15-150 Fall 2019

Stephen Brookes

Lecture 7
Sorting a tree of integers
1 Outline

- Representing integer trees in ML.
- Tree-based mergesort: a lesson in design and implementation.
- Specifications, correctness and proofs
- Work and span analysis

2 Background

As in previous lecture, we refer to:

datatype order = LESS | EQUAL | GREATER;

(* compare : int * int -> order *)
fun compare(x:int, y:int):order =
    if x<y then LESS else
    if y<x then GREATER else EQUAL;

(* compare(x,y)=LESS if x<y *)
(* compare(x,y)=EQUAL if x=y *)
(* compare(x,y)=GREATER if x>y *)

A list of integers is *sorted* if each item in the list is \( \leq \) all items that occur later in the list.

We will also refer to the *ins* function, used as a helper when we did insertion sort on lists of integers.

(* ins : int * int list -> int list *)
(*REQUIRES L is sorted *)
(* ENSURES ins(x, L) evaluates to a sorted permutation of x::L. *)

fun ins (x, [ ] ) = [x]
| ins (x, y::L) = case compare(x, y) of
    GREATER => y::ins(x, L)
| _ => x::y::L
3 Trees in ML

We can use a recursive datatype definition to introduce a type whose values represent (binary) trees. In fact we can do this in a uniform and general way, parameterized by a choice of the type of data to appear at the nodes of trees. In ML a “type variable” is written like ’a or ’b. The datatype definition

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

introduces a type constructor tree, together with two value constructors
Empty: ‘a tree and Node: ‘a tree * ‘a * ‘a tree -> ‘a tree for building tree values. Since this is a user-defined type, these are the only ways you can build tree values. Every value of type int tree is either Empty, or has the form Node(A, x, B) where A and B are values of type int tree, and x is an integer value. We say that A is the left-child and B is the right-child; x is the integer “at the root”.

Example: The expression Node(Empty, 42, Node(Empty, 99, Empty)) has type int tree. This expression is also a value.

A value of type int tree represents a binary tree with integers at its nodes; the Empty tree value contains no integer data. Every non-empty tree has a piece of data at its root and two sub-trees or children, which may be empty.

The constructors can also be used for pattern-matching against values of type tree. The pattern Empty only matches the value Empty. A pattern Node(p1, p, p2), in which p1, p and p2 are patterns, matches tree values of the form Node(v1, v, v2) such that p1 matches v1, p matches v, and p2 matches v2. For example, the pattern Node(A, x, B) matches non-empty tree values and binds A to the left subtree, x to the root value, and B to the right subtree. Similarly, Node(Empty, x, Empty) matches only non-empty trees with a single node, and binds x to the value at the root.

It is convenient to draw pictures of tree values, rather than always using the ML syntax for tree expressions. In pictures we usually omit drawing Empty nodes explicitly. We draw the root node at the top, sub-trees lower, left subtree to the left, and so on. (We usually don’t draw Empty explicitly, but sometimes we may do that instead.) For example, let t be the ML expression below:

```
Node(Empty, 42, Node(Empty, 9, Empty)).
```

This tree value T can be drawn (without showing Empty sub-trees) as:
And the tree Node(T, 0, T) looks like:

```
   0
  / \
42 42
\  \
9   9
```

In each case all the “missing” Empty subtrees can easily be filled in; their positions are implied by the tree shape.

Some simple ML code for building “full binary trees”:

```ml
fun Leaf(x:int): int tree = Node(Empty, x, Empty);

fun Full(x:int, n:int): int tree =
  if n=0 then Empty else
    let val T = Full(x, n-1) in Node(T, x, T) end;
```

The function Leaf:int -> int tree builds a tree with a single node. We refer to this as a “leaf”. The expression Full(2, 5) evaluates to a “full” binary tree with 2 at each node and with depth (or height) 5.

Draw pictures of Leaf 42 and Full(42, 3). Note that the expression Full(42, 3) evaluates to a tree value with 7 nodes, each with the integer 42.

**Evaluation and equality**

To evaluate the ML expression Node(e1, e, e2) we must evaluating e1 to a tree value (say v1), evaluate x to an integer (say v), and evaluate e2 to a tree value (say v2). The final value obtained by this evaluation is then Node(v1, v, v2).

Two ML expressions of type int tree are equal if they both evaluate to the same tree value, or they both fail to terminate.

For example, Leaf 42 and Full(42, 1) are equal, because they both evaluate to Node(Empty, 42, Empty).

The type int tree defined as above is actually an ML equality type (because int is an equality type), so we can use ML = for testing when two tree
values are identical. We won’t use this feature in our sorting functions, but it may be handy for testing.

**Structural induction for trees**

To reason about trees, and functions on trees, we need a form of induction that works with trees. The *structure* of the datatype definition for trees is the key here. Every tree value is either `Empty`, or has the form `Node(l,x,r)`, where `l` and `r` are tree values and `x` is an integer value. This amounts to an inductive way to generate tree values. Initially we start with the tree value `Empty`. Then we generate all trees obtainable by applying the `Node` constructor to already existing trees and some integer. And we keep repeating this process. At each stage the set of trees generated so far grows larger.

We can prove a property like “for all trees `T`, `P(T)` holds”, as follows:

(i) Base case: Show that `P(Empty)` holds.

(ii) Inductive step: Assume as Induction Hypothesis that `l` and `r` are tree values such that `P(l)` and `P(r)` hold; show that for all integer values `x`, `P(Node(l,x,r))` holds.

(iii) It follows from (i) and (ii) that `P(T)` holds, for all tree values `T`.

This proof method is called *structural induction for trees*.

There is also a corresponding principle of structural induction for function definitions. To define a function `F` on all tree values:

- Give a clause defining `F(Empty)`.
- Give a clause defining `F(Node(A,x,B))` in terms of `F(A)` and `F(B)`.

Such clauses are sufficient to completely specify the intended value of `F(T)`, for all tree values `T`. We say that these clauses constitute a *definition (of F)* by *structural induction on trees*.

For every datatype definition in ML there is an analogous principle of structural induction. We will see many examples later in the semester. You have already seen one: the kind of list induction discussed earlier is basically a form of structural induction, since the ML list types are defined in terms of `nil` and “`cons`”.

5
size and depth of trees

The size of a tree is the number of nodes it contains. So the size of Empty is 0, and the size of a non-empty tree is the sum of 1 and the sizes of its two children. We can easily define an ML function size : ‘a tree -> int that computes the size of a tree, using structural recursion:

fun size Empty = 0
| size (Node(t1, _, t2)) = size t1 + size t2 + 1

It is easy to check that

size(Full(42,3)) = 7

Intuitively, size(t) is the number of nodes in t; using structural induction for trees, it is easy to prove this.

The depth of a tree is the length of the longest path from the root of the tree to an Empty subtree. A path is a sequence of nodes. The depth of an empty tree is defined to be 0.

For all trees t, size(t) ≥ 0 and depth(t) ≥ 0; and if t' is a child of t, then depth t' < depth t and size t' < size t. So we can also use induction on tree depth, or induction on tree size, as techniques for proving properties of trees, or of functions operating on trees.

NOTE: structural induction on trees, induction on tree size, and induction on tree depth, as well as simple and complete induction on non-negative integers, are all special cases of a general technique known as well-founded induction.

In-order traversal

Here is a function that builds a list of integers from an integer tree, by making an in-order traversal of the tree, collecting data into a list. In-order traversal of a non-empty tree involves traversing the left-child, then the root, and then traversing the right-child; we also use in-order traversal on the sub-trees. Obviously this description suggests that we define a recursive function!

This function is used mainly in specifications, but serves as an example of how to define a function that operates on trees: use clauses, one for the empty tree and one for non-empty trees, using pattern-matching to give names to the components of a tree.
\begin{verbatim}
(* trav : 'a tree -> 'a list *)
fun trav Empty = [ ]
  | trav (Node(t1, x, t2)) = trav t1 @ (x :: trav t2);
%
(* Specification: *)
(* trav t = a list of the data in the tree t, *)
(* as seen in an in-order traversal of the tree *)
%
(* Say "x is in t" if x is a member of the list trav(t). *)

For example, for the tree t above, trav(t) = [42, 9]. And

\[
\text{trav(Node(t, 0, t))} = \text{trav t} @ (0 :: \text{trav t})
= [42, 9] @ (0 :: [42, 9])
= [42, 9, 0, 42, 9].
\]

We prove, by structural induction on trees, that for all trees \( t \), \( \text{trav}(t) \) evaluates to a list of length equal to \( \text{size}(t) \). This is the same as saying “\( \text{trav}(t) = \text{a list of length size}(t) \)”. Proof: by structural induction on \( t \).

- Base case: For \( t = \text{Empty} \). Since \( \text{size(Empty)} = 0 \) we must show that \( \text{trav Empty} = [ ] \). This is obvious from the function definition.

- Inductive case: For \( t = \text{Node}(t1, x, t2) \). Let \( n1 = \text{size } t1 \) and \( n2 = \text{size } t2 \). So \( \text{size}(t) \) is \( n1+n2+1 \). Assume as Ind. Hyp. that

  (i) \( \text{trav } t1 = \text{a list (say } L1) \text{ of length } \text{size}(t1) \)
  (ii) \( \text{trav } t2 = \text{a list (say } L2) \text{ of length } \text{size}(t2) \).

Then by definition of \( \text{trav} \) we have

\[
\text{trav } (\text{Node}(t1, x, t2))
= (\text{trav } t1) @ (x :: \text{trav } t2)
= L1 @ (x :: L2)
\quad \text{by (i) and (ii)}
= \text{a list of length } n1 + 1 + n2
= \text{a list of length } \text{size}(\text{Node}(t1, x, t2))
\]

as needed.
\end{verbatim}