Chapter 11
A Fourth Look at ML

11.1 Introduction

You may recall from Chapter 6 that the type `bool` is not primitive in ML. Although it is a predefined type, it is actually constructed with an ML definition—*this* ML definition:

```
datatype bool = true | false;
```

Similarly, the `list` type constructor is not primitive—it is predefined, but its ML definition is simply this:

```
datatype 'element list = nil |
   :: of 'element * 'element list;
```

You can write your own `datatype` definitions in ML, defining types as simple as `bool` (which is really just an enumeration) or as complicated as polymorphic tree structures. Defining new ML types is the subject of this chapter. This is also the last chapter on ML, so its conclusion gives an overview of the parts of ML that weren't covered.

11.2 Enumerations

One of the simplest uses of the datatype definition in ML is to create an enumerated type. Here is an example:

```
- datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun;
datatype day = Fri | Mon | Sat | Sun | Thu | Tue | Wed
- fun isWeekDay x = not (x = Sat orelse x = Sun);
val isWeekDay = fn : day -> bool
- isWeekDay Mon;
val it = true : bool
- isWeekDay Sat;
val it = false : bool
```

This example defined an enumerated type called day and its members, Mon through Sun. The type was then used in a function isWeekDay. Notice that ML's usual type inference applies to new types as well; ML decided that the domain type for isWeekDay must be day, since the function compares its input for equality with the day values Sat and Sun.

The name day in this example is called a *type constructor* and the member names are called *data constructors*. (This terminology will make more sense in later examples, when you see how type constructors and data constructors can both have parameters.) A common ML style convention is to capitalize the names of data constructors, as was done in the example.

Here is another example of an enumeration:

```
- datatype flip = Heads | Tails;
datatype flip = Heads | Tails
- fun isHeads x = (x = Heads);
val isHeads = fn : flip -> bool
- isHeads Tails;
val it = false : bool
- isHeads Mon;
Error: operator and operand don't agree [tycon mismatch]
  operator domain: flip
  operand:         day
```

As you can see, ML is strict about these new types—it is not allowing us to pass a value of type day to a function that expects a value of type flip. In some languages, the underlying representation the language system uses for enumerations is visible to the programmer. In C, for example, an enum definition creates named constants with integer values, so any value of an enumeration can be used just like an integer. ML does not expose the underlying representation. It may be using small integers to represent the different day values, and it may be using the same small integers to represent the different flip values, but there is no way to tell. ML does not permit any operations that would allow the program to detect the representation. The only operations permitted are comparisons for equality (either explicitly or implicitly through the use of patterns).

You can use the data constructors in patterns. For example, we could, and probably should, rewrite the isWeekday function using a pattern-matching style:

```
fun isWeekDay Sat = false
|   isWeekDay Sun = false
|   isWeekDay _ = true;
```

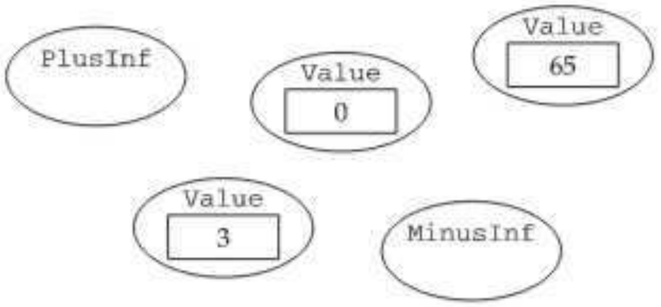
In this simple case the data constructors work just like constants in a pattern.

11.3 Data Constructors with Parameters

You can add a parameter to a data constructor by adding the keyword of followed by the type of the parameter. Here, for example, is a datatype definition for a type exint with three data constructors, one of which has a parameter:

```
datatype exint = Value of int | PlusInf | MinusInf;
```

The exint type includes three different kinds of things: Value, PlusInf, and MinusInf. By declaring the Value data constructor as Value of int, we are saying that each Value will contain an int, which is to be given as a parameter to the data constructor: Value 3, Value 65, and so on. For each different int there is a different possible Value. A Value is like a wrapper that contains an int, while a PlusInf or a MinusInf is like an empty wrapper. This illustration shows some possible things of type exint.



You might use the exint type as an extended kind of integer that includes representations for all the int values, plus special representations for positive and negative infinities.

If you look at how ML reports the types of the data constructors, you will see that while PlusInf and MinusInf look like constants of type exint, Value is a function that takes an int and returns an exint.

```
- PlusInf;
val it = PlusInf : exint
- MinusInf;
val it = MinusInf : exint
- Value;
val it = fn : int -> exint
- Value 3;
val it = Value 3 : exint
```

Although each Value contains an int, it cannot be treated as an int. For example, if we try to add one Value to another, we get an error:

```
- val x = Value 5;
val x = Value 5 : exint
- x + x;
Error: overloaded variable not defined at type
  symbol: +
  type: exint
```

So how can we get the wrapped-up int out of a Value? It's simple—by pattern matching. We can treat (Value x) as a pattern that not only matches a Value, but also binds the variable x to the integer inside it. Remember that pattern matching is used in many different places in ML: in val definitions, in case expressions, and most commonly in pattern-matching function definitions. Here we use a pattern in the val definition to extract the integer from the previous variable x:

```
- val (Value y) = x;
val y = 5 : int
```

That val definition used a nonexhaustive pattern; it worked only because the variable x was bound to a Value, not a PlusInf or a MinusInf. In other contexts we might need to use an exhaustive pattern to cover all three possibilities for an exint. For example, this val definition uses an *exhaustive* pattern as part of a case expression to convert each exint into a string:

```
- val s = case x of
=           PlusInf => "infinity" |
=           MinusInf => "-infinity" |
=           Value y => Int.toString y;
val s = "5" : string
```

Int.toString is a predefined function that takes an int parameter and returns the corresponding string.

This pattern-matching function definition defines a function `square` that returns an `exint` representing the square of an `exint` input.

```
- fun square PlusInf = PlusInf
=  |    square MinusInf = PlusInf
=  |    square (Value x) = Value (x * x);
val square = fn : exint -> exint
- square MinusInf;
val it = PlusInf : exint
- square (Value 3);
val it = Value 9 : exint
```

Pattern-matching function definitions are especially important when you are working with your own `datatype` definitions, since pattern matching is the only way to extract the values that were passed to the data constructors.

Incidentally, one more use of pattern matching in ML is for exception handling. This book will not cover ML's exception handling (although it will look at Java's exception handling later on), but here is one example of it just to whet your appetite. This example extends the previous function so that it handles integer overflow on the multiplication by returning a `PlusInf`.

```
- fun square PlusInf = PlusInf
=  |    square MinusInf = PlusInf
=  |    square (Value x) = Value (x * x)
=         handle Overflow => PlusInf;
val square = fn : exint -> exint
- square (Value 10000);
val it = Value 100000000 : exint
- square (Value 100000);
val it = PlusInf : exint
```

11.4 Type Constructors with Parameters

The type constructor for a `datatype` definition can have parameters too. The parameters for a type constructor are type parameters, and the result is a polymorphic type constructor. For example, here is the `option` type constructor, which is predefined in ML:

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

The type constructor is named `option`. It takes a type `'a` as a parameter. The data constructors are `NONE` and `SOME`. The `SOME` constructor takes a parameter of type `'a`.

It seems odd at first that the parameter 'a is placed before the type constructor name, but that is how it will be used. Just as the list type constructor makes types like int list and real list, the option type constructor will make types like int option and real option.

```
- SOME 4;
val it = SOME 4 : int option
- SOME 1.2;
val it = SOME 1.2 : real option
- SOME "pig";
val it = SOME "pig" : string option
```

The option type constructor is useful for functions whose result is not always defined. For example, the result of an integer division is not defined when the divisor is zero, so we might write:

```
- fun optdiv a b =
=   if b = 0 then NONE else SOME (a div b);
val optdiv = fn : int -> int -> int option
- optdiv 7 2;
val it = SOME 3 : int option
- optdiv 7 0;
val it = NONE : int option
```

Many predefined ML functions use option. You can use it in your own functions too.

Here is a longer example of a polymorphic datatype definition. It defines a type constructor called bunch so that a value of type 'x bunch is either a single element of type 'x or a list of elements of type 'x.

```
datatype 'x bunch =
   One of 'x |
   Group of 'x list;
```

As usual, ML figures out the actual type of any bunch value.

```
- One 1.0;
val it = One 1.0 : real bunch
- Group [true,false];
val it = Group [true,false] : bool bunch
```

Although ML can figure out the type of a bunch value, it does not have to resolve the type of a bunch in all cases. Polymorphic functions like size, below, can work on values of any bunch type.

```
- fun size (One _) = 1
= |   size (Group x) = length x;
val size = fn : 'a bunch -> int
- size (One 1.0);
val it = 1 : int
- size (Group [true,false]);
val it = 2 : int
```

Here is an example of a use of bunch that does force ML to resolve the type:

```
- fun sum (One x) = x
= |   sum (Group xlist) = foldr op + 0 xlist;
val sum = fn : int bunch -> int
- sum (One 5);
val it = 5 : int
- sum (Group [1,2,3]);
val it = 6 : int
```

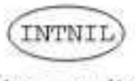
Because the + operator (through foldr) is applied to the list elements, ML knows that the type of the parameter to sum must be int bunch.

11.5 Recursively Defined Type Constructors

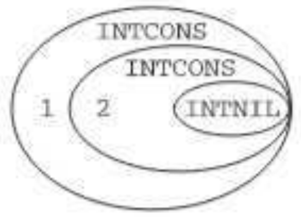
In the data constructors of a datatype definition, you can use the type constructor being defined. This kind of recursive definition is a source of expressive power in ML programs. Here, for example, is a definition of a type constructor called intlist, which is almost like the ML type int list.

```
datatype intlist =
  INTNIL |
  INTCONS of int * intlist;
```

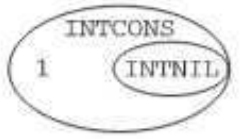
Notice the recursion. The second element of the pair required by the INTCONS data constructor is of the type being defined—intlist. This new type works just like a plain int list, but with INTNIL in place of nil and with the INTCONS constructor in place of the :: operator. This illustration shows some possible values of type intlist.



the empty list



the list [1,2]



the list [1]

This ML session constructs the values pictured above:

```
- INTNIL;
val it = INTNIL : intlist
- INTCONS (1,INTNIL);
val it = INTCONS (1,INTNIL) : intlist
- INTCONS (1,INTCONS(2,INTNIL));
val it = INTCONS (1,INTCONS (2,INTNIL)) : intlist
```

Of course, ML does not display those values in the standard list notation; you would have to write your own function to do that. The function `intlistLength`, below, computes the length of an `intlist`. Compare this with the function `listLength` that follows it, which computes the length of an ordinary `int list` using the same method. See how similar the two types are?

```
fun intlistLength INTNIL = 0
  | intlistLength (INTCONS(_,tail)) =
      1 + (intListLength tail);

fun listLength nil = 0
  | listLength (_::tail) =
      1 + (listLength tail);
```

Of course, there is one major difference between the predefined `list` type constructor and `intlist`: the predefined `list` is parametric, while `intlist` works only for lists of integers. This is easily remedied. The following datatype, `mylist`, is just like `intlist`, but parametric. Compare the two definitions and you will see that the only change was to replace the `int` type with a type variable, added as a parameter for the type constructor.

```
datatype 'element mylist =
  NIL |
  CONS of 'element * 'element mylist;
```

This is now a parametric list type constructor, just like the predefined `list`. We can make a `real mylist` or an `int mylist`, and ML's type inference handles it in the usual way.

```
- CONS(1.0, NIL);
val it = CONS (1.0,NIL) : real mylist
- CONS(1, CONS(2, NIL));
val it = CONS (1,CONS (2,NIL)) : int mylist
```

Almost no change is necessary to the length-computing function to make it handle this new polymorphic list. We just have to use the new data constructors `CONS` and `NIL`.

```
fun myListLength NIL = 0
  | myListLength (CONS(_,tail)) =
      1 + myListLength(tail);
```

Almost anything you can do with the predefined `list` types you can now do with a `mylist` type. Here is a function to add up the elements of an `int mylist`. This is not polymorphic, since the use of the + operator forces ML to nail down the element type.

```
fun addup NIL = 0
  | addup (CONS(head,tail)) =
      head + addup tail;
```

With predefined `list` types, we would not bother to write a function like `addup`. As was shown in Chapter 9, it is easier to use `foldr` to add up an `int list x` just by writing `foldr op + 0 x`. Here is an implementation of `foldr` for the `mylist` type:

```
fun myfoldr f c NIL = c
  | myfoldr f c (CONS(a,b)) =
      f(a, myfoldr f c b);
```

We can now add up an `int mylist x` just by writing `myfoldr op + 0 x`.

In short, the type constructor `mylist` works just like the predefined type constructor `list`. Take any function that works on `list` types, replace `nil` with `NIL` and `::` with `CONS`, and you have a function that works on `mylist` types. Of course, you would not really want to replace the predefined list types with your own list types. The point of these examples is to demonstrate the use of `datatype` to make polymorphic type constructors and to emphasize that there is no primitive magic to the predefined list types. They can be defined using ordinary ML constructs.

Actually, there is still one minor difference between the `mylist` type constructor and the predefined `list` type constructor: the predefined `::` function is an operator, while `CONS` is not. This is trivial to fix. How new ML operators are defined won't be shown in any detail, but here is an example to whet your appetite:

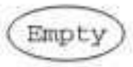
```
- infixr 5 CONS;
infixr 5 CONS
- 1 CONS 2 CONS NIL;
val it = 1 CONS 2 CONS NIL : int mylist
```

Now you can write `1 CONS 2 CONS NIL` just as you can write `1::2::nil` using the predefined list types. (The `infixr` statement declares that the function in question should now be treated as a right-associative binary operator. The 5 in the example is the precedence level—in ML, this is the same precedence level as the `::` operator.)

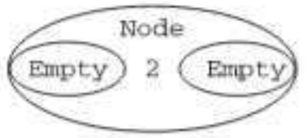
The `mylist` type constructor shows how a recursively defined `datatype` can be used for polymorphic lists. The same trick can be used for polymorphic binary trees. Here is the `datatype` definition:

```
datatype 'data tree =
   Empty |
   Node of 'data tree * 'data * 'data tree;
```

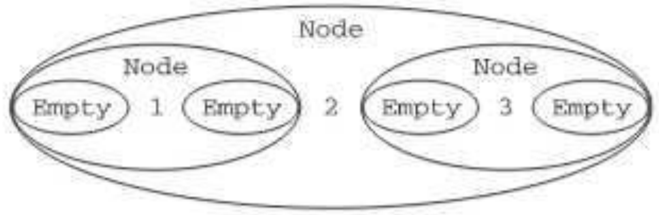
In this definition, a tree can be empty or can consist of a left child, a data element, and a right child. The type variable `'data` is the type of the element stored in the node. This illustration shows some possible values of type `int tree`:



the empty tree



the tree 2



the tree 2
1 3

This ML session constructs the values pictured above:

```
- val treeEmpty = Empty;
val treeEmpty = Empty : 'a tree
- val tree2 = Node(Empty,2,Empty);
val tree2 = Node (Empty,2,Empty) : int tree
- val tree123 = Node(Node(Empty,1,Empty),
=                    2,
=                    Node(Empty,3,Empty));
```

Here are some examples of functions that operate on these trees. This one takes an `int tree` parameter and returns a similar tree, but with 1 added to each integer:

```
fun incall Empty = Empty
  | incall (Node(x,y,z)) =
      Node(incall x, y+1, incall z);
```

For example, applying it to the previous value `tree123` produces this result:

```
- incall tree123;
val it = Node (Node (Empty,2,Empty),
               3,
               Node (Empty,4,Empty)) : int tree
```

This next function adds up all the numbers in an `int tree`:

```
fun sumall Empty = 0
  | sumall (Node(x,y,z)) =
      sumall x + y + sumall z;
```

Applying it to the value `tree123` produces this result:

```
- sumall tree123;
val it = 6 : int
```

The previous functions `incall` and `sumall` are not polymorphic, since they work only on `int tree` values. This polymorphic function takes any type of tree and produces a list of its values:

```
fun listall Empty = nil
  | listall (Node(x,y,z)) =
      listall x @ y :: listall z;
```

For example, applying it to the value `tree123` produces this result:

```
- listall tree123;
val it = [1,2,3] : int list
```

One more polymorphic example: this is an exhaustive search of any type of tree for a given value:

```
fun isintree x Empty = false
  | isintree x (Node(left,y,right)) =
       x = y
       orelse isintree x left
       orelse isintree x right;
```

Using it to search the value `tree123` produces these results:

```
- isintree 4 tree123;
val it = false : bool
- isintree 3 tree123;
val it = true : bool
```

Note that the `isintree` function makes no assumptions about the order in which elements appear in the tree; it searches the whole tree until the element is found. For ordered trees (such as binary search trees) you can do better—can and will, if you do Exercise 12.

11.6 Conclusion—Farewell to ML

This chapter discussed the `datatype` definition in ML. It explored some examples showing simple enumerations, data constructors with parameters, type constructors with parameters, and recursively defined types for lists and trees. This is the end of the introduction to ML.

At this point, which of these describes you best?

A. I just couldn't get comfortable with ML. I hope I never have to write another piece of code in a functional language.

B. I really liked ML and I want to do more.

Did you answer A? Don't worry; you are not alone. Learning a little bit of a language like ML is an interesting and mind-expanding exercise, but not everyone feels comfortable with ML programming. The next language is Java. Perhaps that will suit you better.

Did you answer B? Then you may be interested in some of the parts of ML that were skipped. Here is a list of some of them:

- Records—ML supports records, which are handled a bit like tuples except that the elements are named instead of numbered.
- Arrays—ML supports arrays, with elements that can be altered. This is not in the purely functional part of ML, but for some applications arrays are just too useful to leave out.

- References—Again, this is an imperative part of ML. References can be used to get the effect of assignable variables.
- Exception handling—ML uses pattern matching in its exception handling mechanism.
- Encapsulation—For larger programs, you want a way to build separate modules whose parts can be hidden from each other. This is a big part of ML (and of any modern language), though these small examples and exercises did not require it. ML supports *structures*, which are collections of datatypes, functions, and so on; *signatures*, which give a way of describing the interface of a structure, separate from its implementation; and *functors*, which are like functions that operate on whole structures.
- API—As always, the language libraries dwarf the language itself. The *standard basis* defines the functions (not to mention type constructors, structures, signatures, functors, and so on) that are predefined in standard ML environments. Some of these are present in the *top-level* environment and are referred to by their simple names, like `concat` and `explode`. Most, however, are in structures and are referred to by their full names, like `Int.maxInt`, `Real.Math.sqrt`, and `List.nth`. Many other popular libraries are not part of the standard basis. For example, a library called eXene is for developing ML applications that work in X Window System.
- Compilation Manager—A system for managing the many files that make up large ML applications. This serves some of the same purposes as the Unix `make` facility.

This book has used Standard ML, but there are several other dialects of ML. For example, OCaml is a dialect that adds class-based, object-oriented features, and OCaml in turn forms the basis of F#, a multi-paradigm language developed at Microsoft Research for the .NET platform.

ML supports a function-oriented style of programming. It favors a problem-solving approach that composes solutions to algorithmic problems using many small (usually side-effect-free) functions. That same general style is supported by other functional languages as well. If you find that this style suits you, you might want to look at other functional languages, like Lisp and Haskell. The "Further Reading" section at the end of this chapter includes a reference for each of them.

Exercises

Exercise 1 Write a `datatype` definition for a type `suit` whose values are the four suits of a deck of playing cards.

Exercise 2 Using your definition from Exercise 1, write a function `suitname` of type `suit -> string` that returns a `string` giving the name of a `suit`.

Exercise 3 Write a `datatype` definition for a type `number` whose values are either integers or real numbers.

Exercise 4 Using your definition from Exercise 3, write a function `plus` of type `number -> number -> number` that adds two numbers, coercing `int` to `real` only if necessary.

Exercise 5 Write a function addup of type `intnest -> int` that adds up all the integers in an `intnest`. Use this definition for `intnest`. (Be careful as you type. `INT` is not the same as `int`!)

```
datatype intnest =
  INT of int |
  LIST of intnest list;
```

Exercise 6 Write a function `prod` of type `int mylist -> int` that takes an `int mylist x` and returns the product of all the elements of `x`. If the list is `NIL` your function should return 1. Here again is the definition of `mylist`, as seen earlier in this chapter:

```
datatype 'element mylist =
  NIL |
  CONS of 'element * 'element mylist;
```

Exercise 7 Write a function `reverse` of type `'a mylist -> 'a mylist` that takes a `mylist a` and returns a `mylist` of all the elements of `a`, in reverse order. (Use the `mylist` definition from Exercise 6.)

Exercise 8 Write a function `append` of type `'a mylist -> 'a mylist -> 'a mylist` that takes two `mylist` values, `a` and `b`, and returns the `mylist` containing all the elements of `a` followed by all the elements of `b`. (Use the `mylist` definition from Exercise 6.)

Exercise 9 Write a function `appendall` of type `'a list tree -> 'a list` that takes a tree of lists and returns the result of appending all the lists together. Put the list for a node together in this order: first the contents of the left subtree, then the list at this node, and then the contents of the right subtree. Here again is the definition of `tree`, as seen earlier in this chapter:

```
datatype 'data tree =
   Empty |
   Node of 'data tree * 'data * 'data tree;
```

Exercise 10 A complete binary `tree` is one in which every `Node` has either two `Empty` children or two `Node` children, but not one of each. Write a function `isComplete` of type `'a tree -> bool` that tests whether a `tree` is complete. (Use the `tree` definition from Exercise 9.)

Exercise 11 A binary search tree is a binary tree with special properties. It may be `Empty`. It may be a `Node` containing a left subtree, a data item `x`, and a right subtree. In this case all the data items in the tree are different, all the items in the left subtree are smaller than `x`, all the items in the right subtree are greater than `x`, and the left and right subtrees are also binary search trees. Write a function `makeBST` of type `'a list -> ('a * 'a -> bool) -> 'a tree` that organizes the items in the list into a binary search tree. The tree need not be balanced. You may assume that no item in the list is repeated.

Exercise 12 Write a function `searchBST` of type `''a tree -> (''a * ''a -> bool) -> ''a -> bool` that searches a binary search tree for a given data element. (Refer to Exercise 11 for the definition of a binary search tree.) You should not search every node in the tree, but only those nodes that, according to the definition, might contain the element you are looking for.

Further Reading

This excellent introductory book on ML has already been mentioned:

> Ullman, Jeffrey D. *Elements of ML Programming*. Upper Saddle River, NJ: Prentice Hall, 1998.

This ML book is somewhat more advanced:

> Paulson, Larry C. *ML for the Working Programmer.* New York: Cambridge University Press, 1996.

If you liked ML, you may also be interested in other functional languages. Two important ones are Lisp and Haskell. Lisp is the oldest of the functional languages and is still widely used for artificial-intelligence research. This book does not focus on AI, but gives a variety of examples of Lisp code:

Graham, Paul. *ANSI Common Lisp*. Upper Saddle River, NJ: Prentice Hall, 1995.

Haskell is a modern functional language that differs from ML in several important ways. One major difference is that Haskell is a *lazy* language; it evaluates only as much of the program as necessary to get the answer.

Thompson, Simon. *Haskell: The Craft of Functional Programming*. Boston, MA: Addison-Wesley, 1999.