You should hand in attempts to the questions indicated by (*) to your tutor at the start of each tutorial. Showing effort at answering the indicated questions will contribute to the 4% “Tutorial Preparation” component of the course; your attempts will not be marked for correctness.

1 Induction on Lists

1.1 An Easy One (*)

We are now all familiar with the append operator for joining two lists together. We would probably agree that it is associative:

\[ xs ++ (ys ++ zs) = (xs ++ ys) ++ zs \]

Prove this property using structural induction.

1.2 Arguing by Cases

The examples in the lecture all use a recipe where we simplify repeatedly using equations that come from the function definitions, until we prove the equality required.

In the following example, you will need to do some case analysis in the proof. It is part of the exercise to work out what the cases are.

\[ \text{elem } z (xs ++ ys) = \text{elem } z xs || \text{elem } z ys \]

Prove this property of lists using structural induction.

1.3 A Really Hard One

It may seem obvious, but when you define an operation reverse as follows,

\[
\begin{align*}
\text{reverse } [] &= [] \\
\text{reverse } (x : xs) &= \text{reverse } xs ++ [x]
\end{align*}
\]

it is hard to prove that:
reverse (reverse xs) = xs.

Determine where the difficulty is. Can you find a way around the problem - a lemma, perhaps? .. or maybe a different definition of reverse.

Does this definition of reverse contain more nested recursions than a more efficient definition? Will a proof about reverse correspondingly contain more nested inductions than a proof about a different definition of list reversal?

Don’t spend too long thinking about this one.

2 Induction on Trees

2.1 Reverse

What does it mean to reverse a binary tree?
The following is probably a definition that we could agree on:

\[ \text{revT} :: \text{Tree } a \rightarrow \text{Tree } a \]
\[ \text{revT} \, \text{Nul} = \text{Nul} \quad \text{-- T1} \]
\[ \text{revT} \, (\text{Node } x \, t1 \, t2) = \text{Node } x \, (\text{revT} \, t2) \, (\text{revT} \, t1) \quad \text{-- T2} \]

Again we will expect that the following is true. So, prove it!

\[ \text{revT} \, (\text{revT} \, t) = t \]

Additionally, prove the following:

\[ \text{count} (\text{revT} \, t) = \text{count} \, (t) \]

2.2 Flattening

We can turn a tree into a list containing the same entries with the tail recursive function flat, where \( \text{flat} \, t \, \text{acc} \) returns the result of flattening the tree \( t \), appended to the front of the list \( \text{acc} \). Thus, for example,

\[ \text{flat} \, (\text{Node } 5 \, (\text{Node } 3 \, \text{Nul} \, \text{Nul}) \, (\text{Node } 6 \, \text{Nul} \, \text{Nul})) \, [1,2] = [3,5,6,1,2] \]

\[ \text{flat} :: \text{Tree } a \rightarrow [a] \rightarrow [a] \]
\[ \text{flat} \, \text{Nul} \, \text{acc} = \text{acc} \quad \text{-- (F1)} \]
\[ \text{flat} \, (\text{Node } a \, t1 \, t2) \, \text{acc} = \text{flat} \, t1 \, (a : \text{flat} \, t2 \, \text{acc}) \quad \text{-- (F2)} \]

We can get the sum of entries in a list by the function sumL

\[ \text{sumL} :: [\text{Int}] \rightarrow \text{Int} \]
\[ \text{sumL} \, [] = 0 \quad \text{-- (S1)} \]
\[ \text{sumL} \, (x : xs) = x + \text{sumL} \, xs \quad \text{-- (S2)} \]
We can get the sum of entries in a tree by the function \( \text{sumT} \)

\[
\begin{align*}
\text{sumT} :: \text{Tree \ Int} \to \text{Int} \\
\text{sumT} \ \text{Nil} & = 0 \quad -- \ (T1) \\
\text{sumT} \ (\text{Node} \ n \ t1 \ t2) & = n + \text{sumT} \ t1 + \text{sumT} \ t2 \quad -- \ (T2)
\end{align*}
\]

Prove by structural induction on the structure of the tree argument, that for all \( t \) and \( \text{acc} \),

\[
\text{sumL} \ (\text{flat} \ t \ \text{acc}) = \text{sumT} \ t + \text{sumL} \ \text{acc}
\]

3 Induction with Functions of Multiple Variables

The two issues that often crop up when proving theorems about such functions are:

- It is often not clear what variable to do induction on.
- The beginner may not get the inductive hypothesis right. One has to remember that the other variables are still implicitly universally quantified.

The following function is one that successively takes elements from the front of one list and puts them onto the front of a second list.

\[
\begin{align*}
\text{slinky} :: \ [a] \to \ [a] \to \ [a] \\
\text{slinky} \ [] & = \text{ys} \quad -- \ S1 \\
\text{slinky} \ (x:xs) \ ys & = \text{slinky} \ xs \ (x:ys) \quad -- \ S2
\end{align*}
\]

For example, \((\text{slinky} \ [1,2] \ [3,4]) = [2,1,3,4]\).

Each of the following equations are theorems about the \text{slinky} function

\[
\begin{align*}
(a) \ & \text{slinky} \ (\text{slinky} \ xs \ ys) \ zs = \text{slinky} \ ys \ (xs ++ zs) \\
(b) \ & \text{slinky} \ xs \ (\text{slinky} \ ys \ zs) = \text{slinky} \ (ys ++ xs) \ zs \\
(c) \ & \text{slinky} \ xs \ (ys ++ zs) = \text{slinky} \ xs \ ys \ ++ \ zs
\end{align*}
\]

3.1 Proving Property (a)

- Take it as given that we do induction on \( xs \) and check that this makes the base case trivial.
- The step case is now

\[
\text{slinky} \ (\text{slinky} \ (x:xs) \ ys) \ zs = \text{slinky} \ ys \ ((x:xs) ++ zs)
\]

- Attack the step case using an instance of

\[
\forall ys,zs. \ \text{slinky} \ (\text{slinky} \ xs \ ys) \ zs = \text{slinky} \ ys \ (xs ++ zs)
\]
3.2 Do one yourself

Prove lemma (b).

Prove lemma (c).

3.3 Reverse, again

Can we use slinky to do the very hard Question 1.3? (Hint: describe, in words, what does slinky do?)
4 Appendix: Function definitions

\[\text{count :: Tree } a \rightarrow \text{ Int}\]
\[\text{count Nul} = 0 \quad \text{-- C1}\]
\[\text{count (Node } x \ t_1 \ t_2) = 1 + \text{count } t_1 + \text{count } t_2 \quad \text{-- C2}\]

\[\text{length :: } [a] \rightarrow \text{ Int}\]
\[\text{length } [] = 0 \quad \text{-- L1}\]
\[\text{length } (x:xs) = 1 + \text{length } xs \quad \text{-- L2}\]

\[\text{map :: } (a \rightarrow b) \rightarrow [a] \rightarrow [b]\]
\[\text{map } f [] = [] \quad \text{-- M1}\]
\[\text{map } f (x:xs) = f x : \text{map } f xs \quad \text{-- M2}\]

\[\text{([], ++) :: } [a] \rightarrow [a] \rightarrow [a]\]
\[\text{[]} ++ ys = ys \quad \text{-- A1}\]
\[(x:xs) ++ ys = x : (xs ++ ys) \quad \text{-- A2}\]

\[\text{elem :: Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{ Bool}\]
\[\text{elem } y [] = \text{False} \quad \text{-- E1}\]
\[\text{elem } y (x:xs)\]
\[\quad | x == y = \text{True} \quad \text{-- E2}\]
\[\quad | \text{otherwise} = \text{elem } y xs \quad \text{-- E3}\]

\[\text{(||) :: Bool \rightarrow Bool \rightarrow Bool}\]
\[\text{True || } _ = \text{True} \quad \text{-- O1}\]
\[\text{False || } x = x \quad \text{-- O2}\]

\[\text{reverse :: } [a] \rightarrow [a]\]
\[\text{reverse []} = [] \quad \text{-- R1}\]
\[\text{reverse } (x:xs) = \text{reverse } xs ++ [x] \quad \text{-- R2}\]