15-150 Fall 2020

Lectures 7 and 8

A datatype of trees
Sorting a tree of integers

1 Outline

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

COMMENT
To avoid confusion I usually prefer to use the variable name `T` (and `U`, `T1`, `T2`) to range over trees (and `t` for types). Sometimes I use `L` and `R` for trees, even though we've mainly used them for lists so far; it's hard to resist using `L` for the left-child of a tree, for example, when I need a short variable name to fit on a slide! Please don't get confused; the context (and the needed type) should make it clear which one is intended.

2 Background

As before, as discussed previously, we refer to the type `order` and the integer comparison function:

```
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) = if x > y then y::ins(x, L) else 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 is 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` (a postfix operator), and values

```
Empty: 'a tree
Node: 'a tree * 'a * 'a tree -> 'a tree
```

which serve as constructors for building tree values. *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". We can use these constructors to build values of any type of the form `t tree`, but for now we will just use integer trees, of type `int tree`. For example, the four expressions

```
Empty
Node(Empty, 2, Empty)
Node(Empty, 42, Node(Empty, 2, Empty))
Node(Empty, 1+1, Empty)
```

all have type `int tree`. The first three of these expressions are values. The second one is the value of the fourth one.

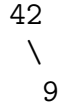
A value of type `int tree` represents a binary tree with integers at its nodes; the `Empty` tree value contains no integer data. Every non-empty tree has a piece of data at its root and two sub-trees or children, which may be empty.

The constructors can be used for pattern-matching against tree values. The pattern `Empty` only matches the value `Empty`, the empty tree. 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.

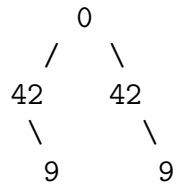
It is often convenient to draw pictures of tree values, rather than always using the ML syntax for tree expressions. In pictures we usually omit drawing Empty nodes explicitly. We draw the root node at the top, subtrees lower, left subtree to the left, and so on. For example, let `T` be the ML expression below:

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

`T` can be drawn (without showing `Empty` subtrees) as:



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



(For a tree this large the ML rendering of its value is going to be awkwardly long and nested.) All the "missing" `Empty` subtrees can easily be filled in; their positions are implied by the tree shape. (For example the picture for `T` is missing three `Empty`'s: the left child of the root node and the two children of the other node.)

Some simple ML code for building "full binary trees":

```
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 may refer to a tree like 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 evaluate `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 is `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` 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 tree values, and prove properties of 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(left,x,right)`, where `left` and `right` 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.

Let P be a property of trees. We can prove that "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 `left` and `right` are tree values such that P(`left`) and P(`right`) hold; show that for all integer values `x`, P(`Node(left,x,right)`) holds.

(iii) It follows from (i) and (ii) that $P(\texttt{T})$ 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 $F(\texttt{Empty})$.
- Give a clause defining $F(\texttt{Node}(\texttt{A}, \texttt{x}, \texttt{B}))$ in terms of $F(\texttt{A})$ and $F(\texttt{B})$.

Such clauses are sufficient to completely specify the intended value of $F(\texttt{T})$, 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. In fact 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".

size and depth

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 define an ML function `size : 'a tree -> int` that computes the size of a tree, using structural recursion:

```
fun size Empty = 0
 |  size (Node(left, _, right)) = size left + size right + 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 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`, $\texttt{size}(\texttt{T}) \geq 0$ and $\texttt{depth}(\texttt{T}) \geq 0$; and if $\texttt{T}'$ is a child of `T`, then $\texttt{depth}\ \texttt{T}' < \texttt{depth}\ \texttt{T}$ and $\texttt{size}\ \texttt{T}' < \texttt{size}\ \texttt{T}$. So we can also use *induction*

on tree depth, or *induction on tree size*, as techniques for proving properties of trees or 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, also using 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.

```
(* inord : 'a tree -> 'a list *)
fun inord Empty = [ ]
|   inord (Node(T1, x, T2)) = inord T1 @ (x :: inord T2)

(* ENSURES
      inord T = a list of the data at nodes of T,
                  as seen in an in-order traversal
*)
```

We say "`x` is in `T`" if `x` is a member of the list `inord(T)`.

For example, for the tree `T` above, `inord(T)=[42,9]`. And

```
inord(Node(T, 0, T)) = inord T @ (0 :: inord T)
                     = [42, 9] @ (0 :: [42, 9])
                     = [42, 9, 0, 42, 9].
```

We prove that for all trees `T`, `inord(T)` evaluates to a list of length `size(T)`. This is the same as saying "`size(T) = length(inord T)`".

Proof: by structural induction on `T`.

- Base case: When `T` is `Empty`, since `size(Empty) = 0` we must show that `inord Empty = [ ]`. This is obvious from the function definition.
- Inductive case: For `T` of form `Node(T1, x, T2)`, let `n1 = size T1` and `n2 = size T2`, so that `size(T)` is `n1+n2+1`. Since `T1` and `T2` are children of `T`, by the Induction Hypothesis for `T1` and `T2` we can assume that

  (i) `inord T1 = a list (say L1) of length n1`

  (ii) `inord T2 = a list (say L2) of length n2`.

  Then by definition of `inord` we have

```
inord (Node(T1, x, T2))
   = (inord T1) @ (x :: inordinord T2)
   = L1 @ (x :: L2)                  by (i) and (ii)
   = a list of length n1 + n2 + 1
   = a list of length size(Node(T1, x, T2))
```

  as needed [1].

4 Sorting trees

When is a tree sorted?

A tree is said to be *sorted* if at each node, the integer at that node is $\geq$ all integers in the left subtree and $\leq$ the integers in the right subtree. Here is an inductive way to characterize this notion of sortedness:

- The tree `Empty` is sorted.
- A non-empty tree value `Node(A,x,B)` is sorted if and only if every integer in `A` is $\leq$ `x`, every integer in `B` is $\geq$ `x`, and `A` and `B` are sorted.

For brevity we may sometimes write `A` $\leq$ `x` $\leq$ `B`, with `A` and `B` being tree values and `x` an integer.

Note how similar this definition is to how we could have defined sortedness for integer lists:

[1] We use the fact that `length(L1 @ (x::L2)) = length L1 + length L2 + 1`.

- The empty list is sorted.
- A non-empty list value `x::L` is sorted if and only if every integer in `L` is $\geq$ `x`, and `L` is sorted.

While it is possible to write an ML function for testing if a tree is sorted or not, we will not bother to do so here. We won't ever need to check for sortedness in implementing an algorithm for sorting trees; instead we'll design our code so that it is guaranteed to return a sorted tree, even without checking.

How to sort a tree

Our algorithm for sorting an integer tree can be described informally as follows:

- If the tree is empty, do nothing. It's sorted.
- Otherwise, (recursively) sort the two children, then merge the sorted children into a single sorted tree, and finally insert the root value into its correct position.

This suggests that we design helper functions for *inserting* an item into a sorted tree, and *merging* two sorted trees into one. Later we will see that the merging operation itself needs a helper function, for *splitting* a tree into two subtrees, using a given integer value to determine which items from the tree go into the first or second subtree. Also it will be important to think carefully about what assumptions it is safe to make about the arguments to be supplied to these helper functions.

Insertion for trees

The tree-based analogue of the insertion function `ins` on lists turns out to be just what we need, a truly helpful function in the code that follows. We use capitalization to distinguish this function from the `ins` function on lists.

```
(* Ins : int * int tree -> int tree       *)
(* REQUIRES T is a sorted tree    *)
(* ENSURES Ins(x, T) = a sorted tree consisting of x and all of T *)
```

```
fun Ins (x, Empty) = Node(Empty, x, Empty)
|   Ins (x, Node(T1, y, T2)) =
     if x>y then Node(T1, y, Ins(x, T2)) else Node(Ins(x, T1), y, T2)
```

Compare this code with the code for `ins`. See how similar it is.

We now show how to prove by structural induction that `Ins` satisfies this specification. Let $P(\texttt{T})$ be the property that

> For all integers `x`, if `T` is sorted, then `Ins(x,T)` evaluates to a sorted tree consisting of `x` and the items of `T`.

We prove "For all tree values `T`, $P(\texttt{T})$ holds", by structural induction on `T`.

- The base case is simple.

  $P(\texttt{Empty})$ holds, because `Empty` is sorted and `Ins(x, Empty)` evaluates to `Node(Empty, x, Empty)`. This is a tree value (because `x` is an integer value by assumption, and `Empty` is a tree value), is obviously sorted, and consists of just `x`, as required.

- For the inductive step we argue as follows.

  Suppose `T1` and `T2` are tree values such that $P(\texttt{T1})$ and $P(\texttt{T2})$ hold, and let `y` be an integer value. We show that $P(\texttt{Node(T1,y,T2)})$ holds. To do this, let `x` be an integer value and suppose `Node(T1, y, T2)` is sorted. We must show that `Ins(x, Node(T1, y, T2))` evaluates to a sorted tree value consisting of `x` and all items of `Node(T1, x, T2)`.

  We also know that `T1` and `T2` are sorted, and $\texttt{T1} \leq \texttt{y} \leq \texttt{T2}$. From the definition of `Ins` we see that

  (a) Either `x>y`, in which case we have

  ```
  Ins(x, Node(T1, y, T2)) =>* Node(T1, y, Ins(x, T2)).
  ```

  By induction hypothesis $P(\texttt{T2})$, `Ins(x, T2)` evaluates to a sorted tree (say `U2`) consisting of `x` and all of `T2`, so we get

  ```
  Ins(x, Node(T1, y, T2)) =>* Node(T1, y, U2),
  ```

  Clearly $\texttt{T1} \leq \texttt{y} \leq \texttt{U2}$, and `T1` and `U2` are sorted tree values, so this is a sorted tree value; it consists of `x` and `y` and all of `T1` and `T2`. Thus $P(\texttt{Node(T1,y,T2)})$ holds in this case.

  (b) Or $\texttt{x} \leq \texttt{y}$, in which case we have

```
Ins(x, Node(T1, y, T2)) =>* Node(Ins(x, T1), y, T2).
```

By induction hypothesis P(`T1`), `Ins(x, T1)` evaluates to a sorted tree (say `U1`) consisting of `x` and all of `T1`, so we get

```
Ins(x, Node(T1, y, T2)) =>* Node(U1, y, T2),
```

Clearly `U1` $\leq$ `y` $\leq$ `T2`, and `U1` and `T2` are sorted tree values, so this is a sorted tree value; it consists of `x` and `y` and all of `T1` and `T2`. Thus P(`Node(T1, y, T2)`) holds in this case also.

So P(`Node(T1, y, T2)`) holds (in all cases), as needed.

The above specification and proof use evaluational notation and evaluational reasoning. One can also state and prove an equational specification for `Ins`, using "value equations" derived from the function definition. It follows from the function definition that for all integer values `x` and `y`, and all tree values `T1` and `T2`, the following equations hold:

```
Ins(x, Empty) = Node(Empty, x, Empty)
Ins(x, Node(T1, y, T2)) = Node(T1, y, Ins(x, T2))   if x>y
Ins(x, Node(T1, y, T2)) = Node(Ins(x, T1), y, T2)   if x<=y
```

Just as above, where we dealt with evaluation to value, we can prove that

> For all integer values `x` and all sorted tree values `T`,
> there is a sorted tree value `S` consisting of `x` and all of `T`,
> such that `Ins(x, T) = S`.

To prove this result you can basically adapt the evaluational proof steps from above and write a corresponding equational proof.

Splitting a sorted tree

In adapting the mergesort algorithm to operate on trees we need a suitable analog to the `split` function. (Try to figure out how to merge two sorted trees. At some point you'll find the need to split a tree into two trees.) 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 (by analogy with the way the split function on lists worked). Instead, we will start from a tree and an integer, and break the tree into two trees, one consisting of items

less-or-equal to the integer and the other consisting of items greater than or equal to the integer. (There is some wiggle room here concerning where the items equal to this integer should go.) We will only ever need to use this method on a sorted tree, as you will observe when we develop the code. We also design the function so that when applied to a sorted tree it produces a pair of sorted trees. 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 * int tree -> int tree * int tree *)
(*
    REQUIRES T is sorted
    ENSURES SplitAt(x, T) = (A, B) where
                A and B are sorted trees containing all of T,
            and A <= x <= B.
*)

fun SplitAt(x, Empty) = (Empty, Empty)
 |  SplitAt(x, Node(T1, y, T2)) =
     if y > x
     then
          let
            val (L1, R1) = SplitAt(x, T1)
          in
            (L1, Node(R1, y, T2))
          end
      else (* x <= y *)
          let
             val (L2, R2) = SplitAt(x, T2)
          in
             (Node(T1, y, L2), R2)
          end
```

This function is structurally inductive, because in the recursive clause `SplitAt(x, Node(T1, y, T2))` calls `SplitAt(x, T1)` or `SplitAt(x, T2)`, in each case making a recursive call on a subtree. We prove that `SplitAt` satisfies this specification, by structural induction. To be precise, we prove "For all tree values `T`, $P(\texttt{T})$", where $P(\texttt{T})$ is the property that if `T` is sorted,

then for all values `x`, `SplitAt(x, T)` is equal to a pair of sorted trees `(A,B)` such that `A` $\leq$ `x` $\leq$ `B` and `A`, `B` contain the items of `T`.

- Base case: `Empty` is sorted and has no elements, so we need to show that for all values `x`, `SplitAt(x,Empty)` is equal to a pair of sorted trees with no elements. By definition we have `SplitAt(x, Empty)=(Empty,Empty)`, and the requirements hold trivially.

- Inductive step: Let `T` be a sorted tree of form `Node(T1, y, T2)`. We also know that `T1` and `T2` are sorted, and that `T1` $\leq$ `y` $\leq$ `T2`. Assume the Induction Hypothesis that `SplitAt` satisfies the spec on `T1` and on `T2`. We show that `SplitAt(x, T)` is equal to a pair of sorted trees with the required properties. There are two sub-cases to analyze, branching on the result of comparing the values of `x` and `y`.

  (a) If `y` $>$ `x` we have

  ```
  SplitAt(x, T) = (L1, Node(R1, y, T2))
  ```

  where `(L1, R1) = SplitAt(x, T1)`. By Induction Hypothesis, `L1` $\leq$ `x` $\leq$ `R1` and `L1`, `R1` are sorted trees consisting of the items from `T1`. So `L1` $\leq$ `x` $\leq$ `Node(R1,y,T2)` and `R1` $\leq$ `y` $\leq$ `T2`. And `Node(R1, y, T2)` is a sorted tree consisting of the items from `T1`, `y` and `L2`. Together with `L1` this covers all the items from `T`.

  (b) If `y` $\leq$ `x` we have

  ```
  SplitAt(x, T) = (Node(T1, y, L2), R2)
  ```

  where `(L2, R2) = SplitAt(x, T2)`.

  By induction hypothesis, `L2` $\leq$ `x` $\leq$ `R2` and `L2`, `R2` are sorted trees containing the items from `T2`. (Fill in the remaining details.)

That completes the proof.

Merging two sorted trees

Now the tree-based analog of `merge` on sorted integer lists: a function that takes a pair of sorted trees and combines them into a single (sorted) tree. We use `SplitAt` as a helper.

```
(* Merge : int tree * int tree -> int tree *)
(* REQUIRES T1 and T2 are sorted trees *)
(* ENSURES Merge(T1, T2) = a sorted tree
             consisting of the items from T1 and 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
```

The proof that `Merge` satisfies its specification relies on the fact that `SplitAt` satisfies 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! We claim that "For all sorted tree values `T`, P(`T`) holds", where P(`T`) is the property that

> For all sorted tree values `U`, `Merge(T, U)` is equal to a sorted tree value containing the items from `T` and `U`.

The proof is by structural induction on `T`. (Since sorted trees have sorted children, it is OK to do this kind of structural induction on sorted trees!)

- P(`Empty`) holds obviously. (Fill in the details.)
- For the inductive case, suppose `L1` and `R1` are sorted tree values for which P(`L1`) and P(`R1`) hold. Let `x` be an integer value and assume `Node(L1, x, R1)` is sorted. We must show that P(`Node(L1, x, R1)`) holds. Let `U` be a sorted tree value. By definition of `Merge` we have

  ```
  Merge(Node(L1, x, R1), U)
      = Node(Merge(L1, L2), x, Merge(R1, R2))
  ```

where `(L2, R2) = SplitAt(x, U)`. By the proven spec for `SplitAt` (which is applicable here because `U` is a sorted tree), $\texttt{L2} \leq \texttt{x} \leq \texttt{R2}$ and `L2`, `R2` are sorted and contain the items from `U`. By assumption that the original tree is sorted, we have $\texttt{L1} \leq \texttt{x} \leq \texttt{R1}$, and `L1`, `R1` are sorted. So by Induction Hypothesis, there are sorted tree values `L` and `R` such that

```
Merge(L1, L2) = L,  Merge(R1, R2) = R
```

and `L` contains the items from `L1`, `L2`, and `R` contains the items from `R1` and `R2`. Hence $\texttt{L} \leq \texttt{x} \leq \texttt{R}$ and `Node(L, x, R)` is a sorted tree, consisting of the items from `L1`, `x`, `R1`, `U`, as required.

That completes the proof. (We skipped over a few details – make sure you understand how to fill in the gaps.)

Lesson

It's important to notice how in the above code analysis the specs for the various helper functions play a crucial rôle. Just in the nick of time, it turned out we could appeal to an induction hypothesis, which was applicable *because we had shown that one of the helper functions behaved well.* We were careful to only use a helper function with arguments that satisfy the requirements of the helper spec, and in the proof details we confirmed that the guarantees made by the helper functions (when used in this manner) were sufficient to ensure that the rest of the code meets its own spec. If we hadn't shown that `SplitAt` preserves sortedness, we would have no basis for claiming that `Merge` preserves sortedness. The lesson is: choose your helper functions wisely, choose their specs wisely (with an eye to how you will use them), and nail down the correctness proof (at least with a sketch of the key details).

The tree-sorting function Msort

Using `Ins` and `Merge`, and guided by their (proven) specs, we are now ready to define a mergesorting function for integer trees. The hard work has already been done; now comes the easy and more immediately rewarding part! The tree sorting function in ML is defined in a way that mimics the algorithm we sketched earlier.

```
(*
   Msort : int tree -> int tree *)
   REQUIRES T is a value of type tree
   ENSURES Msort(T) = a sorted tree consisting of the items of T
*)

fun Msort Empty = Empty
|   Msort (Node(T1, x, T2)) = Ins (x, Merge(Msort T1, Msort T2))
```

Again the proof that `Msort` meets this spec uses the already proven facts 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.

5 Exploration

To illustrate how the various functions discussed above actually work on a specific tree example, try running the following code and drawing pictures of the tree values produced in each stage.

```
val T1 = Node(Leaf 3, 6, Leaf 1);
val T2 = Node(Leaf 5, 2, Empty);
val T  = Node(T1, 4, T2);

val M1 = Msort T1;
val M2 = Msort T2;
val M = Merge(M1, M2);

val S = Ins(4, M);
```

You might also want to try some examples using `SplitAt`.

Furthermore, it may be useful to define some functions for extracting a list containing the integers in a tree. In lab you will discuss *traversal lists* and in-order, pre-order and post-order traversal of trees. It turns out that an integer tree is sorted if and only if its in-order traversal list is a sorted list of integers. So one can easily check if a tree is sorted by looking at its in-order traversal list. Just for some irrelevant fun(?), try to figure out a

decent specification of what `SplitAt`, `Ins` and `Merge` do when applied to arguments that do NOT satisfy the REQUIRES assumptions used above.

6 Comments on sorting trees

For an integer list `L` there is just *one* sorted list that contains the same items as `L`. So it makes sense to talk about "computing *the* sorted version of `L`". In contrast, for a collection of at least 2 integers there can be multiple 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. It would be nice if we guaranteed to produce *balanced* trees, in which at each node the numbers of integers in the two child subtrees differ by at most 1. However, even if we start with a balanced (unsorted) tree, the functions that we have defined so far do not always produce balanced results. We will return to this point shortly.

7 Size analysis

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

(1) `size(Ins(x, T)) = size(T) + 1`

(2) If `SplitAt(y, T) = (T1, T2)`, then

`size(T1) + size(T2) = size(T).`

(3) `size(Merge(T1, T2)) = size T1 + size T2.`

(4) `size(Msort T) = size T.`

In each case, you can prove the result by structural induction.
Check that these results are consistent with the examples from before.

8 Depth analysis

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. [Of course, to do the proofs we would need access to the function definition for `depth`, which is given in lab.]

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

(2) If `SplitAt(y, T) = (T1, T2)`, then

`depth(T1) <= depth(T) and depth(T2) <= depth(T).`

(3) `depth(Merge(T1, T2)) <= depth T1 + depth T2`.

(4) `depth(Msort T) <= depth T`.

Check that these results are consistent with the examples from earlier.

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`$(\mathtt{x}, \mathtt{T})$ makes a single recursive call, on a subtree with depth decreased by 1.

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

- `Merge`$(\mathtt{T1}, \mathtt{T2})$ 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 their depth is about the logarithm (base 2) of their size, `Msort`$(\mathtt{T})$ 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`$(\mathtt{T})$ on balanced trees `T` 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 analysis! We assumed that the trees passed by `Msort` to `Merge` were balanced. However, this is not necessarily true: even if we assume 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, it isn't [2]. Our analysis really only predicts that the span can't be better than this bound, because we obtained this bound by making the most optimistic assumptions about tree structure.

Later we will discuss how to implement binary trees with an insertion operation guaranteed to build trees with a reasonable degree of balance. 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.

[2] Look at the examples from before: are the various trees constructed there balanced?

10 Self-test

1. A student pointed out that we can define a version of mergesort for trees 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))
   ```

   The idea is that Merging with a single node tree is essentially the same as inserting.

   (i) Is this function extensionally equivalent to the previous function?

   (ii) Does this function satisfy the same specification as before, i.e. does it still sort?

   (iii) Is it as efficient, or more efficient, asymptotically,than the previous version?

2. Write an ML function `leaves : int tree -> int list` such that for all tree values `T`, `leaves(T)` = the list of all integers occuring at leaf nodes of `T`. For example, `leaves(Full(42,3))=[42,42,42,42]`.

3. Let `T` be a tree with depth d. Show that the size of `T` at most $2^d - 1$?

4. Write an ML function `treesum : int tree -> int` such that for all tree values `T`, `treesum(T)` evaluates to the sum of the integers at the nodes of `T`. For example, `treesum(Full(42,3))` should evaluate to 294.

5. Write an Ml function `leafsum : int tree -> int` such that for all tree values `T`, `leafsum(T)` evaluates to the sum of the integers at leaf nodes of `T`. We interpret this quantity as 0 if the tree is `Empty`.
   Do not use `leaves` from above!

6. State and prove a theorem about the value of `treesum (Full(x, n))` when `n` is a non-negative integer.

7. How many different tree values `T` have exactly three nodes containing the integers 1,2,3? Of these, how many are sorted trees?

8. Define an ML function `balanced : int tree -> bool` such that for all tree values `T`, `balanced(T)=true` if at each node of `T` the sizes of the left and right children differ by at most 1. Otherwise `balanced(T)=false`.

9. Calculate the value produced by the following piece of code:

```
val T1 = Node(Leaf 1, 6, Leaf 3);
val T2 = Node(Empty, 5, Leaf 2);
val T = Node(T1, 4, T2);
Msort T;
```

10. Prove that `Msort` satisfies its specification. You can assume given proofs that `Merge` and `Ins` satisfy their specifications.

11. Let T_n be the tree value of the expression `Full`$(42, n)$, for $n \geq 0$. This is a full binary tree of depth n with 42 at each node. State and prove an assertion about the (shape of the) value of `Msort`(T_n).