Lists

Michael P. Fourman

February 2, 2010

1 Introduction

The *list* is a fundamental datatype in most functional languages. ML is no exception; `list` is a built-in ML *type constructor.* However, to introduce the idea of a list, we will, first, show how to define a type of lists of integers as a new type. The following datatype declaration

```
datatype intlist = nil
                 | :: of int * intlist;
```

introduces a new type, `intlist`, and constructors

```
nil : intlist
::  : int * intlist -> intlist
```

just as in our earlier examples of datatype declarations, but it introduces a new idea. This datatype declaration is *recursive:* `nil` is a constant of type intlist, and `::` allows us to construct a new list from a pair consisting of an integer and an existing list. The constructor `::` is infix[1], and associates to the right (so `2::3::nil` is read as `2::(3::nil)`). We can use the constructors to build lists of integers:

```
> 3 :: nil;
val it = 3 :: nil : intlist
> 2 :: it;
val it = 2 :: 3 :: nil : intlist
> 1 :: it;
val it = 1 :: 2 :: 3 :: nil : intlist
```

Notice the order in which we add elements to the list: last first.

[1]We pronounce `::` as, *cons*, for historical reasons: in LISP, the original functional language, it is the fundamental *cons*tructor.

Every value, `v`, of type `intlist` must be constructed according to one of the clauses in the datatype declaration: either it is the list `nil`, or it was constructed as `(h :: t)` from an integer `h` and a list `t`, in the latter case, we call `h` the *head,* and `t` the *tail,* of `v`.

```
> val (h :: t) = 1 :: 2 :: 3 :: nil;
val h = 1 : int   val t = 2 :: 3 :: nil : intlist
```

We can use the constructors in patterns to define functions on lists:

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

This function counts the members of a list:

```
> length(4 :: 5 :: 6 :: nil);
val it = 3 : int
```

Most functions on lists follow the example of `length`; the function definition has two clauses matching those of the datatype declaration. For example, here is a function to sum the elements of a list

```
fun sum nil      = 0
  | sum (h :: t) = h + sum t;
```

2 Lists in ML

The introduction has given examples of the structure of lists in ML. By simply omitting the datatype declaration, all the examples given can be run using the built-in lists[2]. However, the built-in lists have two advantages over our *ad-hoc* declaration.

First, the syntax `1::2::3::nil` is cumbersome for such a simple object – especially since we will use lists so frequently. The built-in lists provide the constructors `nil` and `::`, as above, but they also provide an alternative syntax, for example, the alternative syntax for `1::2::3::nil` is `[1,2,3]`. This alternative syntax can also be used in patterns:

```
fun lastPair [a,b]    = (a,b)
  | lastPair (_ :: t) = lastPair t;
```

In this, artificial, example, we return a pair composed of the last two elements of the argument list. Notice the difference between the two patterns; the first

[2]The declaration of `intlist` introduces new constructors `nil` and `::`. These hide the built-in versions with the same names. So to use the built-in lists you should start a new ML session.

matches lists with exactly two elements, the second matches any list with one or more elements. The compiler generates a warning, because our patterns don't cover all possibilities; the empty list, `[]`, or `nil`, is not covered. (What happens if we call `lastPair` with a singleton list `[x]` as argument?)

Second, our declaration only provides for lists of integers. The built-in lists provide for lists of any type of value, but the members of any one list must all have the same type. Here are some examples:

```
[1,2,3]          : int list
["fred", "joe"] : string list
[]               : 'a list
```

If T is a type, then T `list` is a type. This is why we call `list` a type constructor; it constructs new types out of old. Notice the type of the empty list `[]:'a list`. Here, `'a` is a *type variable;* it can be replaced by any type. This is our first example of *polymorphism,* a powerful feature of ML's type system. Later, we will see how to make datatype declarations that introduce polymorphic type constructors, such as `list`. Now, we give an example, to show that polymorphism allows us to write general-purpose functions. Here is the definition of the `length` function again:

```
fun length []       = 0
  | length (_ :: t) = 1 + length t;
```

the compiler responds, as usual, by giving the type of the function

```
val length = fn : 'a list -> int
```

The type includes a type variable; the function `length` is polymorphic. This means we can apply the same function to lists of integers, lists of strings, lists of lists, and so on. Because the compiler infers the type, you don't need to worry about the details of polymorphism. You get the benefits, of type safety, and of generic code that can be applied to many different types of argument, without having to provide type annotations for the compiler to check.

As we have said, we can build lists of any type, but each element of the list must be of the *same* type, i.e. the lists must be homogeneous. This seems like quite a restriction, but it is necessary if we want static typechecking. In cases where we want non-homogenous lists, we have to declare a datatype, to form a single type that is the sum of the various types we wish to include.

Built-in functions on lists

`@` — **append** Two lists may be joined using the infix function append, `@`. For example, the expression `[1,2,3] @ [4,5]` evaluates to `[1,2,3,4,5]`. Although the append function is predefined, we could easily define it ourselves:

```
fun       [] @ l = l
  | (h :: t) @ l = h :: (t @ l);
```

`rev` — **list reversal** The predefined function `rev` reverses its argument; `rev[1,2,3]` evaluates to `[3,2,1]`. Again, we could define `rev` ourselves, had it not been provided. Implementing `rev` provides an instructive example.

One approach to reversing a non-empty list is to reverse the tail and then to add the head to the end of the result. To reverse the tail is simple, we just use a recursive call of the function. How can we add the head to the end of the result? The append function can be used, but this joins two lists together, not a list and an element of the list. However, we can make the head into a singleton list (`[h]` in the above example), and then `@` may be used to append it to the end of the reversed tail. A simple-minded version of the function can be written as:

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

This implementation of `rev` is not very efficient. A calculation shows that the time to reverse a list is $O(n^2)$ when n is the length of the list. The following function can reverse the list in $O(n)$ time, a significant difference if the list is long.

```
local
  fun revto   ([], rl) = rl
    | revto (h::t, rl) = revto(t,  h::rl)
in
  fun rev l = revto(l, [])
end;
```

`map` — **applying a function to each member of a list** Consider the following function, which takes the square root of each member of a list.

```
fun mapsqrt []       = []
  | mapsqrt (h :: t) = (sqrt h) :: (mapsqrt t);
```

It follows our standard pattern for list recursion. If we wanted to apply a different function (say, `square`) to each element of a list, we could

use this declaration as a template, change the name of the function (say to `mapsquare`), and replace the call to `sqrt` by a call to `square`. Doing this occasionally would not be taxing. However, we use this pattern of computation frequently in functional programming, so ML provides a built-in function `map` to build a function that acts on lists, from a function that acts on members

```
map square [1,2,3];    (* should return [1,4,9]    *)
map sqrt   [4.0, 9.0]; (* should return [2.0, 3.0] *)
```

Function application associates to the left, so `map square [1,2,3]` is read as

`(map square) [1,2,3]`.

The function `map` takes the function `square: int -> int` as argument, and returns a function `(map square): int list -> int list` as result. Functions that take functions as arguments, return them as results, or do both, are known as *higher order functions,* or, *functionals.* They are useful because they allow us to package up commonly occurring patterns of usage.

We will defer giving our own implementation of `map` "from scratch" until later.

3 Anonymous and curried functions

Suppose we want a function, `inclist`, to add 1 to each element of a list. We could declare a function to add one to an integer, and then use the built-in function `map`

```
fun inc x = 1 + x;
fun inclist xs = map inc xs;
```

However, this seems heavy-handed — and making the declaration of `inc` local would make the example even more baroque. In this section we introduce two neater ways of introducing the function.

Anonymous Functions

We want a simple function, that given x returns $x+1$. ML allows us to write an expression, "`fn x => x+1`," whose value is that function, without even bothering to give it a name. So we could write our example as

```
fun inclist xs = map (fn x => x+1) xs;
```

Here `fn` is a keyword, introducing an anonymous function expression; `x` is a pattern, introducing the formal parameter, and `x+1` is the *body*, which is evaluated when the function is applied. Patterns, and alternative clauses, can

be used in anonymous functions, just as they are in `case` expressions. Since the function has no name, it can't refer to itself; so, there are no recursive anonymous functions. Anonymous functions should only be used for fairly short functions that are passed as parameters to other functions. If a function is likely to be used in a number of different places, or it is easy to associate an informative name with the function, then a conventional declaration is to be preferred.

The idea of anonymous functions forms the basis of the λ-calculus, an elegant mathematical model of computation, discovered by Haskell Curry. The λ-calculus notation for our function is $\lambda x.x + 1$, with λ in place of ML's `fn`, and a `.` instead of `=>`.

Curried functions

To introduce the next idea, we write a rather odd version of the integer addition function.

```
fun plus (y:int) = fn x => x + y;
```

This is a function that takes an integer argument, `y` and returns an anonymous function. For example, `plus 1` is the function `fn x => x+1`. This is just the increment function! So we could write

```
fun inclist xs = map (plus 1) xs;
```

When we apply `plus` to an integer, the type of the returned function is `int -> int`, so the type of `plus` is `int -> (int -> int)`. Contrast this with the type of `+` on integers which is `(int*int) -> int`. The function `plus` is an example of a *curried* function (named after Curry, of the lambda-calculus) that takes its arguments one at a time, rather than all at once, packaged as a tuple.

We don't have to apply `plus` to both of its arguments immediately. We can apply it to the first argument, and then use the result in a variety of contexts, applying it to a range of different 'second arguments'. Instead of defining the increment function explicitly we can just type

```
val inc = plus 1;
```

Note the use of a `val` declaration here. Although we are defining a function, we are not supplying a pattern and a body, and so we must treat it just like any other value declaration. Applying a curried function to only some of its arguments is known as partial application. The built-in function `map` is also curried. We could define `inclist` by partially applying `map`

```
val inclist = map (plus 1);
```

To add two numbers together using `plus` we must first apply it to one integer, this returns a function which we then apply to the second integer; in `(plus 3) 2`, the expression `(plus 3)` denotes a function, which adds 3 to its argument. Since application associates to the left we can drop the parentheses, just writing `plus 3 2`. SML allows us to use this form as an extension of the pattern-matching syntax for function declarations. This provides us with a convenient syntax for defining curried functions. The `plus` example can be defined without using an anonymous function as follows:

```
fun plus x y = x + y;
```

It may appear that we haven't saved much by defining `inc` in terms of `plus`, and our discussion of the example has certainly been long-winded. However, defining a new function by partially applying a curried function to only some of its arguments is a powerful, but subtle, technique. It is well worthwhile spending some time studying a simple example.

Implementing `map`

Using the techniques introduced above, it is simple to develop an implementation of `map`. We take the implementation of `mapsqrt` as a template

```
fun mapsqrt []       = []
  | mapsqrt (h :: t) = (sqrt h) :: (mapsqrt t);
```

To generalise from this example, we introduce an extra parameter, `f`, and use this in place of `sqrt` in the body of the declaration. Of course, we also have to pass the extra parameter in to the recursive calls.

```
fun map f []       = []
  | map f (h :: t) = (f h) :: (map f t);
```

4 Sets using Lists

In this section, we see how lists can be used to represent sets. Many datastructures are designed to represent sets of items, and variations on this theme will form a substantial part of the course. Later, we will often consider sets of records, each consisting of an integer *key*, and some associated data. We will be interested in storing large numbers of such records, and efficiently finding the data associated with a given key. To maintain the set of records, we will need to implement operations on sets.

In this section we give a simple implementation of some basic operations on sets. To make this example simple, we consider finite sets of integers.

We begin by specifying an interface, to define the operations we wish to implement.

```
signature SetSig =
sig
    type Item
    type Set

    val empty     : Set
    val isEmpty   : Set -> bool
    val member    : Set -> Item -> bool
    val insert    : Item * Set  -> Set
    val delete    : Item * Set  -> Set
    val union     : Set  * Set  -> Set
    val intersect : Set  * Set  -> Set
end;
```

These operations correspond to the familiar operations of set-theory: there is an empty set, we can check whether an item is a member of a set, insert and delete elements, and take unions and intersections of sets. Some operations, for example, set complement, have been omitted intentionally: we only intend to represent finite sets (at least, for this example). In this section we show how to define such functions using unordered lists, partly to give further examples of functions on lists, and partly to motivate material that will be covered later in the course. The resulting functions are not particularly efficient, mainly because the elements of the list are not ordered, but also because of the bottlenecks introduced by the choice of a list for the representation. We delay discussing more efficient representations until later.

The representation we have in mind is straight-forward: a list will represent the set of its elements. The intention is to model sets by lists containing no duplicates. Figure 1 gives an implementation of our specification. The way we have implemented the function `delete` relies on the fact that the list representing a set contains no duplicates; to delete `e` we remove the first occurrence of `e` in the list, if there are no duplicates, this is enough.

Unfortunately we cannot prove that properties such as

```
member(e, delete(e, l)) = false
```

hold for any list `l` because some lists will have duplicates and `delete` will do the wrong thing in this case. There are two solutions to this problem. We could redefine `delete` to remove all occurrences of an item. However, if we know that the argument contains no duplicates this is inefficient. Instead,

```
structure IntSet : SetSig =
struct
    type Item = int
    type Set  = Item list
    exception Select

    val empty = []
    fun isEmpty [] = true
      | isEmpty  _ = false

    fun member  []      e  = false
      | member (h :: t) e  = (e = h) orelse member t e;

    fun insert(e, s)       = if member s e then s
                             else e :: s;

    fun delete(e, [])      = []
      | delete(e, h::t)    = if e = h then t
                             else h :: (delete(e, t));

    fun select []          = raise Select
      | select (h :: t)    = (h, t)

    fun union([]  , s)     = s
      | union(h::t, s)     = insert(h, union(t, s));

    fun intersect([],   s) = []
      | intersect(h::t, s) =
          if member s h then h :: intersect(t, s)
                        else      intersect(t, s);
end;
```

Figure 1: an Implementation of Sets

we make sure that the functions that return sets ensure this property. If we only build our lists representing sets using the set functions defined above, they will contain no duplicates. This approach is risky, because we have no control over the ways in which lists may be built. Ways of enforcing the constraint that sets are built using only the functions given above will be pursued in Lecture 7.

Exercise 1 *What is the time-complexity, in terms of the size of the set(s) being manipulated, of the various operations in our implementation?*

Add ordered list implementation of sets here. Show how functors may be used to construct different implementations.