Chapter 11

Higher-Order Functions

11.1 Functions as Values

Values of function type are *first-class*, which means that they have the same rights and privileges as values of any other type. In particular, functions may be passed as arguments and returned as results of other functions, and functions may be stored in and retrieved from data structures such as lists and trees. We will see that first-class functions are an important source of expressive power in ML.

Functions which take functions as arguments or yield functions as results are known as *higher-order functions* (or, less often, as *functionals* or *operators*). Higher-order functions arise frequently in mathematics. For example, the differential operator is the higher-order function that, when given a (differentiable) function on the real line, yields its first derivative as a function on the real line. We also encounter functionals mapping functions to real numbers, and real numbers to functions. An example of the former is provided by the definite integral viewed as a function of its integrand, and an example of the latter is the definite integral of a given function on the interval $[a, x]$, viewed as a function of a, that yields the area under the curve from a to x as a function of x.

Higher-order functions are less familiar tools for many programmers since the best-known programming languages have only rudimentary mechanisms to support their use. In contrast higher-order functions play a prominent role in ML, with a variety of interesting applications. Their use may be classified into two broad categories:

1. *Abstracting patterns of control.* Higher-order functions are design patterns that "abstract out" the details of a computation to lay bare the skeleton of the solution. The skeleton may be fleshed out to form a solution of a problem by applying the general pattern to arguments that isolate the specific problem instance.

2. *Staging computation.* It arises frequently that computation may be *staged* by expending additional effort "early" to simplify the computation of "later" results. Staging can be used both to improve efficiency and, as we will see later, to control sharing of computational resources.

11.2 Binding and Scope

Before discussing these programming techniques, we will review the critically important concept of *scope* as it applies to function definitions. Recall that Standard ML is a *statically scoped* language, meaning that identifiers are resolved according to the static structure of the program. A *use* of the variable *var* is considered to be a reference to the *nearest lexically enclosing declaration of var*. We say "nearest" because of the possibility of shadowing; if we re-declare a variable *var*, then subsequent uses of *var* refer to the "most recent" (lexically!) declaration of it; any "previous" declarations are temporarily shadowed by the latest one.

This principle is easy to apply when considering sequences of declarations. For example, it should be clear by now that the variable `y` is bound to `32` after processing the following sequence of declarations:

```
val x = 2           (* x=2 *)
val y = x*x         (* y=4 *)
val x = y*x         (* x=8 *)
val y = x*y         (* y=32 *)
```

In the presence of function definitions the situation is the same, but it can be a bit tricky to understand at first.

Here's an example to test your grasp of the lexical scoping principle:

```
val x = 2
fun f y = x+y
val x = 3
val z = f 4
```

After processing these declarations the variable `z` is bound to `6`, not to `7`! The reason is that the occurrence of `x` in the body of `f` refers to the *first* declaration of `x` since it is the nearest lexically enclosing declaration of the occurence, *even though* it has been subsequently re-declared.

This example illustrates three important points:

1. Binding is not assignment! If we were to view the second binding of `x` as an assignment statement, then the value of `z` would be `7`, not `6`.

2. Scope resolution is *lexical*, not *temporal.* We sometimes refer to the "most recent" declaration of a variable, which has a temporal flavor, but we always mean "nearest lexically enclosing at the point of occurrence".

3. "Shadowed" bindings are not lost. The "old" binding for `x` is still available (through calls to `f`), even though a more recent binding has shadowed it.

One way to understand what's going on here is through the concept of a *closure*, a technique for implementing higher-order functions. When a function expression is evaluated, a copy of the environment is attached to the function. Subsequently, all free variables of the function (*i.e.*, those variables not occurring as parameters) are resolved with respect to the environment attached to the function; the function is therefore said to be "closed" with respect to the attached environment. This is achieved at function application time by "swapping" the attached environment of the function for the environment active at the point of the call. The swapped environment is restored after the call is complete. Returning to the example above, the environment associated with the function `f` contains the declaration `val x = 2` to record the fact that at the time the function was evaluated, the variable `x` was bound to the value `2`. The variable `x` is subsequently re-bound to `3`, but when `f` is applied, we temporarily reinstate the binding of `x` to `2`, add a binding of `y` to `4`, then evaluate the body of the function, yielding `6`. We then restore the binding of `x` and drop the binding of `y` before yielding the result.

11.3 Returning Functions

While seemingly very simple, the principle of lexical scope is the source of considerable expressive power. We'll demonstrate this through a series of examples.

To warm up let's consider some simple examples of passing functions as arguments and yielding functions as results. The standard example of passing a function as argument is the `map'` function, which applies a given function to every element of a list. It is defined as follows:

```
fun map' (f, nil) = nil
  | map' (f, h::t) = (f h) :: map' (f, t)
```

For example, the application

```
map' (fn x => x+1, [1,2,3,4])
```

evaluates to the list `[2,3,4,5]`.

Functions may also yield functions as results. What is surprising is that we can create *new* functions during execution, not just return functions that have been previously defined. The most basic (and deceptively simple) example is the function `constantly` that creates constant functions: given a value `k`, the application `constantly k` yields a function that yields `k` whenever it is applied. Here's a definition of `constantly`:

```
val constantly = fn k => (fn a => k)
```

The function constantly has type `'a -> ('b -> 'a)`. We used the `fn` notation for clarity, but the declaration of the function `constantly` may also be written using `fun` notation as follows:

```
fun constantly k a = k
```

Note well that a *white space* separates the two successive arguments to `constantly`! The meaning of this declaration is precisely the same as the earlier definition using `fn` notation.

The value of the application `constantly 3` is the function that is constantly `3`; *i.e.*, it always yields `3` when applied. Yet nowhere have we defined the function that always yields `3`. The resulting function is "created" by the application of `constantly` to the argument `3`, rather than merely

"retrieved" off the shelf of previously-defined functions. In implementation terms the result of the application `constantly 3` is a closure consisting of the function `fn a => k` with the environment `val k = 3` attached to it. The closure is a data structure (a pair) that is created by each application of `constantly` to an argument; the closure is the representation of the "new" function yielded by the application. Notice, however, that the *only* difference between any two results of applying the function `constantly` lies in the attached environment; the underlying function is *always* `fn a => k`. If we think of the lambda as the "executable code" of the function, then this amounts to the observation that no new *code* is created at run-time, just new *instances* of existing code.

This also points out why functions in ML are not the same as code pointers in C. You may be familiar with the idea of passing a pointer to a C function to another C function as a means of passing functions as arguments or yielding functions as results. This may be considered to be a form of "higher-order" function in C, but it must be emphasized that code pointers are significantly less powerful than closures because in C there are only *statically many* possibilities for a code pointer (it must point to one of the functions defined in your code), whereas in ML we may generate *dynamically many* different instances of a function, differing in the bindings of the variables in its environment. The non-varying part of the closure, the code, is directly analogous to a function pointer in C, but there is no counterpart in C of the varying part of the closure, the dynamic environment.

The definition of the function `map'` given above takes a function and list as arguments, yielding a new list as result. Often it occurs that we wish to map the same function across several different lists. It is inconvenient (and a tad inefficient) to keep passing the same function to `map'`, with the list argument varying each time. Instead we would prefer to create a instance of map specialized to the given function that can then be applied to many different lists. This leads to the following definition of the function `map`:

```
fun map f nil = nil
  | map f (h::t) = (f h) :: (map f t)
```

The function `map` so defined has type `('a->'b) -> 'a list -> 'b list`. It takes a function of type `'a -> 'b` as argument, and yields another function of type `'a list -> 'b list` as result.

The passage from `map'` to `map` is called *currying*. We have changed a two-argument function (more properly, a function taking a pair as argument) into a function that takes two arguments in succession, yielding after the first a function that takes the second as its sole argument. This passage can be codified as follows:

```
fun curry f x y = f (x, y)
```

The type of `curry` is

```
('a*'b->'c) -> ('a -> ('b -> 'c)).
```

Given a two-argument function, `curry` returns another function that, when applied to the first argument, yields a function that, when applied to the second, applies the original two-argument function to the first and second arguments, given separately.

Observe that `map` may be alternately defined by the binding

```
fun map f l = curry map' f l
```

Applications are implicitly left-associated, so that this definition is equivalent to the more verbose declaration

```
fun map f l = ((curry map') f) l
```

11.4 Patterns of Control

We turn now to the idea of abstracting patterns of control. There is an obvious similarity between the following two functions, one to add up the numbers in a list, the other to multiply them.

```
fun add_up nil = 0
  | add_up (h::t) = h + add_up t
fun mul_up nil = 1
  | mul_up (h::t) = h * mul_up t
```

What precisely is the similarity? We will look at it from two points of view.

One view is that in each case we have a binary operation and a unit element for it. The result on the empty list is the unit element, and the result on a non-empty list is the operation applied to the head of the list and the result on the tail. This pattern can be abstracted as the function `reduce` defined as follows:

```
fun reduce (unit, opn, nil) =
    unit
  | reduce (unit, opn, h::t) =
    opn (h, reduce (unit, opn, t))
```

Here is the type of `reduce`:

```
val reduce : 'b * ('a*'b->'b) * 'a list -> 'b
```

The first argument is the unit element, the second is the operation, and the third is the list of values. Notice that the type of the operation admits the possibility of the first argument having a different type from the second argument and result.

Using reduce, we may re-define `add_up` and `mul_up` as follows:

```
fun add_up l = reduce (0, op +, l)
fun mul_up l = reduce (1, op *, l)
```

To further check your understanding, consider the following declaration:

```
fun mystery l = reduce (nil, op ::, l)
```

(Recall that "`op ::`" is the function of type `'a * 'a list -> 'a list` that adds a given value to the front of a list.) What function does `mystery` compute?

Another view of the commonality between `add_up` and `mul_up` is that they are both defined by induction on the structure of the list argument, with a base case for `nil`, and an inductive case for `h::t`, defined in terms of its behavior on `t`. But this is really just another way of saying that they are defined in terms of a unit element and a binary operation! The difference is one of perspective: whether we focus on the pattern part of the clauses (the inductive decomposition) or the result part of the clauses (the unit and operation). The recursive structure of `add_up` and `mul_up` is abstracted by the `reduce` functional, which is then specialized to yield `add_up` and `mul_up`. Said another way, *the function reduce abstracts the pattern of defining a function by induction on the structure of a list*.

The definition of `reduce` leaves something to be desired. One thing to notice is that the arguments `unit` and `opn` are carried unchanged through the recursion; only the list parameter changes on recursive calls. While this might seem like a minor overhead, it's important to remember that

multi-argument functions are really single-argument functions that take a tuple as argument. This means that each time around the loop we are constructing a new tuple whose first and second components remain fixed, but whose third component varies. Is there a better way? Here's another definition that isolates the "inner loop" as an auxiliary function:

```
fun better_reduce (unit, opn, l) =
    let
        fun red nil = unit
          | red (h::t) = opn (h, red t)
    in
        red l
    end
```

Notice that each call to `better_reduce` creates a *new* function `red` that uses the parameters `unit` and `opn` of the call to `better_reduce`. This means that `red` is bound to a closure consisting of the code for the function together with the environment active at the point of definition, which will provide bindings for `unit` and `opn` arising from the application of `better_reduce` to its arguments. Furthermore, the recursive calls to `red` no longer carry bindings for `unit` and `opn`, saving the overhead of creating tuples on each iteration of the loop.

11.5 Staging

An interesting variation on `reduce` may be obtained by *staging* the computation. The motivation is that `unit` and `opn` often remain fixed for many different lists (*e.g.*, we may wish to sum the elements of many different lists). In this case `unit` and `opn` are said to be "early" arguments and the list is said to be a "late" argument. The idea of staging is to perform as much computation as possible on the basis of the early arguments, yielding a function of the late arguments alone.

In the case of the function `reduce` this amounts to building `red` on the basis of `unit` and `opn`, yielding it as a function that may be later applied to many different lists. Here's the code:

```
fun staged_reduce (unit, opn) =
    let
        fun red nil = unit
          | red (h::t) = opn (h, red t)
    in
        red
    end
```

The definition of `staged_reduce` bears a close resemblance to the definition of `better_reduce`; the only difference is that the creation of the closure bound to `red` occurs *as soon as* `unit` *and* `opn` *are known*, rather than each time the list argument is supplied. Thus the overhead of closure creation is "factored out" of multiple applications of the resulting function to list arguments.

We could just as well have replaced the body of the let expression with the function

```
fn l => red l
```

but a moment's thought reveals that the meaning is the same.

Note well that we would *not* obtain the effect of staging were we to use the following definition:

```
fun curried_reduce (unit, opn) nil = unit
  | curried_reduce (unit, opn) (h::t) =
    opn (h, curried_reduce (unit, opn) t)
```

If we unravel the `fun` notation, we see that while we are taking two arguments in succession, we are *not* doing any useful work in between the arrival of the first argument (a pair) and the second (a list). A *curried* function does not take significant advantage of staging. Since `staged_reduce` and `curried_reduce` have the same iterated function type, namely

```
('b * ('a * 'b -> 'b)) -> 'a list -> 'b
```

the contrast between these two examples may be summarized by saying *not every function of iterated function type is curried*. Some are, and some aren't. The "interesting" examples (such as `staged_reduce`) are the ones that *aren't* curried. (This directly contradicts established terminology, but it is necessary to deviate from standard practice to avoid a serious misapprehension.)

The time saved by staging the computation in the definition of `staged_reduce` is admittedly minor. But consider the following definition of an append function for lists that takes both arguments at once:

```
fun append (nil, l) = l
  | append (h::t, l) = h :: append(t,l)
```

Suppose that we will have occasion to append many lists to the end of a given list. What we'd like is to build a specialized appender for the first list that, when applied to a second list, appends the second to the end of the first. Here's a naive solution that merely curries append:

```
fun curried_append nil l = l
  | curried_append (h::t) l = h :: append t l
```

Unfortunately this solution doesn't exploit the fact that the first argument is fixed for many second arguments. In particular, each application of the result of applying `curried_append` to a list results in the first list being traversed so that the second can be appended to it.

We can improve on this by staging the computation as follows:

```
fun staged_append nil = fn l => l
  | staged_append (h::t) =
    let
        val tail_appender = staged_append t
    in
        fn l => h :: tail_appender l
    end
```

Notice that the first list is traversed *once* for all applications to a second argument. When applied to a list $[v_1, \ldots, v_n]$, the function `staged_append` yields a function that is equivalent to, but not quite as efficient as, the function

`fn l =>` v_1 `::` v_2 `:: ... ::` v_n `:: l.`

This still takes time proportional to n, but a substantial savings accrues from avoiding the pattern matching required to destructure the original list argument on each call.

11.6 Sample Code

Here is the code for this chapter.