Trees

Michael P. Fourman

February 2, 2010

Trees come in many shapes and sizes. We begin this note with simple syntax trees, and then consider more abstract varieties.

1 Abstract Syntax

Expressions, programs, and sentences, are normally presented as strings of characters. The syntax of a language determines which strings are legal. More important, it allows us to ascribe a hierarchical structure to a legal phrase; for example, an ML conditional expression is composed from three sub-expressions, which may, in turn, have their own internal structure. Trees are ideal for representing hierarchic structures. An *abstract syntax tree* is what we are left with when we discard the concrete representation of a phrase as a string, and retain only its structure.

Here is an example. We represent the abstract syntax of a class of algebraic expressions by an ML datatype declaration:

```
datatype Expn = X
              | K of int
              | ** of Expn * Expn
              | ++ of Expn * Expn
```

The algebraic expression $4x \times (3x + 5)$ is represented by the ML value
`(((K 4) ** X) ** (((K 3) ** X) ++ (K 5)))`.

By contrast, $4x{\times}3x{+}5$ is represented as `((((K 4) ** X) ** ((K 3) ** X)) ++ (K 5))`. The difference is in the bracketing, and is quite obscure. To make it clear, we draw the two ML values as trees:

In Practical 4 you will see how to take an expression represented as a string, and produce the corresponding abstract syntax tree. For the moment, we look at the inverse process: we take an abstract syntax tree and produce a string. There are various ways to do this: infix, prefix, and postfix.

```
fun inf X        = " x "
  | inf (K n)    = makestring n
  | inf (a ** b) = inf a ^ " * " ^ inf b
  | inf (a ++ b) = inf a ^ " + " ^ inf b
```

This function produces the same string "4 * 3 * x + 5" for the two different trees; without parentheses, or precedence rules, this concrete syntax is ambiguous. The abstract syntax represents the difference structurally.

We can also use prefix or postfix syntax; these are unambiguous. Can you see why?

```
fun pre X        = " x "
  | pre (K n)    = makestring n
  | pre (a ** b) = " * " ^ pre a ^" "^ pre b
  | pre (a ++ b) = " + " ^ pre a ^" "^ pre b

fun pos X        = " x "
  | pos (K n)    = makestring n
  | pos (a ** b) = pos a ^" "^ pos b ^ " * "
  | pos (a ++ b) = pos a ^" "^ pos b ^ " + "
```

Prefix notation is also called "Polish" notation, as it was used in early studies of syntax by Polish logicians in the 1920s. Postfix notation is therefor called "Reversed Polish Notation", or RPN. The RPN for our first example is "4 x * 3 x * 5 + *"; the second is given by "4 x * 3 x * * 5 +"

Algebraic transformations may be expressed in terms of operations on trees. For example, the associative law $(x + y) + z = x + (y + z)$ can be pictured by drawing the two trees.

The operations that transform one form to the other are called tree *rotations*

```
fun rotateright
       ((x ++ y) ++ z) = x ++ (y ++ z)

and rotateleft
       (x ++ (y ++ z)) = (x ++ y) ++ z
```

We now generalise our discussion to more abstract trees that can be used to represent many different kinds of syntax, and much more besides.

2 Binary Trees

A binary tree is either a leaf, or a node with two subtrees

```
datatype BinTree = Lf
                 | Nd of BinTree * BinTree
```

This declaration would capture the structure of a binary tree. But it gives us bare trees, which are of little use. Usually, we will want to dress our trees in data, putting data items at the nodes. So we define a polymorphic type of trees, including a data item of unknown, but fixed, type at each node. We introduce a type variable `'a` to represent this unknown type.

```
datatype 'a BinTree =
         Lf
       | Nd of 'a BinTree * 'a * 'a BinTree
```

This makes `BinTree` into a type constructor, just like the built-in type constructor `list`. In fact the analogy is quite strong: lists are just unary trees, `nil` is a leaf, and cons, `::`, creates a new node, decorated with a data item. Since a list can only have one leaf, there is not much to be gained by attaching data to the leaf of a list. Trees, however, generally have many leaves, so it will be useful to attach data to leaves, as well as nodes. For this purpose, in our final version of the declaration for `BinTree`, we introduce two type variables (there is no reason to make the decoration of the leaves have the same type as that of the nodes):

```
datatype ('a, 'b)BinTree =
    Lf of 'b
  | Nd of  ('a, 'b)BinTree
         * 'a
         * ('a, 'b)BinTree
```

We can represent our syntax examples by defining new types to represent the atoms and operations of our language for expressions:

```
datatype Atom = X  | K of int
     and Opn  = ** | ++

Nd(
   Nd(Lf(K 4), **, Lf X),
   ** ,
   Nd(
      Nd(Lf(K 3), **, Lf X),
      ++,
      Lf(K 5)
     )
  );

val it = Nd ... : (Opn, Atom) BinTree
```

(The indentation is intended to indicate the hierarchical structure. This syntax is much more unwieldy, but we would define auxiliary functions to create values of this type. The benefit is that general-purpose, polymorphic functions on binary trees can be applied to a wide variety of applications without change.

As usual, we can define functions on the new datatype using pattern-matching. For example, the height function: the height of a leaf is zero, and the height of an internal node is one greater than the maximum of the heights of its children.

```
fun height (Lf _) = 0
  | height (Nd(l,_,r)) =
       1 + max (height l) (height r)
```

As usual, the pattern of the function definition follows that of the datatype declaration. Here is another example, a function to list the (data in the) leaves of a binary tree:

```
fun leaves (Lf a)       = [a]
  | leaves (Nd(l,_,r)) = leaves l @ leaves r
```

This is correct, but the use of append means that its worst case complexity is quadratic, $O(n^2)$, where n is the number of leaves to be listed. Here is another implementation, this time with linear, $O(n)$, complexity:

```
local
  fun accl(Lf a,      acc) = a :: acc
    | accl(Nd(l,_,r), acc) =
                accl(l, accl(r, acc))
in
  fun leaves t = accl(t, [])
end
```

There is often a conflict between clarity and efficiency; we should only sacrifice clarity with good cause. This sacrifice is only defensible when there is a significant efficiency gain, *and* the function you are optimising is critical to the performance of the system as a whole.

We use the same trick to implement functions that list the nodes of a tree in three orders, corresponding to the infix, prefix, and postfix notations given earlier. Again we also give clear, but lamentably inefficient, alternative implementations.

```
fun inf (Lf _)       = []
  | inf (Nd(l, v, r) = inf l @ [v] @ inf r

local
   fun infacc (Lf _, acc) = acc
     | infacc (Nd(l, v, r), acc) =
       infacc(l, v :: infacc(r, acc))
in
   fun inf t = infacc(t, [])
end

fun pre (Lf _)       = []
  | pre (Nd(l, v, r) = v :: pre l @ pre r

local
   fun preacc (Lf _, acc) = acc
     | preacc (Nd(l, v, r), acc) =
          v :: preacc(l, preacc(r, acc))
in
   fun pre t = preacc(t, [])
end

fun pos (Lf _)       = []
  | pos (Nd(l, v, r) = pos l @ pos r @ [v]
local
   fun posacc (Lf _, acc) = acc
     | posacc (Nd(l, v, r), acc) =
          posacc(l, posacc(r, v ::  acc))
in
   fun pos t = posacc(t, [])
end
```

How do we see that the complexity of the efficient versions is $O(n)$? We observe that the cons operation, `::`, is applied to each node exactly once.

Binary Search Trees

We can search for an item in a tree very efficiently if the elements are ordered, and the tree is balanced. In this section we look at functions to insert and retrieve elements from a *binary search tree* without worrying about keeping the tree balanced. Deletion, and techniques for balancing will be covered later in the course. For simplicity, we assume that we are dealing with `(int * 'a)` trees, where the integer is the key, and is used to order the tree. The function to look for an element with a particular key may be written as

```
fun lookup (e: int) Lf = raise Lookup
  | lookup e (Nd(t1, (k,v), t2) =
      if      e < k then lookup e t1
      else if e > k then lookup e t2
      else v;
```

Insertion looks like

```
fun insert p Leaf = Nd(Leaf, p, Leaf)
  | insert (p as (e:int, _))
           (Nde(t1, r as (k,_), t2) =
      if      e < k then Nd(insert p t1, r, t2)
      else if e > k then Nd(t1, r, insert p t2)
      else Nd(t1,p, t2) (* replace existing entry *)
```

Note that the function returns a new tree, leaving the original unchanged.

3 Bushy Trees

A tree consists of a collection of *nodes* — each of which may have some number of *children*. A node with no children (zero is a perfectly good number) is called a leaf; other nodes are called *internal* nodes. One distinguished node, called the *root,* has no parents; every other node is a descendant of the root (in just one way).

To implement a type of trees in ML, we consider a structure to represent a node, together with the subtrees whose roots are its children. We represent this collection of subtrees as a list. Using this idea, bare trees may be represented by the following datatype:

```
datatype Tr = Tr of Tr list
```

This seems a bit odd, but we can get started since a leaf is just `Tr([])`. We can define functions on trees by pattern-matching;

```
fun height (Tr []) = 0
  | height (Tr xs) = 1 + maxl (map height xs);
```

This makes use of the function `maxl` given in an appendix to this note.

Since we'll want to decorate these trees too, we again make a polymorphic declaration. This time there is no need to distinguish the decorations of nodes and leaves.

```
datatype 'a Tr = Tr of 'a * 'a Tr list
```

For instance a function to sum the labels on the nodes of a tree is given by

```
     fun sumTree (Tr (a, ts)) =
            a + sum (map sumTree ts)
```

Appendix: Some useful functions

The following functions are provided in the Prelude

```
fun max a b :int = if a > b then a else b
fun min a b :int = if a < b then a else b

exception Maxl Minl
fun maxl []       = raise Maxl
  | maxl [a]      = a
  | maxl (h :: t) = max h (maxl t)

fun minl []       = raise Minl
  | minl [a]      = a
  | minl (h :: t) = min h (minl t)

fun sum []        = 0
  | sum (h :: t)  = h + sum t

fun prod []       = 1
  | prod (h :: t) = h * prod t

fun times a b = a * b : int
and plus  a b = a + b : int
and mult  a b = a * b : real
and add   a b = a + b : real
```