Chapter 9

Programming with Lists

9.1 List Primitives

In chapter 5 we noted that aggregate data structures are especially easy to handle in ML. In this chapter we consider another important aggregate type, the *list* type. In addition to being an important form of aggregate type it also illustrates two other general features of the ML type system:

1. *Type constructors*, or *parameterized types*. The type of a list reveals the type of its elements.

2. *Recursive types*. The set of values of a list type are given by an inductive definition.

Informally, the values of type *typ* `list` are the finite lists of values of type *typ*. More precisely, the values of type *typ* `list` are given by an *inductive definition*, as follows:

1. `nil` is a value of type *typ* `list`.

2. if h is a value of type *typ*, and t is a value of type *typ* `list`, then $h::t$ is a value of type *typ* `list`.

3. Nothing else is a value of type *typ* `list`.

The type expression *typ* `list` is a postfix notation for the application of the *type constructor* `list` to the type *typ*. Thus `list` is a kind of "function" mapping types to types: given a type *typ*, we may apply `list` to it

to get another type, written *typ* `list`. The forms `nil` and `::` are the *value constructors* of type *typ* `list`. The nullary (no argument) constructor `nil` may be thought of as the empty list. The binary (two argument) constructor `::` constructs a non-empty list from a value h of type *typ* and another value t of type *typ* `list`; the resulting value, h`::`t, of type *typ* `list`, is pronounced "h cons t" (for historical reasons). We say that "h is *cons'd onto* t", that h is the *head* of the list, and that t is its *tail*.

The definition of the values of type *typ* `list` given above is an example of an *inductive definition*. The type is said to be *recursive* because this definition is "self-referential" in the sense that the values of type *typ* `list` are defined in terms of (other) values of the same type. This is especially clear if we examine the types of the value constructors for the type *typ* `list`:

```
val nil : typ list
val (op ::) : typ * typ list -> typ list
```

The notation `op ::` is used to *refer* to the `::` operator as a function, rather than to *use* it to form a list, which requires infix notation.

Two things are notable here:

1. The `::` operation takes as its second argument a value of type *typ* `list`, and yields a result of type *typ* `list`. This self-referential aspect is characteristic of an inductive definition.

2. Both `nil` and `op ::` are *polymorphic* in the type of the underlying elements of the list. Thus `nil` is the empty list of type *typ* `list` for any element type *typ*, and `op ::` constructs a non-empty list independently of the type of the elements of that list.

It is easy to see that a value *val* of type *typ* `list` has the form

val_1`::(`val_2`:: (`$\cdots$ `::(`val_n`::nil)`$\cdots$`))`

for some $n \geq 0$, where val_i is a value of type typ_i for each $1 \leq i \leq n$. For according to the inductive definition of the values of type *typ* `list`, the value *val* must either be `nil`, which is of the above form, or val_1`::`val', where val' is a value of type *typ* `list`. By induction val' has the form

`(`val_2`:: (`$\cdots$ `::(`val_n`::nil)`$\cdots$`))`

and hence *val* again has the specified form.

By convention the operator `::` is *right-associative*, so we may omit the parentheses and just write

val_1`::`val_2`::`$\cdots$`::`val_n`::nil`

as the general form of *val* of type *typ* `list`. This may be further abbreviated using *list notation*, writing

`[` val_1`,` val_2`, ..., ` val_n `]`

for the same list. This notation emphasizes the interpretation of lists as finite sequences of values, but it obscures the fundamental inductive character of lists as being built up from `nil` using the `::` operation.

9.2 Computing With Lists

How do we compute with values of list type? Since the values are defined inductively, it is natural that functions on lists be defined recursively, using a clausal definition that analyzes the structure of a list. Here's a definition of the function `length` that computes the number of elements of a list:

```
fun length nil = 0
    | length (_::t) = 1 + length t
```

The definition is given by induction on the structure of the list argument. The base case is the empty list, `nil`. The inductive step is the non-empty list `_::t` (notice that we do not need to give a name to the head). Its definition is given in terms of the tail of the list `t`, which is "smaller" than the list `_::t`. The type of `length` is `'a list -> int`; it is defined for lists of values of any type whatsoever.

We may define other functions following a similar pattern. Here's the function to append two lists:

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

This function is built into ML; it is written using infix notation as exp_1 `@` exp_2. The running time of `append` is proportional to the length of the first list, as should be obvious from its definition.

Here's a function to reverse a list.

```
fun rev nil = nil
  | rev (h::t) = rev t @ [h]
```

Its running time is $O(n^2)$, where n is the length of the argument list. This can be demonstrated by writing down a recurrence that defines the running time $T(n)$ on a list of length n.

$$\begin{aligned} T(0) &= O(1) \\ T(n+1) &= T(n) + O(n) \end{aligned}$$

Solving the recurrence we obtain the result $T(n) = O(n^2)$.

Can we do better? Oddly, we can take advantage of the *non-associativity* of `::` to give a tail-recursive definition of `rev`.

```
local
    fun helper (nil, a) = a
      | helper (h::t, a) = helper (t, h::a)
in
    fun rev' l = helper (l, nil)
end
```

The general idea of introducing an accumulator is the same as before, except that by re-ordering the applications of `::` we reverse the list! The helper function reverses its first argument and prepends it to its second argument. That is, `helper (l, a)` evaluates to `(rev l) @ a`, where we assume here an independent definition of `rev` for the sake of the specification. Notice that `helper` runs in time proportional to the length of its first argument, and hence `rev'` runs in time proportional to the length of its argument.

The correctness of functions defined on lists may be established using the principle of *structural induction*. We illustrate this by establishing that the function `helper` satisfies the following specification:

> for every l and a of type *typ* `list`, `helper(`l`, `a`)` evaluates to the result of appending a to the reversal of l.

That is, there are no pre-conditions on l and a, and we establish the post-condition that `helper (`l`, `a`)` yields `(rev `l`) @ `a.

The proof is by structural induction on the list l. If l is `nil`, then `helper (`l`,`a`)` evaluates to a, which fulfills the post-condition. If l is the list h`::`t,

then the application `helper (`l`, `a`)` reduces to the value of `helper (`t`, (`h`::`a`))`. By the inductive hypothesis this is just `(rev `t`) @ (`h` :: `a`)`, which is equivalent to `(rev `t`) @ [`h`] @ `a. But this is just `rev (`h`::`t`) @ `a, which was to be shown.

The principle of structural induction may be summarized as follows. To show that a function works correctly for every list l, it suffices to show

1. The correctness of the function for the empty list, `nil`, and
2. The correctness of the function for $h::t$, assuming its correctness for t.

As with mathematical induction over the natural numbers, structural induction over lists allows us to focus on the *basic* and *incremental* behavior of a function to establish its correctness for all lists.

9.3 Sample Code

Here is the code for this chapter.

Chapter 10

Concrete Data Types

10.1 Datatype Declarations

Lists are one example of the general notion of a *recursive type*. ML provides a general mechanism, the `datatype` declaration, for introducing programmer-defined recursive types. Earlier we introduced `type` declarations as an abbreviation mechanism. Types are given names as documentation and as a convenience to the programmer, but doing so is semantically inconsequential — one could replace all uses of the type name by its definition and not affect the behavior of the program. In contrast the `datatype` declaration provides a means of introducing a *new* type that is distinct from all other types and that does not merely stand for some other type. It is the means by which the ML type system may be extended by the programmer.

The `datatype` declaration in ML has a number of facets. A `datatype` declaration introduces

1. One or more new type constructors. The type constructors introduced may, or may not, be mutually recursive.

2. One or more new value constructors for each of the type constructors introduced by the declaration.

The type constructors may take zero or more arguments; a zero-argument, or *nullary*, type constructor is just a type. Each value constructor may also take zero or more arguments; a nullary value constructor is just a constant. The type and value constructors introduced by the declaration are "new" in the sense that they are distinct from all other type and value

constructors previously introduced; if a datatype re-defines an "old" type or value constructor, then the old definition is shadowed by the new one, rendering the old ones inaccessible in the scope of the new definition.

10.2 Non-Recursive Datatypes

Here's a simple example of a nullary type constructor with four nullary value constructors.

```
datatype suit = Spades | Hearts | Diamonds | Clubs
```

This declaration introduces a new type `suit` with four nullary value constructors, `Spades`, `Hearts`, `Diamonds`, and `Clubs`. This declaration may be read as introducing a type `suit` such that a value of type `suit` is either `Spades`, or `Hearts`, or `Diamonds`, or `Clubs`. There is no significance to the ordering of the constructors in the declaration; we could just as well have written

```
datatype suit = Hearts | Diamonds | Spades | Clubs
```

(or any other ordering, for that matter). It is conventional to capitalize the names of value constructors, but this is not required by the language.

Given the declaration of the type `suit`, we may define functions on it by case analysis on the value constructors using a clausal function definition. For example, we may define the suit ordering in the card game of bridge by the function

```
fun outranks (Spades, Spades) = false
  | outranks (Spades, _) = true
  | outranks (Hearts, Spades) = false
  | outranks (Hearts, Hearts) = false
  | outranks (Hearts, _) = true
  | outranks (Diamonds, Clubs) = true
  | outranks (Diamonds, _) = false
  | outranks (Clubs, _) = false
```

This defines a function of type `suit * suit -> bool` that determines whether or not the first `suit` outranks the second.

Data types may be *parameterized* by a type. For example, the declaration

```
datatype 'a option = NONE | SOME of 'a
```

introduces the unary type constructor `'a option` with two value constructors, `NONE`, with no arguments, and `SOME`, with one. The values of type *typ* `option` are

1. The constant `NONE`, and
2. Values of the form `SOME` *val*, where *val* is a value of type *typ*.

For example, some values of type `string option` are `NONE`, `SOME "abc"`, and `SOME "def"`.

The option type constructor is pre-defined in Standard ML. One common use of option types is to handle functions with an optional argument. For example, here is a function to compute the base-*b* exponential function for natural number exponents that defaults to base 2:

```
fun expt (NONE, n) = expt (SOME 2, n)
  | expt (SOME b, 0) = 1
  | expt (SOME b, n) =
    if n mod 2 = 0 then
       expt (SOME (b*b), n div 2)
    else
       b * expt (SOME b, n-1)
```

The advantage of the option type in this sort of situation is that it avoids the need to make a special case of a particular argument, *e.g.*, using `0` as first argument to mean "use the default exponent".

A related use of option types is in aggregate data structures. For example, an address book entry might have a record type with fields for various bits of data about a person. But not all data is relevant to all people. For example, someone may not have a spouse, but they all have a name. For this we might use a type definition of the form

```
type entry = { name:string, spouse:string option }
```

so that one would create an entry for an unmarried person with a `spouse` field of `NONE`.

Option types may also be used to represent an optional result. For example, we may wish to define a function `reciprocal` that returns the reciprocal of an integer, if it has one, and otherwise indicates that it has

no reciprocal. This is achieve by defining `reciprocal` to have type `int -> int option` as follows:

```
fun reciprocal 0 = NONE
  | reciprocal n = SOME (1 div n)
```

To use the result of a call to *reciprocal* we must perform a case analysis of the form

```
case (reciprocal exp
  of NONE => exp1
   | SOME r => exp2
```

where exp_1 covers the case that *exp* has no reciprocal, and exp_2 covers the case that *exp* has reciprocal `r`.

10.3 Recursive Datatypes

The next level of generality is the recursive type definition. For example, one may define a type *typ* `tree` of binary trees with values of type *typ* at the nodes using the following declaration:

```
datatype 'a tree =
  Empty |
  Node of 'a tree * 'a * 'a tree
```

This declaration corresponds to the informal definition of binary trees with values of type *typ* at the nodes:

1. The empty tree `Empty` is a binary tree.
2. If *tree_1* and *tree_2* are binary trees, and *val* is a value of type *typ*, then `Node (`*tree_1*`,` *val*`,` *tree_2*`)` is a binary tree.
3. Nothing else is a binary tree.

The distinguishing feature of this definition is that it is *recursive* in the sense that binary trees are constructed out of other binary trees, with the empty tree serving as the base case.

(Incidentally, a *leaf* in a binary tree is here represented as a node both of whose children are the empty tree. This definition of binary trees is analogous to starting the natural numbers with zero, rather than one. One can think of the children of a node in a binary tree as the "predecessors" of that node, the only difference compared to the usual definition of predecessor being that a node has two, rather than one, predecessors.)

To compute with a recursive type, use a recursive function. For example, here is the function to compute the *height* of a binary tree:

```
fun height Empty = 0
  | height (Node (lft, _, rht)) =
    1 + max (height lft, height rht)
```

Notice that `height` is called recursively on the children of a node, and is defined outright on the empty tree. This pattern of definition is another instance of structural induction (on the `tree` type). The function `height` is said to be defined by induction on the structure of a tree. The general idea is to define the function directly for the base cases of the recursive type (*i.e.*, value constructors with no arguments or whose arguments do not involve values of the type being defined), and to define it for non-base cases in terms of its definitions for the constituent values of that type. We will see numerous examples of this as we go along.

Here's another example. The *size* of a binary tree is the number of nodes occurring in it. Here's a straightforward definition in ML:

```
fun size Empty = 0
  | size (Node (lft, _, rht)) =
    1 + size lft + size rht
```

The function `size` is defined by structural induction on trees.

A word of warning. One reason to capitalize value constructors is to avoid a pitfall in the ML syntax that we mentioned in chapter 2. Suppose we gave the following definition of `size`:

```
fun size empty = 0
  | size (Node (lft, _, rht)) =
    1 + size lft + size rht
```

The compiler will warn us that the second clause of the definition is *redundant*! Why? Because `empty`, spelled with a lower-case "e", is a *variable*, not

a *constructor*, and hence matches *any* tree whatsoever. Consequently the second clause never applies. By capitalizing constructors we can hope to make mistakes such as these more evident, but in practice you are bound to run into this sort of mistake.

The `tree` data type is appropriate for binary trees: those for which each node has exactly two children. (Of course, either or both children might be the empty tree, so we may consider this to define the type of trees with *at most* two children; it's a matter of terminology which interpretation you prefer.) It should be obvious how to define the type of *ternary* trees, whose nodes have at most three children, and so on for other fixed arities. But what if we wished to define a type of trees with a *variable* number of children? In a so-called *variadic tree* some nodes might have three children, some might have two, and so on. This can be achieved in at least two ways. One way combines lists and trees, as follows:

```
datatype 'a tree =
  Empty |
  Node of 'a * 'a tree list
```

Each node has a *list* of children, so that distinct nodes may have different numbers of children. Notice that the empty tree is distinct from the tree with one node and no children because there is no data associated with the empty tree, whereas there is a value of type `'a` at each node.

Another approach is to simultaneously define trees and "forests". A variadic tree is either empty, or a node gathering a "forest" to form a tree; a forest is either empty or a variadic tree together with another forest. This leads to the following definition:

```
datatype 'a tree =
  Empty |
  Node of 'a * 'a forest
and 'a forest =
  None |
  Tree of 'a tree * 'a forest
```

This example illustrates the introduction of two *mutually recursive datatypes*.

Mutually recursive datatypes beget mutually recursive functions. Here's a definition of the size (number of nodes) of a variadic tree:

```
fun size_tree Empty = 0
  | size_tree (Node (_, f)) = 1 + size_forest f
and size_forest None = 0
  | size_forest (Tree (t, f')) = size_tree t + size_forest f'
```

Notice that we define the size of a tree in terms of the size of a forest, and *vice versa*, just as the type of trees is defined in terms of the type of forests.

Many other variations are possible. Suppose we wish to define a notion of binary tree in which data items are associated with branches, rather than nodes. Here's a `datatype` declaration for such trees:

```
datatype 'a tree =
    Empty |
    Node of 'a branch * 'a branch
and 'a branch =
    Branch of 'a * 'a tree
```

In contrast to our first definition of binary trees, in which the branches from a node to its children were *implicit*, we now make the branches themselves *explicit*, since data is attached to them.

For example, we can collect into a list the data items labelling the branches of such a tree using the following code:

```
fun collect Empty = nil
  | collect (Node (Branch (ld, lt), Branch (rd, rt))) =
    ld :: rd :: (collect lt) @ (collect rt)
```

10.4 Heterogeneous Data Structures

Returning to the original definition of binary trees (with data items at the nodes), observe that the *type* of the data items at the nodes must be the same for every node of the tree. For example, a value of type `int tree` has an integer at every node, and a value of type string tree has a string at every node. Therefore an expression such as

```
Node (Empty, 43, Node (Empty, "43", Empty))
```

is ill-typed. The type system insists that trees be *homogeneous* in the sense that the type of the data items is the same at every node.

It is quite rare to encounter heterogeneous data structures in real programs. For example, a dictionary with strings as keys might be represented as a binary search tree with strings at the nodes; there is no need for heterogeneity to represent such a data structure. But occasionally one might wish to work with a *heterogeneous* tree, whose data values at each node are of different types. How would one represent such a thing in ML?

To discover the answer, first think about how one might manipulate such a data structure. When accessing a node, we would need to check at run-time whether the data item is an integer or a string; otherwise we would not know whether to, say, add 1 to it, or concatenate `"1"` to the end of it. This suggests that the data item must be *labelled* with sufficient information so that we may determine the type of the item at run-time. We must also be able to recover the underlying data item itself so that familiar operations (such as addition or string concatenation) may be applied to it.

The required labelling and discrimination is neatly achieved using a datatype declaration. Suppose we wish to represent the type of integer-or-string trees. First, we define the type of values to be integers or strings, marked with a constructor indicating which:

```
datatype int_or_string =
  Int of int |
  String of string
```

Then we define the type of interest as follows:

```
type int_or_string_tree =
  int_or_string tree
```

Voila! Perfectly natural and easy — heterogeneity is really a special case of homogeneity!

10.5 Abstract Syntax

`Datatype` declarations and pattern matching are extremely useful for defining and manipulating the *abstract syntax* of a language. For example, we may define a small language of arithmetic expressions using the following declaration:

```
datatype expr =
  Numeral of int |
  Plus of expr * expr |
  Times of expr * expr
```

This definition has only three clauses, but one could readily imagine adding others. Here is the definition of a function to evaluate expressions of the language of arithmetic expressions written using pattern matching:

```
fun eval (Numeral n) = Numeral n
  | eval (Plus (e1, e2)) =
    let
        val Numeral n1 = eval e1
        val Numeral n2 = eval e2
    in
        Numeral (n1+n2)
    end
  | eval (Times (e1, e2)) =
    let
        val Numeral n1 = eval e1
        val Numeral n2 = eval e2
    in
        Numeral (n1*n2)
    end
```

The combination of `datatype` declarations and pattern matching contributes enormously to the readability of programs written in ML. A less obvious, but more important, benefit is the error checking that the compiler can perform for you if you use these mechanisms in tandem. As an example, suppose that we extend the type `expr` with a new component for the reciprocal of a number, yielding the following revised definition:

```
datatype expr =
  Numeral of int |
  Plus of expr * expr |
  Times of expr * expr |
  Recip of expr
```

First, observe that the "old" definition of `eval` is no longer applicable to values of type `expr`! For example, the expression

```
eval (Plus (Numeral 1, Numeral 2))
```

is ill-typed, even though it doesn't use the `Recip` constructor. The reason is that the re-declaration of `expr` introduces a "new" type that just happens to have the same name as the "old" type, but is in fact distinct from it. This is a boon because it reminds us to recompile the old code relative to the new definition of the `expr` type.

Second, upon recompiling the definition of `eval` we encounter an *inexhaustive match* warning: the old code no longer applies to every value of type `expr` according to its new definition! We are of course lacking a case for `Recip`, which we may provide as follows:

```
fun eval (Numeral n) = Numeral n
  | eval (Plus (e1, e2)) = ... as before ...
  | eval (Times (e1, e2)) = ... as before ...
  | eval (Recip e) =
    let
        val Numeral n = eval e
    in
        Numeral (1 div n)
    end
```

The value of the checks provided by the compiler in such cases cannot be overestimated. When recompiling a large program after making a change to a `datatype` declaration the compiler will automatically point out *every line of code* that must be changed to conform to the new definition; it is impossible to forget to attend to even a single case. This is a tremendous help to the developer, especially if she is not the original author of the code being modified and is another reason why the static type discipline of ML is a positive benefit, rather than a hindrance, to programmers.

10.6 Sample Code

Here is the code for this chapter.