15-150 Fall 2013

Stephen Brookes

Lecture 8
Sorting an integer tree

1 Outline

- Representing integer trees in ML.
- Tree-based mergesort.
- Specifications, correctness and profs
- Work and span analysis

2 Background

As in previous lecture, we refer to:

```
type order = LESS | EQUAL | GREATER;

(* Comparison for integers *)

(* 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. Here is an ML function that checks for this property. We only use it in specifications!

```
(* sorted : int list -> bool *)
fun sorted [ ] = true
 |  sorted [x] = true
 |  sorted (x::y::L) = (compare(x,y) <> GREATER) andalso sorted(y::L);

(* sorted L = true iff L is <-sorted. *)
```

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 *)
fun ins (x, [ ]) = [x]
|   ins (x, y::L) = case compare(x, y) of
                        GREATER => y::ins(x, L)
                      |  _       => x::y::L;

(* Specification: *)
(* If L is sorted, ins(x, L) evaluates to a sorted permutation of x::L. *)
```

This insertion function for integer lists will be useful later.

3 Integer trees in ML

```
datatype tree = Empty | Node of tree * int * tree;

(* Empty : tree   *)
(* Node : tree * int * tree -> tree *)
```

Example: The expression `Node(Empty, 42, Node(Empty, 99, Empty))` has type `tree`.

We have here introduced a new type, named `tree`, along with constructors `Empty` and `Node` for building and pattern-matching on values of this type. Since this is a user-defined type, *these are the only ways you can build values of type* `tree`. And every value of type `tree` is either `Empty`, or has the form `Noe(l,x,r)` where `l` and `r` are also values of type `tree`, and `x` is an integer value.

A value of type `tree` represents a binary tree with integers at its (internal) nodes; leaf nodes are empty (i.e. have no data attached). With this representation, every non-empty tree has a piece of data and two sub-trees or children, which may be empty.

In a tree value of form `Node(l,x,r)` we say that `l` is the left-child and `r` is the right-child; `x` is the integer "at the root".

We can draw pictures of trees by putting the root integer at the top, as usual, and we may omit drawing leaf nodes. For example, let `t` be the tree `Node(Empty, 42, Node(Empty, 9, Empty))`. This can be drawn as:

```
t =   42
        \
         9
```

And the tree `Node(t, 0, t)` looks like:

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

Structural induction for trees

To reason about functions on trees we need a form of induction that works with trees. The form of the datatype definition for trees is the key here. Every tree (every value of type `tree`) is either `Empty`, or is a non-empty tree of the form `Node(l,x,r)`, where `l` and `r` are *smaller* tree values. We can prove a property true of all tree values, say "for all trees `T`, P(`T`) holds", as follows:

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

(ii) Inductive step: For a non-empty tree of form `Node`(`l`, `x`, `r`), assume as Induction Hypothesis that P(`l`) and P(`r`) hold; show that 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.*

For every recursive datatype definition in ML there is an analogous version of structural induction. We will see many examples later in the semester. You have already seen one: list induction is basically a form of structural induction, since the ML integer list type is defined in terms of `nil` and "cons".

depth and size

Let `max:int*int -> int` be the usual integer maximum function:

```
fun max(x:int, y:int):int = if x>y then x else y;
```

We define the functions `depth` and `size` of type `tree -> int` by:

```
fun depth Empty = 0
 |  depth (Node(t1, _, t2)) = max(depth t1, depth t2) + 1;

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

Intuitively, `size(t)` computes the number of non-leaf nodes in `t`, and `depth(t)` computes the length of the longest path from the "root" of `t` to a leaf node.

Using structural induction for trees, it is easy to prove that:

(1) For every value `t` of type `tree`,

```
size t = a non-negative integer.
```

(2) For every value `t` of type `tree`,

```
depth t = a non-negative integer.
```

We refer to `depth(t)` as the depth (or height) of `t`; this corresponds to the length of the longest path from the root of `t` to a leaf.

We refer to `size(t)` as the size of `t`; this corresponds to the number of integers in `t`.

So for all trees `t`, $\texttt{size}(\texttt{t}) \geq 0$ and $\texttt{depth}(\texttt{t}) \geq 0$; and if $\texttt{t}'$ is a child of `t`, then $\texttt{depth t}' < \texttt{depth t}$ and $\texttt{size t}' < \texttt{size t}$. 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 a 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.

```
(* trav : tree -> int list *)
fun trav Empty = [ ]
|   trav (Node(t1, x, t2)) = trav t1 @ (x :: trav t2);

(* Specification: *)
(* trav t = a list of the integers 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

```
trav(Node(t, 0, t)) = trav t @ (0 :: trav t)
                    = [42, 9] @ (0 :: [42, 9])
                    = [42, 9, 0, 42, 9].
```

We prove, by structural induction on trees, that for all trees `t`, `trav(t)` evaluates to a list of length equal to `size(t)`. This is the same as saying "`trav(t) = a list of length size(t)`".

Proof: by structural induction on `t`.

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

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

  (i) `trav t1 = a list (say L1) of length size(t1)`

  (ii) `trav t2 = a list (say L2) of length size(t2)`.

  Then by definition of `trav` we have

  ```
  trav (Node(t1, x, t2))
     = (trav t1) @ (x :: trav t2)
     = L1 @ (x :: L2)                     by (i) and (ii)
     = a list of length n1 + 1 + n2
     = a list of length size(Node(t1, x, t2))
  ```

  as needed.

Sorted trees

Informally, we say that a value of type `tree` is sorted if the integers in the tree occur in sorted order. More precisely we intend this to mean that the in-order traversal list of the tree is sorted.

Equivalently, an empty tree is sorted, and a non-empty tree is sorted if and only if its two sub-trees are sorted, every integer in the left subtree is less-or-equal to the integer at the root, and every integer in the right subtree is greater-or-equal than the integer at the root. (These two characterizations of sortedness for trees are equivalent, but we won't prove that here.)

We can easily implement an ML function for testing sortedness:

```
fun Sorted t = sorted(trav t);

(* Specification: *)
(* Sorted(t)=true if and only if t is a sorted tree. *)
```

Using `trav` like this is an easy way to make a slightly vague assertion about the contents of a tree into a rigorous one. So, a tree is called "sorted" if and only if its traversal list is sorted in the sense we are familiar with. Similarly, we will say that one tree `t1` is a "permutation" of another tree `t2` if and only if the list `trav`(`t1`) is a permutation of `trav`(`t2`).

We will only use this `Sorted` function in testing, never in our code! It would be ridiculously expensive to make calls to this function.

Insertion for trees

Insertion sort is not well suited to parallel implementation, even if we are inserting into a tree. Nevertheless the tree-based analogue of the insertion function on lists is still of interest. We use capitalization to distinguish this function from the `ins` function on lists used in the previous class.

```
(* Ins : int * tree -> tree *)
fun Ins (x, Empty) = Node(Empty, x, Empty)
|   Ins (x, Node(t1, y, t2)) =
     case compare(x,y) of
            GREATER   => Node(t1, y, Ins(x, t2))
          | _         => Node(Ins(x, t1), y, t2);
```

Compare this code with the code for `ins`.

The tree-based insertion function satisfies the following specification (which actually has two parts):

```
(* For all t:tree, if Sorted(t)=true then Sorted(Ins(x,t))=true.        *)
(* For all y:int, y is in Ins(x,t) if and only if (x=y) or (y is in t). *)
```

Another way to assert the same properties is:

```
(* For all t:tree, if t is sorted, then Ins(x,t) =  a sorted tree t'  *)
(* such that trav(t') is a permutation of x::trav(t).                 *)
```

Exercise: prove by structural induction on trees that `Ins` satisfies this specification.

Exercise: now prove the same result by induction on tree depth.

Exercise: now prove the same result by induction on size.

We can also prove the following relationship between `Ins` and `ins`:

```
(* For all x:int, t:tree,                                *)
(* if t is sorted then trav(Ins(x,t)) = ins(x, trav t). *)
```

Actually we can just prove this result relating `Ins` to `ins`, and then rely on the spec that we gave earlier (and proved) for `ins`: that will give us enough information to derive the other two-part specification for `Ins`. Here is a proof of this connection between `Ins` and `ins`, by induction on tree structure. Refer back to the definition of `ins`.

- Base case: For `t = Empty`, we have:

  ```
  trav(Ins(x, Empty)) = trav(Node(Empty,x,Empty) = [x]
  ins(x, trav Empty) = ins(x, [ ]) = [x]
  ```

- Inductive case: For a sorted tree $\texttt{t} = \texttt{Node}(\texttt{t1}, \texttt{y}, \texttt{t2})$, assume as Induction Hypothesis that the relationship holds for trees with smaller depth than `t`. We do case analysis based on the result of comparing `x` and `y`. If $\texttt{x} > \texttt{y}$, then:

  ```
  trav(Ins(x, Node(t1, y, t2)) = trav(Node(t1, y, Ins(x, t2))
       = trav t1 @ (y :: trav(Ins(x, t2)))   by def of trav
       = trav t1 @ (y :: ins(x, trav t2))    by ind hyp
       = trav t1 @ ins(x, y :: trav t2)    by def of ins, x>y
       = ins(x, trav t1 @ (y::t2))        because t is sorted
       = ins(x, Trav (Node(t1, y, t2)))    by def of Trav
  ```

  Note that the use of the induction hypothesis is justified because `t2` has smaller depth than `t` and `t2` is also sorted (because it is a child of the sorted tree `t`).

  The other cases ($x \leq y$) are similar. That completes the proof.

Now show that this result connecting `Ins` and `ins`, together with the specification we gave earlier for `ins`, implies the two-part spec for `Ins`.

4 Splitting a tree

In adapting the mergesort algorithm to operate on trees we need a suitable analog to the `split` function. It isn't easy to figure out a good way to hew a tree into two roughly equal sized pieces, based solely on the structure of the tree. Instead, we will start from a tree and an integer, and break the tree into two trees that consist of the items in the tree less-or-equal to the integer and the items greater than the integer. We will only ever need to use this method on a sorted tree, as you will observe when we develop the code. Indeed the design of this function takes advantage of the assumption that the tree is already sorted, a fact that we echo in the way we write the function's specification.

```
(* SplitAt : int * tree -> tree * tree *)
fun SplitAt(y, Empty) = (Empty, Empty)
 |  SplitAt(y, Node(t1, x, t2)) =
     case compare(x, y) of
           GREATER => let
                         val (l1, r1) = SplitAt(y, t1)
                      in
                         (l1, Node(r1, x, t2))
                      end
         |  _      => let
                         val (l2, r2) = SplitAt(y, t2)
                      in
                         (Node(t1, x, l2), r2)
                      end;

(* Specification: *)
(* If Sorted(t) = true, then                                         *)
(*     SplitAt(y, t) = a pair (t1, t2) such that                      *)
(*     every item in t1 is less-or-equal to y, and                    *)
(*     every item in t2 is greater-or-equal to y, and                 *)
(*     trav(t1)@trav(t2) is a permutation of trav(t).                 *)
```

Prove that `SplitAt` satisfies this specification, by induction on the depth of the tree.

- Base case: For `t = Empty`, we get `t1 = Empty` and `t2 = Empty`, and the requirements in the spec hold trivially.
- Inductive step: Let `t` be `Node(t1, x, t2)` and assume that `SplitAt` satisfies the spec on all trees with smaller depth than `t`. Show that `SplitAt(y,t)` has the desired properties. There are two sub-cases to analyze, branching on the result of comparing the values of `x` and `y`. The assumption that `t` is sorted is crucial here. Be sure to check why!

5 Merging two trees

Now the tree-based analog of `merge`: a function that takes a pair of sorted trees and combines their data into a single (also sorted) tree.

```
(* Merge : tree * tree -> tree *)
fun Merge (Empty, t2) = t2
 |  Merge (Node(l1,x,r1), t2) = let
                                  val (l2, r2) = SplitAt(x, t2)
                                in
                                  Node(Merge(l1, l2), x, Merge(r1, r2))
                                end;

(* Specification: *)
(* If Sorted(t1)=true & Sorted(t2)=true,            *)
(*   Merge(t1, t2) = a sorted tree t such that      *)
(*   trav(t) is a permutation of trav(t1)@trav(t2). *)
```

The proof that `Merge` meets this spec relies on the fact (shown above) that `SplitAt` meets its own specification. Indeed, we deliberately chose a spec for `SplitAt` that would help us to prove `Merge` correct. That's one of the skills that we want you to learn: the art of choosing helper functions and specs wisely!

Exercise: do the proof!

6 Mergesort for trees

Using `Ins` and `Merge`, and guided by their specs, we are now ready to define a mergesorting function for integer trees.

```
(* Msort : tree -> tree *)
fun Msort Empty = Empty
|   Msort (Node(t1, x, t2)) = Ins (x, Merge(Msort t1, Msort t2));

(* Specification: *)
(* For all t:tree, Msort(t) = a sorted permutation of t. *)
```

Again the proof that `Msort` meets this spec uses the facts (shown earlier) that `Ins` and `Merge` satisfy their specs. And again these helper specs were carefully chosen to make this all fit together!

Exercise: fill in the proof details. Contrast with the proof given in the earlier lecture notes for the mergesort function on lists.

7 Depth analysis

There can be many different trees containing the same integers. Indeed, there can be many different *sorted* trees containing the same integers. So the specifications and proofs so far don't really tell us much about the shapes of the trees produced by sorting.

We can prove some useful (and intuitively obvious) results about depth. These will be helpful when we analyze the runtime behavior of the code.

The following results are provable, by choosing an appropriate kind of induction.

(1) For all trees `t` and integers `x`,

```
depth(Ins(x, t)) <= depth(t) + 1.
```

[Use structural induction, since `Ins(x, t)` makes a recursive call on a child of `t`.]

(2) For all trees `t` and integers `y`, if `SplitAt(y,t)=(t1, t2)` then

```
depth(t1) <= depth(t) and depth(t2) <= depth(t).
```

[Use structural induction, since `SplitAt(y,t)` makes a recursive call on a child of `t`.]

(3) For all trees `t1` and `t2`,

```
depth(Merge(t1, t2)) <= depth t1 + depth t2.
```

[Use induction on the structure of `t1`.]

(3) For all trees `t`,

```
depth(Msort t) <= depth t.
```

[Use induction on the structure of `t`.]

8 Size analysis

We can also prove some fairly obvious facts about the effects of the operations on the size of a tree.

(1) For all trees `t` and integers `x`,

```
size(Ins(x, t)) = size(t) + 1.
```

(2) For all trees `t` and integers `y`, if `SplitAt(y,t)=(t1, t2)` then

```
size(t1) + size(t2) = size(t).
```

(3) For all trees `t1` and `t2`,

```
size(Merge(t1, t2)) = size t1 + size t2.
```

(3) For all trees `t`,

```
size(Msort t) = size t.
```

In each case, you can find a suitable inductive method: either structural induction, or induction on size, or on depth.

9 Work and span

We've shown how to derive recurrence relations for the *work* of a sequentially executed piece of code, and how to estimate asymptotically what the runtime is on "'large" inputs, using big-O notation.

Now we have some functions operating on trees for which it makes a lot of sense to consider using parallel evaluation. The span of a code fragment is obtained by assuming that we have as many parallel processors as we need, and taking the *maximum* runtime of code pieces that can be evaluated independently; we still use addition for the run ties of code fragments that need to be executed in sequential order, typically because of a data dependency: one fragment needs the result of the other. Operating on trees allows us in principle to sort the left and right children of a node in parallel, since their results do not depend on each other. Of course, these tasks need to be completed before the merging phase. And the splitting phase needs to go first.

These facts guide us in analyzing the span. Here is a rough outline. With trees there are two "largeness" measures of interest: depth and size.

- The work and span for `Ins(x,t)` is $\mathtt{O}(d)$, where d is the depth of `t`. Reason: `Ins`(`x`, `t`) makes a single recursive call, on a subtree with depth decreased by 1.

- `SplitAt`(`y`, `t`) has span $\mathtt{O}(d)$, where $d =$ `depth t`. Reason: makes a single recursive call, on a tree with depth one less.

- `Merge`(`t1`, `t2`) has span $\mathtt{O}(d_1 d_2)$, where d_1, d_2 are `depth t1`, `depth t2`.

- Assuming that the trees produced by `Msort` are *balanced*, so that their depth is about the logarithm of their size, `Msort`(`t`) has span $\mathtt{O}(d^3)$, where d is the depth of `t`. Reason: making the balance assumption leads us to the recurrence

$$\begin{aligned} S_{\mathtt{Msort}}(d) &= S_{\mathtt{Ins(d)}} + S_{\mathtt{Merge}}(d-1) + S_{\mathtt{Msort}}(d-1) \\ &= d + (d-1)^2 + S_{\mathtt{Msort}}(d-1) \end{aligned}$$

  for balanced trees of depth $d > 1$. Expanding out, and observing that the sum of the first d squares is proportional to d^3, we deduce that the span is $\mathtt{O}(d^3)$. Since the size n of a balanced tree and its depth d satisfy $d = \mathtt{O}(log\, n)$, our analysis shows that the span for `Msort`(`t`) on balanced trees of size n is $\mathtt{O}((log\, n)^3)$.

Thus (ignoring constants), when we sort a billion integers in a balanced tree, the length of the longest critical path is about 27000 operations, so we can exploit over a million processors!

This would be true, except for the bug in the above analysis! We assumed implicitly in the rough analysis (and explicitly in the preamble) that the trees passed by `Msort` to `Merge` were balanced. However, this is not necessarily the case, because even if we assumed that the *original* tree was balanced, these two trees have been built by calling `Msort` (albeit on balanced trees). We haven't proven that `Msort` applied to a balanced tree will produce a balanced tree. In fact, this isn't necessarily true. The best that our analysis really predicts is that the span of this algorithm can't actually be better than this bound, because we obtained this bound by making the most optimistic assumptions about the structure of the tree.

Later we will discuss how to implement binary trees with insertion and deletion operations that are guaranteed to build trees with a reasonable balance property built in. When we get there, you might want to come back and see how you could adapt the code above to fit with these better behaved trees.

Exercise:
A student pointed out that there is a way to define a version of tree-mergesort that avoids using `Ins`, instead calling `Merge`:

```
fun Msort’ Empty = Empty
 |  Msort’ (Node(t1, x, t2)) =
          Merge(Node(Empty,x,Empty), Merge(Msort’ t1, Msort’ t2));
```

Would this be extensionally equivalent to the previous function? Does this function satisfy the same specification as before, i.e. does it still sort? And is it as efficient, or more efficient?