Department of Computer Science, Australian National University
COMP2600 / COMP6260 — Formal Methods in Software Engineering
Semester 2, 2015

Week 3 Tutorial

Structural Induction

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.

```
elem z (xs ++ ys) = elem z xs || 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,

```
reverse [] = []
reverse (x : xs) = reverse xs ++ [x]
```

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:

```
revT :: Tree a -> Tree a
revT Nil            = Nil                              -- T1
revT (Node x t1 t2) = Node x (revT t2) (revT t1)    -- T2
```

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

```
revT (revT t) = t
```

Additionally, prove the following:

```
count(revT t) = count (t)
```

2.2 Flattening

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

```
flat (Node 5 (Node 3 Nul Nul) (Node 6 Nul Nul)) [1,2] = [3,5,6,1,2]
```

```
flat :: Tree a -> [a] -> [a]
flat Nul acc            = acc                          -- (F1)
flat (Node a t1 t2) acc = flat t1 (a : flat t2 acc)    -- (F2)
```

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

```
sumL :: [Int] -> Int
sumL []     = 0            -- (S1)
sumL (x:xs) = x + sumL xs  -- (S2)
```

We can get the sum of entries in a tree by the function sumT

```
sumT :: Tree Int -> Int
sumT Nul            = 0                      -- (T1)
sumT (Node n t1 t2) = n + sumT t1 + sumT t2 -- (T2)
```

Prove by structural induction on the structure of the tree argument, that for all `t` and `acc`,

```
sumL (flat t acc) = sumT t + sumL 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.

```
slinky :: [a] -> [a] -> [a]
slinky []     ys = ys                              -- S1
slinky (x:xs) ys = slinky xs (x:ys)                -- S2
```

For example, `(slinky [1,2] [3,4]) = [2,1,3,4]`.

Each of the following equations are theorems about the `slinky` function

```
(a)  slinky (slinky xs ys) zs = slinky ys (xs ++ zs)
(b)  slinky xs (slinky ys zs) = slinky (ys ++ xs) zs
(c)  slinky xs (ys ++ zs)     = slinky xs ys ++ zs
```

3.1 Proving Property (a)

- Take it as given that we do induction on `xs` and check that this makes the base case is trivial.
- The step case is now
  `slinky (slinky (x:xs) ys) zs = slinky ys ((x:xs) ++ zs)`
- Attack the step case using an instance of
  $\forall ys, zs.$ `slinky (slinky xs` ys`)` zs `= 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

```
count :: Tree a -> Int
count Nil              = 0                              -- C1
count (Node x t1 t2)   = 1 + count t1 + count t2        -- C2

length :: [a] -> Int
length []       = 0                                     -- L1
length (x:xs)   = 1 + length xs                         -- L2

map :: (a -> b) -> [a] -> [b]
map f []        = []                                    -- M1
map f (x:xs)    = f x : map f xs                        -- M2

(++) :: [a] -> [a] -> [a]
[]     ++ ys    = ys                                    -- A1
(x:xs) ++ ys    = x : (xs ++ ys)                        -- A2

elem :: Eq a => a -> [a] -> Bool
elem y []       = False                                 -- E1
elem y (x:xs)
    | x == y    = True                                  -- E2
    | otherwise = elem y xs                             -- E3

(||) :: Bool -> Bool -> Bool
True  || _       = True                                 -- O1
False || x       = x                                    -- O2

reverse :: [a] -> [a]
reverse []      = []                                    -- R1
reverse (x:xs)  = reverse xs ++ [x]                     -- R2
```