the local variables. Exactly the same, fixed, area of storage can be used. This of course depends on the compiler's being intelligent enough to recognize the fact, but most compilers, including CAML Light, are. Consequently, re-coding a function so that the recursive core of it is tail recursive can dramatically cut down the use of storage. For functions like the factorial, it is hardly likely that they will be called with large enough values of n to make the stack overflow. However the naive implementations of many list functions can cause such an effect when the argument lists are long.

The additional argument `x` of the `tfact` function is called an *accumulator*, because it accumulates the result as the recursive calls rack 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.

We have remarked that a fixed area of storage can be used for the arguments to a tail recursive function. On this view, one can look at a tail recursive function as a thinly-veiled imperative implementation. There is an obvious parallel with our C implementation of the factorial as an iterative function:

```
int fact(int n)
{ int x = 1;
  while (n > 0)
   { x = x * n;
     n = n - 1;
   }
  return x;
}
```

The initialization `x = 1` corresponds to our setting of `x` to 1 by an outer wrapper function `fact`. The central while loop corresponds to the recursive calls, the only difference being that the arguments to the tail recursive function make explicit that part of the state we are interested in assigning to. Rather than assigning and looping, we make a recursive call with the variables updated. Using similar tricks and making the state explicit, one can easily write essentially imperative code in an ostensibly functional style, with the knowledge that under standard compiler optimizations, the effect inside the machine will, in fact, be much the same.

8.2.2 Minimizing consing

We have already considered the use of stack space. But various constructs in functional programs use another kind of store, usually allocated from an area called the *heap*. Whereas the stack grows and shrinks in a sequential manner based on the flow of control between functions, other storage used by the ML system cannot be reclaimed in such a simple way. Instead, the runtime system occasionally needs to check which bits of allocated memory aren't being used any more, and reclaim them for future applications, a process known as *garbage collection*. A particularly important example is the space used by constructors for recursive types, e.g. `::`. For example, when the following fragment is executed:

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

a new block of memory, called a 'cons cell', is allocated to store the instance of the `::` constructor. Typically this might be three words of storage, one being an identifier for the constructor, and the other two being pointers to the head and tail of the list. Now in general, it is difficult to decide when this memory can be reclaimed. In the above example, we immediately select the tail of the list, so it is clear that the cons cell can be recycled immediately. But in general this can't be decided by looking at the program, since l might be passed to various functions that may or may not just look at the components of the list. Instead, one needs to analyze the memory usage dynamically and perform garbage collection of what is no longer needed. Otherwise one would eventually run out of storage even when only a small amount is ever needed simultaneously.

Implementors of functional languages work hard on making garbage collection efficient. Some claim that automatic memory allocation and garbage collection often works out faster than typical uses of explicit memory allocation in languages like C (`malloc` etc.) While we wouldn't go that far, it is certainly very convenient that memory allocation is always done automatically. It avoids a lot of tedious and notoriously error-prone parts of programming.

Many constructs beloved of functional programmers use storage that needs to be reclaimed by garbage collection. While worrying too much about this would cripple the style of functional programs, there are some simple measures that can be taken to avoid gratuitous consing (creation of cons cells). One very simple rule of thumb is to avoid using `append` if possible. As can be seen by considering the way the recursive calls unroll according to the definition

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

this typically generates n cons cells where n is the length of the first argument list. There are often ways of avoiding appending, such as adding extra accumulator arguments to functions that can be augmented by direct use of consing. A striking example is the list reversal function, which we coded earlier as:

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

This typically generates about $n^2/2$ cons cells, where n is the length of the list. The following alternative, using an accumulator, only generates n of them:

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

Moreover, the recursive core `reverse` is tail recursive, so we also save stack space, and win twice over.

For another typical situation where we can avoid appending by judicious use of accumulators, consider the problem of returning the fringe of a binary tree, i.e. a list of the leaves in left-to-right order. If we define the type of binary trees as:

```
#type btree = Leaf of string
            | Branch of btree * btree;;
```

then a simple coding is the following

```
#let rec fringe =
   fun (Leaf s) -> [s]
     | (Branch(l,r)) -> append (fringe l) (fringe r);;
```

However the following more refined version performs fewer conses:

```
#let fringe =
   let rec fr t acc =
     match t with
       (Leaf s) -> s::acc
     | (Branch(l,r)) -> fr l (fr r acc) in
   fun t -> fr t [];;
```

Note that we have written the accumulator as the second argument, so that the recursive call has a more natural left-to-right reading. Here is a simple example of how either version of `fringe` may be used:

```
#fringe (Branch(Branch(Leaf "a",Leaf "b"),
                Branch(Leaf "c",Leaf "d")));;
- : string list = ["a"; "b"; "c"; "d"]
```

The first version creates 6 cons cells, the second only 4. On larger trees the effect can be more dramatic. Another situation where gratuitous consing can crop up is in pattern matching. For example, consider the code fragment:

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

The 'else' arm creates a cons cell even though what it constructs was in fact the argument to the function. That is, it is taking the argument apart and then rebuilding it. One simple way of avoiding this is to recode the function as:

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

However ML offers a more flexible alternative: using the `as` keyword, a name may be identified with certain components of the pattern, so that it never needs to be rebuilt. For example:

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

8.2.3 Forcing evaluation

We have emphasized that, since ML does not evaluate underneath lambdas, one can use lambdas to delay evaluation. We will see some interesting examples later. Conversely, however, it can happen that one wants to force evaluation of expressions that are hidden underneath lambdas. For example, recall the tail recursive factorial above:

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

Since we never really want to use `tfact` directly, it seems a pity to bind it to a name. Instead, we can make it local to the factorial function:

```
#let fact1 n =
   let rec tfact x n =
     if n = 0 then x
     else tfact (x * n) (n - 1) in
   tfact 1 n;;
```

This, however, has the defect that the local recursive definition is only evaluated after `fact1` receives its argument, since before that it is hidden under a lambda. Moreover, it is then reevaluated each time `fact` is called. We can change this as follows

```
#let fact2 =
   let rec tfact x n =
     if n = 0 then x
     else tfact (x * n) (n - 1) in
   tfact 1;;
```

Now the local binding is only evaluated once, at the point of declaration of `fact2`. According to our tests, the second version of `fact` is about 20% faster when called on the argument `6`. The additional evaluation doesn't amount to much in this case, more or less just unravelling a recursive definition, yet the speedup is significant. In instances where there is a lot of computation involved in evaluating the local binding, the difference can be spectacular. In fact, there is a sophisticated research field of 'partial evaluation' devoted to performing optimizations like this, and much more sophisticated ones besides, automatically. In a sense, it is a generalization of standard compiler optimizations for ordinary languages such as 'constant folding'. In production ML systems, however, it is normally the responsibility of the user to force it, as here.

We might note, in passing, that if functions are implemented by plugging together combinators, with fewer explicit lambdas, there is more chance that as much of the expression as possible will be evaluated at declaration time. To take a trivial example, $f \circ g$ will perform any evaluation of f and g that may be possible, whereas $\lambda x.\ f(g\ x)$ will perform none at all until it receives its argument. On the other side of the coin, when we actually *want* to delay evaluation, we really need lambdas, so a purely combinatory version is impossible.

8.3 Imperative features

ML has a fairly full complement of imperative features. We will not spend much time on the imperative style of programming, since that is not the focus of this course, and we assume readers already have sufficient experience. Therefore, we treat these topics fairly quickly with few illustrative examples. However some imperative features are used in later examples, and some knowledge of what is available will stand the reader in good stead for writing practical ML code.

8.3.1 Exceptions

We have seen on occasion that certain evaluations fail, e.g. through a failure in pattern matching. There are other reasons for failure, e.g. attempts to divide by zero.

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

In all these cases the compiler complains about an 'uncaught exception'. An exception is a kind of error indication, but they need not always be propagated to the top level. There is a type `exn` of *exceptions*, which is effectively a recursive type, though it is usually recursive only vacuously. 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.
```

While certain built-in operations generate (one usually says *raise*) exceptions, this can also be done explicitly 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>
#hd [];;
Uncaught exception: Head_of_empty
```

Normally exceptions propagate out to the top, but they can be 'caught' inside an outer expression by 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 = ""
```

It is a matter of opinion whether exceptions are really an imperative feature. On one view, functions just return elements of a disjoint sum consisting of their visible return type and the type of exceptions, and all operations implicitly pass back exceptions. Another view is that exceptions are a highly non-local control flow perversion, analogous to `goto`.[2] Whatever the semantic view one takes, exceptions can often be quite useful.

[2] Perhaps more precisely, to C's `setjmp` and `longjmp`.

8.3.2 References and arrays

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 references themselves behave more like ordinary ML values. Actually this approach is quite common in C too. For example, if one wants so-called 'variable parameters' in C, where changes to the local variables propagate outside, the only way to do it is to pass a pointer, so that the function can dereference it. Similar techniques are often used where the function is to pass back composite data.

In ML, one sets up a new assignable memory cell with the initial contents `x` by writing `ref x`. (Initialization is compulsory.) This expression yields a reference (pointer) to the cell. Subsequent access to the contents of the cell requires an explicit dereference using the `!` operator, similar to unary `*` in C. The cell is assigned to using a conventional-looking assignment statement. For example:

```
#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
```

Note that in most respects `ref` behaves like a type constructor, so one can pattern-match against it. Thus one could actually define an indirection operator like `!`:

```
#let contents_of (ref x) = x;;
contents_of : 'a ref -> 'a = <fun>
#contents_of x;;
- : int = 4
```

As well as being mutable, references are sometimes useful for creating explicitly shared data structures. One can easily create graph structures where numerous nodes contain a pointer to some single subgraph.

Apart from single cells, one can also use arrays in ML. In CAML these are called `vectors`. An array of elements of type α has type α *vect*. A fresh vector of size `n`, with each element initialized to `x` — once again the initialization is compulsory — 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;;
```

These operations correspond to the expressions `v[m]` and `v[m] = y` in C. The elements of an array are numbered from zero. For 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"
```

8.3.3 Sequencing

There is no need for an explicit sequencing operation in ML, since the normal rules of evaluation allow one to impose an order. For example one can do:

```
#let _ =  x := !x + 1 in
 let _ =  x := !x + 1 in
 let _ =  x := !x + 1 in
 let _ =  x := !x + 1 in
 ();;
```

and the expressions are evaluated in the expected order. Here we use a special pattern _ which throws away the value, but we could use a dummy variable name instead. Nevertheless, it is more attractive to use the conventional notation for sequencing, and this is possible in ML by using a single semicolon:

```
#x := !x + 1;
 x := !x + 1;
 x := !x + 1;
 x := !x + 1;;
```

8.3.4 Interaction with the type system

While polymorphism works very well for the pure functional core of ML, it has unfortunate interactions with some imperative features. For example, consider the following:

```
#let l = ref [];;
```

Then `l` would seem to have polymorphic type α *list ref*. In accordance with the usual rules of let-polymorphism we should be able to use it with two different types, e.g. first

```
#l := [1];;
```

and then

```
#hd(!l) = true;;
```

But this isn't reasonable, because we would actually be writing something as an object of type `int` then reading it as an object of type `bool`. Consequently, some restriction on the usual rule of let polymorphism is called for where references are concerned. There have been many attempts to arrive at a sound but convenient restriction of the ML type system, some of them very complicated. Recently, different versions of ML seem to be converging on a relatively simple method, called the *value restriction*, due to Wright (1996), and CAML implements this restriction, with a twist regarding toplevel bindings. Indeed, the above sequence fails. But the intermediate behaviour is interesting. If we look at the first line we see:

```
#let l = ref [];;
l : '_a list ref = ref []
```

The underscore on the type variable indicates that `l` is not polymorphic in the usual sense; rather, it has a single fixed type, although that type is as yet undetermined. The second line works fine:

```
#l := [1];;
- : unit = ()
```

but if we now look at the type of `l`, we see that:

```
#l;;
- : int list ref = ref [1]
```

The pseudo-polymorphic type has now been fixed. Granted this, it is clear that the last line must fail:

```
#hd(!l) = true;;
Toplevel input:
>hd(!l) = true;;
>         ^^^^
This expression has type bool,
but is used with type int.
```

So far, this seems quite reasonable, but we haven't yet explained why the same underscored type variables oocur in apparently quite innocent purely functional expressions, and why, moreover, they often disappear on eta-expansion, e.g.

```
#let I x = x;;
I : 'a -> 'a = <fun>
#I o I;;
it : '_a -> '_a = <fun>
#let I2 = I o I in fun x -> I2 x;;
- : '_a -> '_a = <fun>
#fun x -> (I o I) x;;
it : 'a -> 'a = <fun>
```

Other techniques for polymorphic references often rely on encoding in the types the fact that an expression may involve references. This seems natural, but it can lead to the types of functions becoming cluttered with this special information. It is unattractive that the particular implementation of the function, e.g. imperative or functional, should be reflected in its type.

Wright's solution, on the other hand, uses just the basic syntax of the expression being let-bound, insisting that it is a so-called *value* before generalizing the type. What is really wanted is knowledge of whether the expression may cause side-effects when evaluated. However since this is undecidable in general, the simple syntactic criterion of its being or not being a value is used. Roughly speaking, an expression is a value if it admits no further evaluation according to the ML rules — this is why an expression can often be made into a value by performing a reverse eta conversion. Unfortunately this works against the techniques for forcing evaluation.

Exercises

1. Define the C combinator as:

```
#let C f x y = f y x;;
```

What does the following function do?

```
#fun f l1 l2 -> itlist (union o C map l2 o f) l1 [];;
```

2. What does this function do? Write a more efficient version.

```
#let rec upto n = if n < 1 then [] else append (upto (n-1)) [n];;
```

3. Define a function to calculate Fibonacci numbers as follows:

```
#let rec fib =
   fun 0 -> 0
     | 1 -> 1
     | n -> fib(n - 2) + fib(n - 1);;
```

   Why is this very inefficient? Write a similar but better version.

4. (*) Can you think of any uses for this exception or similar recursive ones?

```
#exception Recurse of exn;;
```

5. Implement a simple version of quicksort using arrays. The array is first partitioned about some pivot, then two recursive calls sort the left and right portions. Which recursive calls are tail calls? What is the worst-case space usage? How can this be improved dramatically by a simple change?

6. Prove that the two versions of `rev` that we have quoted, inefficient and efficient, always give the same answer.