Contents

1 Datatypes Lab

Welcome to Datatypes Lab! We've covered quite a few concepts over the past few weeks, and some of them may be a little hard to understand. In a previous semester, a 150 TA put together this visual guide to much of what we've covered so far, and we encourage you to give it a read!

1.1 Totality and Valuability

Definition (Valuable)**.** A well-typed expression `e` is *valuable* if there exists a value `v` such that `e` $\Longrightarrow$ `v`. We also say that `e` *evaluates to a value*.

Definition (Total)**.** A well-typed expression `e1 : t1 -> t2`, for types `t1` and `t2`, is *total* if for all values `e2 : t1`, the expression `e1 e2` is valuable.

1.1.1 A Totally Awesome Proof

Task 1.1.

```
fun swap (x : int, y : int) : int * int = (y, x)
```

Now, we claim that for all function values `g : int -> int` and all values `x, y : int`, that

```
swap (g x, g y) = (g y, g x)
```

Proof.

`swap (g x, g y)` $\cong$ `(g y, g x)` (A: definition of `swap`)

□

However, this is false! If we consider a function `g` such that `g 0` loops forever, but `g 1` raises an exception, then `swap(g 0, g 1)` loops forever, but `(g 1, g 0)` raises an exception! Where did we go wrong?

Solution 1.

We need to know that `g x` and `g y` are valuable before stepping through `swap`. The claim is true when we add the condition that g is total, which means that `g x` and `g y` would be valuable. Remember eager evaluation! Arguments to a function are evaluated before stepping into the function.

Task 1.2.

In a proof, what's the main purpose of citing the totality[1] of a function?

[1] Here's a totality help sheet for how to cite totality in proofs.

Solution 2.

The purpose of citing the totality of a function (e.g. `f`) is to help justify the valuability of an expression containing the application of said function (e.g. `f x`).

1.2 Lists and an Option

- An `int list` is either `nil` or cons (`::`) of an `int` and an `int list`
- An `int option` is either `NONE` or `SOME` of an `int`

1.3 Inductively-Defined Datatypes

Datatype declarations can inductively define datatypes with one or more *constructors*.

1.3.1 Some Built-Ins

There are lots of types which are built into SML. For example:

- The `int list` type:
- The `unit` type:
- The `int option` type:

1.3.2 Defining Datatypes

We can define our own datatypes using the `datatype` keyword:

1.4 Induction on Lists

Structural induction is a generalization of mathematical induction that involves inducting on the structure of a datatype in the following way:

- **Base Case(s):** Reason about the **non-recursive constructors** of the datatype (e.g. `[]`).
- **Inductive Case:** Reason about the **recursive constructors** of the datatype (e.g. `::`).

1.5 A List Proof

Task 1.3.

Let's see some totality in action.

```
fun [] @ R = R
  | (x::xs) @ R = x :: (xs @ R)

fun listSum ([] : int list) : int = 0
  | listSum (x::xs) = x + listSum xs
```

Prove for all values `L : int list` and all values `R : int list` that `listSum (L @ R)` ≅ `listSum L + listSum R`.

Proof. We will prove by structural induction.

□

Solution 3.

Base Case. `listSum ([] @ R)` ≅ `listSum [] + listSum R`

LHS:

```
listSum ([] @ R)
≅ listSum R                              (@ clause 1)
```

RHS:

```
listSum [] + listSum R
≅ 0 + listSum R                          (listSum clause 1)
≅ listSum R                              (listSum totality, math)
```

Induction Case. `listSum ((x::xs) @ R)` ≅ `listSum (x::xs) + listSum R`

Induction Hypothesis. `listSum (xs @ R)` ≅ `listSum xs + listSum R`

LHS:

```
listSum ((x::xs) @ R)
≅ listSum (x :: (xs @ R))                (@ clause 2)
≅ x + listSum (xs @ R)                   (listSum clause 2, totality of @)
≅ x + listSum xs + listSum R             (IH)
```

RHS:

```
listSum (x::xs) + listSum R
≅ x + listSum xs + listSum R             (listSum clause 2)
```

Note: We don't need to cite totality of `::` in the first step of the LHS, since constructors are total by definition. Thus, the totality of constructors can be assumed without citation in your proofs.

Note: We will *usually* provide the totality of `@` (or whatever function is relevant) as an additional lemma. If a lemma is not provided, you must prove the totality of the function.

1.6 Structural Induction

Structural induction inducts on the *structure* of a datatype.

Let's outline our base case, IH, and inductive case for the following theorems on this code:

Task 1.4.

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

fun treeSum (Empty : tree) : int = 0
  | treeSum (Node (L,x,R)) = treeSum L + x + treeSum R

fun inorder (Empty : tree) : int list = []
  | inorder (Node (L,x,R)) = inorder L @ (x :: inorder R)

fun listSum ([] : int list) : int = 0
  | listSum (x :: xs) = x + listSum xs
```

We want to prove that for all values `T : tree`, `treesum T` $\cong$ `listSum (inorder T)`.

- **Base Case**:
- **Inductive Hypothesis**:
- **Inductive Case**:

Solution 4.

- **Base Case**: Prove the theorem holds for `T = Empty`.
- **Inductive Hypothesis**: Given values `L, R : tree`, assume that `treeSum L` $\cong$ `listSum (inorder L)` and `treeSum R` $\cong$ `listSum (inorder R)`.
- **Inductive Case**: Given an arbitrary value `x : int`, prove the theorem holds for `T = Node (L, x, R)`.

Task 1.5.

```
datatype nat = Zero | Succ of nat

fun toNat (0 : int) : nat = Zero
  | toNat (x : int) : nat = Succ (toNat (x - 1))

fun toInt (Zero : nat) : int = 0
  | toInt ((Succ n) : nat)   = 1 + (toInt n)

fun natAdd (Zero : nat, m : nat) : nat = m
  | natAdd ((Succ n) : nat, m : nat)   = Succ (natAdd (n, m))

fun lazyAdd (n : nat, m : nat) : nat = toNat ((toInt n) + (toInt m))
```

We want to prove that for all values `n, m : nat`, `natAdd (n, m)` $\cong$ `lazyAdd (n, m)`.
Hint: What are you inducting over?

- **Base Case**:

- **Inductive Hypothesis**:

- **Inductive Case**:

Solution 5.

Fix `m : nat` as an arbitrary value.

- **Base Case**: Prove the theorem for `n = Zero`.
- **Inductive Hypothesis**: Given the value `n' : nat`, assume `natAdd (n', m)` $\cong$ `lazyAdd (n', m)`.
- **Inductive Case**: Prove the theorem for `n = Succ n'`.

2 Having `fun` is valuable!

In this section, we will explore totality and valuability in more detail.

Let `t1`, `t2`, `t3` be non-function types (e.g. `string`, `int list`, but not `int -> int` or `int list -> int`).

State whether each of the following statements are **true or false**. If you say a statement is true, give a justification. If you say a statement is false, give a counterexample.

For some of these problems, you will need to recall the `fact` function:

```
fun fact (0 : int) : int = 1
  | fact n = n * fact (n-1)
```

Task 2.1. (Recommended)

Let `f : t1 -> t2` be total. Then for all values `x : t1`, `f x` is valuable.

Solution 1.

True - by definition of totality.

Task 2.2. (Recommended)

The function `fact : int -> int` is total.

Solution 2.

False - it loops forever on negative inputs.

Task 2.3. (Recommended)

Let `f : t1 -> (t2 -> t3)` be total. Then for all values `x : t1`, `f x` is total.

Solution 3.

False - consider the function `fn x => fact`.

Task 2.4. (Recommended)

Let `f : t1 -> t2` be a value. Then `f` is valuable.

Solution 4.

True - by definition of valuability.

Task 2.5. (Recommended)

The function `fact : int -> int` is valuable.

Solution 5.

True - functions are values!

Task 2.6. (Recommended)

`fact x` is valuable for all nonnegative `x : int`.

Solution 6.

True - since every nonnegative input terminates and reduces to a value, and by definition of valuability.

Task 2.7. (Recommended)

Let `f : t1 -> t2` be a value. If for all values `x : t1`, `f x` is valuable, then `f` is total.

Solution 7.

True - by definition of totality.

3 To cite, or not to cite...

In this section, we will see examples of when and when not to use totality citations in proofs.

We've taken the following proof steps from a proof's inductive case of the following theorem:

For all values `L : int list`, `mult (add L)` ≅ `add (increase L)`

The IH assumes the theorem holds for some value `xs : int list` and the IC is showing the theorem holds for `x::xs`, given an arbitrary value `x : int`.

Here are the functions that these proof steps will be referencing:

```
fun add [] = []
  | add (x::xs) = (x + 1) :: add xs

fun mult [] = []
  | mult (x::xs) = (2 * x) :: mult xs

fun increase [] = []
  | increase (x::xs) = (2 * x + 1) :: increase xs
```

Task 3.1. (Recommended)

Does the following proof step need a totality citation? If so, what function needs to be total?

```
  mult (add (x::xs))
≅ mult ((x + 1) :: add xs)          (clause 2 of add)
```

Solution 1.

No totality citation needed - We are stepping through `add` and we can assume its argument, `x::xs`, is valuable since constructors are total by definition.

Task 3.2. (Recommended)

Does the following proof step need a totality citation? If so, what function needs to be total?

```
  mult ((x + 1) :: add xs)
≅ (2 * (x + 1)) :: mult (add xs)          (clause 2 of mult)
```

Solution 2.

We need the totality of `add` - before stepping through `mult`, we need to know that `(x + 1) :: add xs` is a value.

Task 3.3. (Recommended)

Does the following proof step need a totality citation? If so, what function needs to be total?

```
  (2 * (x + 1)) :: mult (add xs)
≅ (2 * (x + 1)) :: add (increase xs)         (IH)
```

Solution 3.

No totality citation needed - this step just uses the IH.

Task 3.4. (Recommended)

Does the following proof step need a totality citation? If so, what function needs to be total?

```
  (2 * (x + 1)) :: add (increase xs)
≅ ((2 * x + 1) + 1) :: add (increase xs)        (math)
```

Solution 4.

No totality citation needed - this step just uses math.

Task 3.5. (Recommended)

Does the following proof step need a totality citation? If so, what function needs to be total?

```
  ((2 * x + 1) + 1) :: add (increase xs)
≅ add ((2 * x + 1) :: increase xs)        (clause 2 of add)
```

Solution 5.

We need the totality of `increase` - When stepping backwards through code (from definition to function), consider the totality citations that would have been needed going the other way. In this case,

```
  add ((2 * x + 1) :: increase xs)
≅ ((2 * x + 1) + 1) :: add (increase xs)
                    (clause 2 of add, totality of increase)
```

needs the totality of `increase` because we need to know code(2 * x + 1) :: increase xs is a value before we can step through the definition of `add`.

Task 3.6. (Recommended)

Does the following proof step need a totality citation? If so, what function needs to be total?

```
  add ((2 * x + 1) :: increase xs)
≅ add (increase (x::xs))        (clause 2 of increase)
```

Solution 6.

No totality citation needed - To see why, once again think about totality citations for the opposite direction.

```
  add (increase (x::xs))
≅ add ((2 * x + 1) :: increase xs)        (clause 2 of increase)
```

4 Listen up!

Define the following functions in `code/lists/lists.sml`.

Task 4.1. (Recommended)

In `code/lists/lists.sml`, write a function

```
merge : int list * int list -> int list
```

REQUIRES: `l1` and `l2` are sorted

ENSURES: `merge (l1,l2)` $\Longrightarrow$ `l` where `l` is a sorted permutation of `l1 @ l2`

where sorted means nondescending order.

Solution 1.

```
fun merge ([] : int list, L2 : int list) : int list = L2
  | merge(L1, []) = L1
  | merge(x::xs, y::ys) =
      if x < y
      then x::merge(xs, y::ys)
      else y::merge(x::xs, ys)

val () = Test.int_list_eq("merge 1", [], merge([],[]))
val () = Test.int_list_eq("merge 2", [1,2], merge([1,2],[]))
val () = Test.int_list_eq("merge 3", [0,1,2,3,4,5,7,9], merge
    ([1,3,5,7,9],[0,2,4]))
```

Task 4.2. (Recommended)

Define a function

```
head : int list -> int option
```

REQUIRES: true

ENSURES: `head L` $\Longrightarrow$ `SOME x` if `L` is non-empty, where x is the first element, or `NONE` if `L` is empty

Solution 2.

```
fun head ([] : int list) : int option = NONE
  | head (x::_) = SOME x

val () = Test.int_option("head 1", NONE, head [])
val () = Test.int_option("head 2", SOME 1, head [1, 2, 3])
```

Task 4.3.

Prof. Brookes has decided to reward the TAs handsomely for their hard work. He wants to add a bonus \$3 to the February paycheck for each of the TAs. Write a function

```
credit : int list -> int list
```

REQUIRES: true

ENSURES: `credit L` $\Longrightarrow$ `L'` where each element is \$3 more

that takes in a list representing the amounts payable to each of the TAs for the month of February. Credit \$3 to each of the TAs and make their day!

Solution 3.

```
fun credit ([] : int list) : int list = []
  | credit (x::xs) = (x + 3)::credit xs

val () = Test.int_list_eq("credit 1", [], credit [])
val () = Test.int_list_eq("credit 2", [4, 5, 6], credit [1, 2, 3])
```

Task 4.4.

Write a function

```
evens : int list -> int list
```

REQUIRES: true

ENSURES: `evens L` $\Longrightarrow$ `L'` where all odd elements of a list are filtered out, but the order of the original list `L` is preserved

For example,

```
evens [0,0,4] = [0,0,4]
evens [] = []
evens [0,0,4,9,3,2] = [0,0,4,2]
```

Solution 4.

```
fun evens ([] : int list) : int list = []
  | evens (x::xs) =
      case x mod 2 of
           0 => x::evens xs
         | _ => evens xs

val () = Test.int_list_eq("evens 1", [], evens [])
val () = Test.int_list_eq("evens 2", [], evens [1, 3, 5])
val () = Test.int_list_eq("evens 3", [0, 2, 4], evens [0, 1, 2, 3, 4])
```

Task 4.5.

Write a function

```
lastPositive : int list -> int option
```

REQUIRES: true

ENSURES: `lastPositive L` $\Longrightarrow$ `SOME x` where `x` is the *last* positive number in the list, if it exists, or `NONE` if no such `x` exists

Solution 5.

```
fun lastPositive ([] : int list) : int option = NONE
  | lastPositive (x::xs) =
      case (x > 0, lastPositive xs) of
           (_, SOME y) => SOME y
         | (true, NONE) => SOME x
         | (false, NONE) => NONE

val () = Test.int_option("lastPositive 1", NONE, lastPositive [])
val () = Test.int_option("lastPositive 2", NONE, lastPositive [~1, ~2])
val () = Test.int_option("lastPositive 3", SOME 3, lastPositive [~1, 3])
```

```
val () = Test.int_option("lastPositive 4", SOME 3, lastPositive [~1, 3, ~2])
val () = Test.int_option("lastPositive 5", SOME 3, lastPositive [~1, 2, 3])
```

Task 4.6.

Define a function

`sequence : int option list -> int list option`

REQUIRES: true

ENSURES: `sequence L` $\Longrightarrow$ `NONE` if `L` contains at least one `NONE`, or `SOME [x1, x2, ..., xn]` if `L` is of the form `[SOME x1, SOME x2, ..., SOME xn]`

Solution 6.

```
fun sequence ([] : int option list) : int list option = SOME []
  | sequence (NONE::_) = NONE
  | sequence ((SOME x)::xs) =
      case sequence xs of
          NONE => NONE
        | SOME xs' => SOME (x::xs')

val () = Test.int_list_option("sequence 1", SOME [], sequence [])
val () = Test.int_list_option("sequence 1", NONE, sequence [SOME 1, SOME 2, NONE
    , SOME 4])
val () = Test.int_list_option("sequence 1", SOME [1, 2, 3], sequence [SOME 1,
    SOME 2, SOME 3])
```

Task 4.7.

In `code/lists/lists.sml`, write a function

> ```
> bitAnd : int list * int list -> int list
> ```
>
> REQUIRES: `A` and `B` only contain 1s and 0s
>
> ENSURES: `bitAnd (A,B)` $\Longrightarrow$ `l` where `l` is a list with a logical 'and' performed on the corresponding elements of the two lists interpreting 1 as *true* and 0 as *false*. If the end of either list is reached, we stop evaluating

For example,

```
bitAnd ([1],[1]) = [1]
bitAnd ([1,1,0],[0,1,0]) = [0,1,0]
bitAnd ([],[]) = []
bitAnd ([1,0,1],[1,1]) = [1,0]
```

Solution 7.

```
fun bitAnd ([] : int list, _ : int list) : int list = []
  | bitAnd (_ : int list, [] : int list) : int list = []
  | bitAnd (x::xs : int list, y::ys : int list) : int list = (x * y)::bitAnd (xs
    , ys)

val () = Test.int_list_eq("bitAnd 1", [0, 0, 0, 0], bitAnd ([1, 1, 0, 1], [0, 0,
     1, 0]))
val () = Test.int_list_eq("bitAnd 2", [1, 1, 0], bitAnd ([1, 1, 0], [1, 1, 0,
    1]))
val () = Test.int_list_eq("bitAnd 3", [], bitAnd ([], [1, 1, 1]))
```

Task 4.8.

In `code/lists/lists.sml`, write a function

> ```
> interleave : int list * int list -> int list
> ```
>
> REQUIRES: true
>
> ENSURES: `interleave (A,B)` $\Longrightarrow$ `l` where `l` is a list built by alternating between the elements of `A` and `B`, until we reach the end of one of the lists, after which we take the remaining elements from the other list

For example,

```
interleave ([2],[4]) = [2,4]
interleave ([2,3],[4,5]) = [2,4,3,5]
interleave ([2,3],[4,5,6,7,8,9]) = [2,4,3,5,6,7,8,9]
interleave ([2,3],[]) = [2,3]
```

Solution 8.

```
fun interleave ([] : int list, B : int list) : int list = B
  | interleave (A : int list, [] : int list) : int list = A
  | interleave (x::xs : int list, y::ys : int list) : int list = x::y::
    interleave(xs,ys)

val () = Test.int_list_eq("interleave 1", [1,3,2,4], interleave([1,2],[3,4]))
val () = Test.int_list_eq("interleave 2", [], interleave([],[]))
val () = Test.int_list_eq("interleave 3", [1,2], interleave([],[1,2]))
val () = Test.int_list_eq("interleave 4", [1,2], interleave([1,2],[]))
```

5 Proving Totality

Consider the function `foo : int list -> int list` given by

```
fun foo ([] : int list) : int list = []
  | foo (x::xs) =  x :: foo(rev xs)
```

where `rev` is a given function of type `int list -> int list`, such that, for all values `L : int list`, `rev(L)` evaluates to the reverse of list `L`.

Task 5.1.

Prove the following theorem by induction on the length of `L`:

Theorem: For all values `L : int list`, `foo L` evaluates to a value.

You may **NOT** make any assumptions about how `rev` is implemented. Instead, you may make use of the following lemmas:

Lemma 1: For all values `L : int list`, `rev L` evaluates to a value.

Lemma 2: If the value `L : int list` has length n, then `rev L` has length n.

Solution 1.

Proof. We will prove the theorem by induction on the length of `L`

Base Case: `L` has length 0, i.e. `L=[]`
To show: `L` evaluates to a value.

$$\texttt{foo []} \Longrightarrow \texttt{[]} \qquad \text{(defn. of } \texttt{foo}\text{)}$$

Since `foo (L)` evaluated to a value, this case is proven.

Inductive Case: `L` has length $k+1$ for some $k \in \mathbb{N}$.

Inductive Hypothesis: `foo(L')` evaluates to a value for all `L':int list` of length k.

To Show: `foo(L)` evaluates to a value for all `L:int list` of length $k+1$.

Let `L` be a list of length $k+1$. Since `L` has nonzero length, `L` is of the form `x::xs`, for some `x:int` and some `xs:int list`, where `xs` has length k.

By Lemma 1, `rev(xs)` evaluates to a value. Call this value `sx`. By Lemma 2, `sx` has length k (since `xs` has length k).

$$\begin{aligned}
\texttt{foo (x::xs)} &\Longrightarrow \texttt{x :: foo (rev xs)} && \text{(defn. of } \texttt{foo}\text{)} \\
&\Longrightarrow \texttt{x :: foo sx} && \text{(defn. of } \texttt{sx}\text{)} \\
&\Longrightarrow \texttt{x::v} && \text{(for some value } \texttt{v}\text{, by IH)}
\end{aligned}$$

Since `v` and `x` are values, `x::v` is a value, and we're done. □

6 Practice with Nats

We've defined a datatype to represent natural numbers in `code/multNats/multNats.sml`:

```
datatype nat = Zero | Succ of nat
```

`Zero : nat` represents the number 0, while `Succ Zero : nat` represents the number 1, and so on.

Write the function `natMult`, which multiplies `n : nat` and `m : nat`. Feel free to use use the `natAdd` function, but do not use `toInt` or `toNat`.

Task 6.1. (Recommended)

In `code/multNats/multNats.sml`, define the function

> `natMult : nat * nat -> nat`
>
> REQUIRES: true
>
> ENSURES: `natMult (n, m) = toNat (toInt n * toInt m)`

For example, `natMult (Succ (Succ Zero), Succ (Zero) = Succ (Succ Zero)`

Solution 1.

```
fun natMult (Zero : nat, m : nat) : nat = Zero
  | natMult (n, Zero) = Zero
  | natMult (Succ n, Succ m) = natAdd (Succ m, natMult (n, Succ m))
```

7 "Trees Are Never Sad Look At Them Every Once In Awhile They're Quite Beautiful"

—Jaden Smith (@jaden), 9/19/2013

We will take a look at some trees (notice, they are indeed quite beautiful).

Recall the definition of trees from lecture:

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

Recall from lecture that recursive functions on trees usually have the following form:

```
fun foo (Empty : tree) = ...
  | foo (Node(L, x, R)) = ...
```

that is, in order to specify a function by recursion on the structure of a tree, it suffices to specify the value of the function at `Empty` and the value of the function at `Node(L,x,R)` in terms of the value of the function on `L` and `R`.

For example, recall the function `size` which gives the number of `Node`s in a tree:

```
(*  size: tree -> int
 *  REQUIRES: true
 *  ENSURES: size T ==>* the number of nodes in T
 *)
 fun size (Empty : tree) : int = 0
   | size (Node(L, x, R)) = 1 + size L + size R
```

To practice writing tree functions, we'll have you implement the function `depth` which gives the max depth of a tree (the max depth of the empty tree is 0, and the max depth of a nonempty tree is the maximum depth of all the nodes in a tree):

Task 7.1. (Recommended)

In `code/depth/depth.sml`, define

> `depth : tree -> int`
>
> REQUIRES: true
>
> ENSURES: `depth T` $\Longrightarrow$ the maximum depth of `T`

For example,

```
val () = Test.int("Empty tree", 0, depth Empty)
val () = Test.int("Singleton tree", 1, depth (Node (Empty, 1, Empty)
   ))
val () = Test.int("Mini tree", 2, depth (Node (Empty, 4, Node (Empty
   , 5, Empty))))
```

> **Solution 1.**

```
fun depth (Empty : tree) : int = 0
  | depth (Node (L, _, R)) = 1 + Int.max (depth L, depth R)
```

```
val () = Test.int("Empty tree", 0, depth Empty)
val () = Test.int("Singleton tree", 1, depth (Node (Empty, 1, Empty)))
val () = Test.int("Mini tree", 2, depth (Node (Empty, 4, Node (Empty, 5, Empty))
    ))
```

Task 7.2.

Now we'll write a more complex function to operate on trees.

We define the *leaves* of a tree to be nodes which have the form `Node (Empty, x, Empty)` for some `x : int`.

Define the function:

`leaves : tree -> int list`

REQUIRES: true

ENSURES: `leaves T` $\Longrightarrow$ the values at the leaves of `T`

For example,

```
val E = Empty
val T = Node (E, 3, E)
val T' = Node (Node (E, 1, E), 2, T)
val () = Test.int_list_eq("Empty tree leaves", [], leaves E)
val () = Test.int_list_eq("Singleton leaves", [3], leaves T)
val () = Test.int_list_eq("Mini tree leaves", [1 , 3], leaves T')
```

Because we don't care about the order of the returned list, your solution may behave differently than the examples in that regard. You are allowed to use standard library functions, such as `@`.

Solution 2.

```
fun leaves (Empty : tree) : int list = []
  | leaves (Node (Empty, x, Empty)) = [x]
  | leaves (Node (L, _, R)) = leaves L @ leaves R
```

```
val E = Empty
val T = Node (E, 3, E)
val T' = Node (Node (E, 1, E), 2, T)
val () = Test.int_list_eq("Empty tree leaves", [], leaves E)
val () = Test.int_list_eq("Singleton leaves", [3], leaves T)
val () = Test.int_list_eq("Mini tree leaves", [1 , 3], leaves T')
```

8 Trees

Here is a function similar to the flatten function from lecture, which computes an in-order traversal of a tree:

```
fun treeToList (Empty : tree) : int list = []
  | treeToList (Node (l,x,r)) = treeToList l @ (x :: (treeToList r))
```

In this problem, you will define a function to mirror, or invert a tree, so that the in-order traversal of the inversion comes out backwards:

`treeToList (invert T)` $\cong$ `rev (treeToList T)`

Task 8.1.

In `code/invert/invert.sml`, define the function

`invert : tree -> tree`

REQUIRES: true

ENSURES: `treeToList (invert T) = rev (treeToList T)`

For example,

```
val () = Test.general_eq("Mini tree", Node(Node(Empty,1,Empty),5,
    Node(Empty,0,Empty)), invert (Node(Node(Empty,0,Empty),5,Node(
    Empty,1,Empty)))
```

Solution 1.

```
fun invert (Empty : tree) : tree = Empty
  | invert (Node (t1,x,t2)) = Node (invert t2, x , invert t1)
```

```
val () = Test.general_eq("Empty tree", Empty, invert Empty)
val () = Test.general_eq("Singleton", Node(Empty,1,Empty), invert (Node(Empty,1,
    Empty))
val () = Test.general_eq("Mini tree", Node(Node(Empty,4,Empty),8,Node(Empty,15,
    Empty)), invert (Node(Node(Empty,15,Empty),8,Node(Empty,4,Empty)))
```

Task 8.2.

Prove that `invert` is total.

Solution 2.

Proof. We want to show that `invert` is total. This means that we must show that for all values `t : tree`, there exists some value `v : int` such that `invert t` $\Longrightarrow$ `v`.

Note that since our theorem uses reduction (i.e. "steps to" or $\Longrightarrow$), we may not use extensional equivalence (i.e. $\cong$), as it is a weaker statement.

We proceed by structural induction on `t : tree`.

Case (`t = Empty`):
To Show: `invert t` evaluates to a value
`invert t` $\Longrightarrow$ `[]` by clause 1 of `invert`, which is a value, so we are done.

Case (`t = Node(l, x, r)`), for some values `l : tree`, `x : int`, and `r : tree`.
Inductive Hypothesis: Suppose that `invert l` evaluates to a value and that `invert r` evaluates to a value.
To Show: `invert T` evaluates to a value.

`invert t` $\Longrightarrow$	`invert Node (l, x, r)`	(case assumption)
$\Longrightarrow$	`Node (invert r, x, invert l)`	(clause 2 of `invert`)
$\Longrightarrow$	`Node (r', x, l')`	(for some values `r'`, `l'` by our IH)

which is a value, and so we are done.

Hence by structural induction, `invert` is total. □