15–150: Principles of Functional Programming

Sorting Integer Trees

Michael Erdmann[*]
Spring 2023

1 Background

As in the previous lecture, we refer to:

```
datatype order = LESS | EQUAL | GREATER

(* Comparison for integers *)

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

As previously, a list of integers is *sorted* if each item in the list is no greater than all items that occur later in the list. Here is an SML function that checks for this property:

```
(* sorted : int list -> bool
   REQUIRES: true
   ENSURES:  sorted(L) evaluates to true if L is sorted and to false otherwise.
*)
fun sorted [ ] = true
  | sorted [x] = true
  | sorted (x::y::L) = (compare(x,y) <> GREATER) andalso sorted(y::L)
```

[*] Adapted from documents by Stephen Brookes and Dan Licata.

We will also refer to the ins function, which we used as a helper function for sorting integer lists.

```
(* 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)
```

2 Integer Trees in SML

We will use the following integer tree type:

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

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
```

depth and size

Let `max : int*int -> int` be the usual integer maximum function (predefined in SML as `Int.max`):

```
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.

We refer to size(t) as "the size of t" and to depth(t) as the "depth of t".

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.

Aside: 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 subtrees. 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.

(You have probably seen or will see this function in other settings with other names, but the idea is the same throughout: in-order traversal of a tree.)

```
(* trav : tree -> int list
   REQUIRES: true
   ENSURES:  trav(t) evaluates to a list consisting of the integers in t,
             in the same order as seen during an in-order traversal of t.
*)
fun trav Empty = [ ]
  | trav (Node(t1, x, t2)) = trav t1 @ (x :: trav t2)
```

For example, for the tree t on page 2, 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 say "x is in t" if x is a member of the list trav(t).

We prove the following theorem to build familiarity with the terms:

Theorem 1 *For all values* `T : tree`*,* `trav(T)` *evaluates to a list of length* `size(T)`*.*

Proof: By structural induction on `T`.

BASE CASE: `T = Empty`.

We must show that `trav(Empty)` evaluates to a list of length `size(Empty)`.

Observe that `trav(Empty) ==> [ ]` and that `size(Empty) ==> 0`, so the base case follows.

INDUCTIVE STEP: `T = Node(t1, x, t2)`.

Inductive Hypotheses: `trav(t1)` returns a list of length `size(t1)` and `trav(t2)` returns a list of length `size(t2)`.

Need to show: `trav(T)` returns a list of length `size(T)`.

Showing:

Let `L1` be the list `trav(t1)` and `L2` the list `trav(t2)`.
Let `n1` be the integer `size(t1)` and `n2` the integer `size(t2)`.

By definition of `size` we have

```
    size(T)
=   size(Node(t1, x, t2))
==> (size t1) + 1 + size(t2)
==> n1 + 1 + n2
```

By definition of `trav` we have

```
    trav(T)
=   trav(Node(t1, x, t2))
==> (trav t1) @ (x :: trav t2)
==> L1 @ (x :: L2)
```

which, by the inductive hypotheses, reduces to a list of length `n1 + 1 + n2`, as desired. □

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: (i) an empty tree is sorted and (ii) a non-empty tree is sorted if and only if its two subtrees are sorted, every integer in the left subtree is less-than-or-equal to the integer at the root, and every integer in the right subtree is greater-than-or-equal to the integer at the root.

We can implement an SML function for testing tree sortedness:

```
(* Sorted : tree -> bool
   REQUIRES: true
   ENSURES:  Sorted(T) evaluates to true if T is sorted and false otherwise.
*)
fun Sorted T = sorted(trav T)
```

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 a sense with which we are already familiar. 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 the list trav(t2).

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 lecture.

```
(* Ins : int * tree -> tree
   REQUIRES: t is a sorted tree
   ENSURES:  Ins(x,t) evaluates to a sorted tree such that
             trav(Ins(x,t)) is a sorted permutation of x::trav(t).
*)
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 the list function ins given earlier.

In what follows we take as a fact that Ins is total. That is easy to prove, using structural induction.

Theorem 2 *For all* `x : int` *and all sorted* `T : tree`*,* `trav(Ins(x,t))` $\cong$ `ins(x, trav t)`*.*

Proof: By structural induction on tree `T`.

BASE CASE: `T = Empty`.

Need to show that for every integer `x`,
`trav(Ins(x,Empty))` $\cong$ `ins(x, trav Empty)`.

By the definitions of `trav`, `Ins` and `ins` we see

```
    trav(Ins(x,Empty))
==> trav(Node(Empty,x,Empty))          [1st Ins clause]
==> trav(Empty) @ (x :: trav(Empty))   [2nd trav clause]
==> [ ] @ [x]                          [1st trav clause]
==> [x]                                [see append def]

    ins(x, trav Empty)
==> ins(x, [ ])                        [1st trav clause]
==> [x]                                [1st ins clause]
```

That establishes the base case.

INDUCTIVE STEP: `T = Node(t1, y, t2)`.
(Observe that `t1` and `t2` are sorted since `T` is.)

Inductive Hypotheses: For every integer `x`,
`trav(Ins(x,t1))` $\cong$ `ins(x, trav t1)` and
`trav(Ins(x,t2))` $\cong$ `ins(x, trav t2)`.

Need to show: For every integer `x`,
`trav(Ins(x,T))` $\cong$ `ins(x, trav T)`.

Showing:
Suppose `x` > `y` (the other cases are similar, as usual). Then:

```
  trav(Ins(x, Node(t1, y, t2))
≅ trav(Node(t1, y, Ins(x, t2))        [2nd Ins clause, x>y]
≅ (trav t1) @ (y :: trav(Ins(x, t2))) [2nd trav clause, Ins totality]
≅ (trav t1) @ (y :: ins(x, trav t2))  [IH]
≅ (trav t1) @ (ins(x, y :: trav t2))  [step, 2nd ins clause, x>y]
≅ ins(x, (trav t1) @ (y :: trav t2))  [T is sorted, x>y][1]
≅ ins(x, trav (Node(t1, y, t2)))      [2nd trav clause]
```

□

Reminder: The proof methodology we employed to establish the base case was of the form: If expression `e1` reduces to some value `v` and if expression `e2` also reduces to that same value `v`, then `e1` $\cong$ `e2`. Alternatively, we could have shown that `e1` $\cong$ `e2` by establishing a string of equivalences, as we did in the Inductive Step. Often a proof can be done either way, though one must be careful to employ the correct reasoning for the particular methodology one uses. In particular, it is false to say that `e1 ==> e1` if in fact no such reduction occurs, if merely `e1` $\cong$ `e2`. And establishing

[1] Technically, this step requires its own proof. In an exam, we would probably supply it as a separate fact or lemma.

that `e1` ≅ `e2` must itself be done with care, since one must ensure that the expressions of interest really do both have equivalent values or both loop forever or both raise the same exception. See the guide to extensional equivalence on the course's tools webpage for further advice.

3 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 items in the tree less-than-or-equal to the integer and items greater-than-or-equal to the integer, respectively. 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
   REQUIRES: t is sorted
   ENSURES:  SplitAt(x,t) evaluates to a pair (t1,t2) of sorted trees such that:
                 (a) for every Node(_,y,_) in t1, compare(y,x)<>GREATER,
             and (b) for every Node(_,y,_) in t2, compare(y,x)<>LESS,
             and (c) trav(t1)@trav(t2) is a sorted permutation of trav(t).
*)
fun SplitAt(x, Empty) = (Empty, Empty)
  | SplitAt(x, Node(left, y, right)) =
      (case compare(x, y) of
           LESS => let
                      val (t1, t2) = SplitAt(x, left)
                   in
                      (t1, Node(t2, y, right))
                   end
          |  _  => let
                      val (t1, t2) = SplitAt(x, right)
                   in
                      (Node(left, y, t1), t2)
                   end)
```

4 Merging Two Trees

The tree-based analog of merge is a function that takes a pair of sorted trees and combines their data into a single (also sorted) tree.

```
(* Merge : tree * tree -> tree
   REQUIRES: t1 and t2 are sorted
   ENSURES:  Merge(t1,t2) evaluates to a sorted tree t such that
             trav(t) is a sorted permutation of trav(t1)@trav(t2).
*)
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
```

5 Mergesort for Trees

Using Ins and Merge, and guided by their specs, we may now define a mergesorting function for integer trees.

```
(* Msort : tree -> tree
   REQUIRES: true
   ENSURES:  Msort(t) evaluates to a sorted tree such that
             trav(Msort(t)) is a sorted permutation of trav(t).
*)
fun Msort Empty = Empty
  | Msort (Node(t1, x, t2)) = Ins (x, Merge(Msort t1, Msort t2))
```

Compare and contrast this code with the mergesorting function msort for integer lists that we examined during the last lecture. This code no longer needs to do any significant work to split the input data before sorting recursively, since the tree structure already encapsulates a natural split (into left and right parts). However, it may be necessary now to perform some splitting when merging the results of the recursive calls.

6 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 state some intuitive results about depth, that will be helpful when we analyze the runtime behavior of our code. One may prove these results by induction.

1. For all trees t and integers x,

   $$\texttt{depth(Ins(x, t))} \le \texttt{depth(t) + 1}.$$

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

   $$\texttt{depth(t1)} \leq \texttt{depth(t)} \qquad \texttt{depth(t2)} \leq \texttt{depth(t).}$$

3. For all trees `t1` and `t2`,

   $$\texttt{depth(Merge(t1, t2))} \leq \texttt{depth(t1) + depth(t2).}$$

4. For all trees `t`,

   $$\texttt{depth(Msort t)} \leq 2^{\texttt{depth(t)}} - 1.$$

7 Size Analysis

We further state some intuitive results about size. Again, one may use a suitable inductive method to prove these facts.

1. For all trees `t` and integers `x`,

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

2. For all trees `t` and integers `x`, if `SplitAt(x,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).`

4. For all trees `t`,

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

8 Span Analysis

The overall work[2] of our `Msort` implementation is $O(n \log n)$, and the span is $O((\log n)^3)$. We now analyze the span in detail. We focus on span, since the purpose of using trees was to enable parallel-friendly sorting.

- The span for `Ins(x,t)` is `O`(d), where d is the depth of `t`. Reason: $\texttt{Ins}(\texttt{x}, \texttt{t})$ makes a single recursive call, on a subtree whose depth is $d-1$ or less. This produces the recurrence

  $$\begin{array}{rcll} S_{\texttt{Ins}}(0) & = & c_0 & \\ S_{\texttt{Ins}}(d) & \leq & c_1 + S_{\texttt{Ins}}(d_1) & \text{for } d > 0 \quad [\texttt{GREATER}] \\ S_{\texttt{Ins}}(d) & \leq & c_2 + S_{\texttt{Ins}}(d_2) & \text{for } d > 0 \quad [\texttt{otherwise}] \end{array}$$

  for some constants c_0, c_1, c_2, with both d_1 and d_2 no greater than $d-1$, and at least one of d_1 and d_2 exactly equal to $d-1$. This recurrence has a solution that is $O(d)$.

[2] Using the techniques from this course, you should be able to show that the work is $O(n\,(\log n)^2)$. A finer analysis using concavity of the log function can then establish that the work is in fact $O(n \log n)$. You will revisit this problem in 15-210, where you will see an algorithm with span $O((\log n)^2)$, better than what we have here.

- `SplitAt`$(\mathtt{x}, \mathtt{t})$ has span $\mathtt{O}(d)$, where d is the depth of `t`. Again the reason is that `SplitAt` makes a single recursive call, on a subtree whose depth is $d-1$ or less.
- `Merge`$(\mathtt{t1}, \mathtt{t2})$ has span $\mathtt{O}(d_1 d_2)$, where d_1 and d_2 are the depths of `t1` and `t2`, respectively. To see this, observe that `Merge` calls `SplitAt` on `t2` and `Merge` makes two (independent, i.e., parallelizable) recursive calls. The first arguments in these recursive calls are subtrees of `t1` with depths no greater than $d_1 - 1$. The second arguments are subtrees of `t2`, one of which might have depth d_2. There is a base case to the recurrence, and, after simplifying the max over the two recursive calls to `Merge`, the recursive case becomes:

  $$\begin{aligned} S_{\mathtt{Merge}}(d_1, d_2) &\leq k_0 + S_{\mathtt{SplitAt}}(d_2) + S_{\mathtt{Merge}}(d_1 - 1, d_2) \\ &\leq k_1 + k_2 * d_2 + S_{\mathtt{Merge}}(d_1 - 1, d_2) \end{aligned}$$

  for some constants k_0, k_1, and k_2. This recurrence has a solution that is $O(d_1 d_2)$.

- 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`. To see this, observe that `Msort` makes two (independent, i.e., parallelizable) recursive calls to subtrees of depth $d-1$ or less, followed by a call to `Merge` and a call to `Ins`. There is a base case to the recurrence, and, after simplifying the max over the two recursive calls to `Msort`, the recursive case becomes:

  $$\begin{aligned} S_{\mathtt{Msort}}(d) &\leq k + S_{\mathtt{Ins}}(2d) + S_{\mathtt{Merge}}(d-1, d-1) + S_{\mathtt{Msort}}(d-1) \\ &\leq c_0 + c_1 * d + c_2 * (d-1)^2 + S_{\mathtt{Msort}}(d-1) \\ &\leq c_0 + c_1 * d + c_3 * d^2 + S_{\mathtt{Msort}}(d-1) \end{aligned}$$

  for balanced trees of depth $d > 1$, for some constants k, c_0, c_1, and c_2. Expanding out, and observing that the sum of the first d squares is proportional to d^3, we deduce that the span is $\mathtt{O}(d^3)$. For a balanced tree, with size n and depth d, d is $\mathtt{O}(\log n)$. Our analysis therefore 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!

Caution: This would be true, except that we haven't justified our balance assumption. Indeed, there is a bug in the previous analysis. Even if we assume that the *original* tree passed to `Msort` is balanced, we cannot guarantee that the recursive sorting within `Msort` will produce balanced trees or that calling `Merge` on balanced trees will maintain balance. In fact, as written, `Msort` need not maintain balance. There is a simple fix: rebalance within `Msort`. This can be done in a way that does not affect the asymptotic work and span results, but we will not discuss details here. More generally, later in the course, we will discuss how to implement binary trees that maintain balance.

Comment:
There is a way to define a version of `Msort` 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))
```

Are `Msort` and `Msort’` extensionally equivalent?

Which of `Msort` and `Msort’` is more efficient?