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.

We prove $\forall xs.P(xs)$ by structural induction, where $P(xs) =$ `xs ++ (ys ++ zs) = (xs ++ ys) ++ zs` and treat `ys` and `zs` as constants.

Base Case

```
[] ++ (ys ++ zs)        = ([] ++ ys) ++ zs

[] ++ (ys ++ zs)        = ys ++ zs                        -- by A1
                        = ([] ++ ys) ++ zs                -- by A1
```

Step Case

Assume

```
   as ++ (ys ++ zs)     = (as ++ ys) ++ zs              -- (IH)
```

Prove

```
  (a:as) ++ (ys ++ zs)  = ((a:as) ++ ys) ++ zs

  (a:as) ++ (ys ++ zs)  = a : (as ++ (ys ++ zs))        -- by A2
                        = a : ((as ++ ys) ++ zs)        -- by IH
                        = (a : (as ++ ys)) ++ zs        -- by A2
                        = ((a:as) ++ ys) ++ zs          -- by A2
```

Having showed $P(as) \rightarrow P(a : as)$, we can generalise to $\forall x.\forall xs.P(xs) \rightarrow P(x : xs)$. Having showed $P([])$ and $\forall x.\forall xs.P(xs) \rightarrow P(x : xs)$, we have $\forall xs.P(xs)$.

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.

We prove $\forall xs.P(xs)$ by structural induction on xs where $P(xs = \forall xs.$ `elem z (xs ++ ys) = elem z xs || elem z ys`.

<u>Base Case</u>

```
elem z ([] ++ ys)       = elem z [] || elem z ys

elem z ([] ++ ys)       = elem z ys                    -- by A1
                        = False || elem z ys           -- by O2
                        = elem z [] || elem z ys       -- by E1
```

<u>Step Case</u>

Assume

```
elem z (as ++ ys)       = elem z as || elem z ys       -- (IH)
```

Prove

```
elem z ((a:as) ++ ys)   = elem z (a:as) || elem z ys
```

- Case `z == a`

```
elem z ((a:as) ++ ys)   = elem z (a : (as ++ ys))      -- by A2
                        = True                         -- by E2
                        = True || elem z ys            -- by O1
                        = elem z (a:as) || elem z ys   -- by E2
```

- Case `z /= a`

```
elem z ((a:as) ++ ys)   = elem z (a : (as ++ ys))      -- by A2
                        = elem z (as ++ ys)            -- by E3
                        = elem z as || elem z ys       -- by IH
                        = elem z (a:as) || elem z ys   -- by E3
```

Having showed $P(as) \rightarrow P(a : as)$, we can generalise to $\forall x.\forall xs.P(xs) \rightarrow P(x : xs)$. Having showed $P([])$ and $\forall x.\forall xs.P(xs) \rightarrow P(x : xs)$, we have $\forall xs.P(xs)$.

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.

The 'hard structural induction' really is quite hard. It's worth preparing one's self for it. The main point is for them to actually see that even some simple theorems are hard to show mathematically. It would be rewarding if you had students who could solve it by themselves, though.

What they should successfully do on their own is reduce the step case goal to

```
reverse (reverse as ++ [a]) = (a : as)
```

before getting quite stuck.

Either of the following two lemmas, which one can easily prove by induction, can revive this stuck proof.

- `reverse (ys ++ [a]) = a :  (reverse ys)`
- `reverse (as ++ ys) = reverse ys ++ reverse as`

As far as alternative definitions of reverse that might make the problem directly soluble, the other standard defn is shown here. It's still difficult, you use several lemmas. These are explored below under the material relating to `slinky`

```
reverse x = reverse2 x []
where reverse2 []     ys = ys
      reverse2 (x:xs) ys = reverse2 xs (x:ys)
```

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 Nul            = Nul                              -- 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
```

We prove $\forall t.P(t)$ by induction (on t) where $P(t) =$ `revT (revT t) = t`.

<u>Base Case</u>

```
revT (revT Nul)         = Nul

revT (revT Nul)         = revT Nul          -- by T1
                        = Nul               -- by T1
```

<u>Step Case</u>

Assume

```
revT (revT a1)          = a1                                -- (IH1)
revT (revT a2)          = a2                                -- (IH2)
```

Prove

```
revT (revT (Node x a1 a2)) = Node x a1 a2

revT (revT (Node x a1 a2)) = revT (Node x (revT a2) (revT a1))        -- by T2
                           = Node x (revT (revT a1)) (revT (revT a2)) -- by T2
                           = Node x a1 a2                          -- by IH1, IH2
```

Having showed $P($`a1`$) \wedge P($`a2`$) \to P($`Node x a1 a2`$)$, we can generalise to $\forall y.\forall t1.\forall t2.P($`t1`$) \wedge P($`t2`$) \to P($`Node y t1 t2`$)$. Having showed $P($`Nul`$)$ and $\forall y.\forall t1.\forall t2.P($`t1`$) \wedge P($`t2`$) \to P($`Node y t1 t2`$)$, we have $\forall t.P(t)$.

Additionally, prove the following:

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

We prove $\forall t.P(t)$ by induction on t where $P(t) =$ `count(revT t) = count (t)`.

<u>Base Case</u>

```
count(revT Nul) = count Nul

count(revT Nul)        = count Nul          -- by T1
```

Step Case

Assume

```
    count(revT a1) = count a1                              -- (IH1)
    count(revT a2) = count a2                              -- (IH2)
```

Prove

```
count (revT (Node x a1 a2)) = count (Node x a1 a2)
```

```
count (revT (Node x a1 a2))
  = count (Node x (revT a2) (revT a1))        -- by T2
  = 1 + count (revT a2) + count (revT a1)     -- by C2
  = 1 + (count a2) + (count a1)               -- by IH2, IH1
= 1 + (count a1) + (count a2)               -- by commutativity of +
= count (Node x a1 a2)                      -- by C2
```

Having showed $P(\texttt{a1}) \wedge P(\texttt{a2}) \rightarrow P(\texttt{Node x a1 a2})$, we can generalise to $\forall y.\forall t1.\forall t2.P(\texttt{t1}) \wedge P(\texttt{t2}) \rightarrow P(\texttt{Node y t1 t2})$. Having showed $P(\texttt{Nul})$ and $\forall y.\forall t1.\forall t2.P(\texttt{t1}) \wedge P(\texttt{t2}) \rightarrow P(\texttt{Node y t1 t2})$, we have $\forall t.P(t)$.

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
```

Base case: `t = Nul`.

Show that `sumL (flat Nul acc) = sumT Nul + sumL acc`

Proof:

```
sumL (flat Nul acc) = sumL acc -- by (F1)
= 0 + sumL acc                 -- by arithmetic
= sumT Nul + sumL acc          -- by (T1)
```

The inductive hypotheses: ∀`acc`

```
sumL (flat t1 acc) = sumT t1 + sumL acc -- (IH1)
sumL (flat t2 acc) = sumT t2 + sumL acc -- (IH2)
```

Step case:

Show (assuming the IHs), that ∀`acc`,

```
sumL (flat (Node y t1 t2) acc =
sumT (Node y t1 t2) + sumL acc
```

Proof:

```
sumL (flat (Node y t1 t2) acc)
= sumL (flat t1 (y : flat t2 acc)) -- by (F2)
= sumT t1 + sumL (y : flat t2 acc) -- by (IH1)
= sumT t1 + y + sumL (flat t2 acc) -- by (S2)
= sumT t1 + y + sumT t2 + sumL acc -- by (IH2)
= sumT (Node y t1 t2) + sumL acc   -- by (T2)
```

Note: (IH1) is instantiated so that `acc` becomes `(y : flat t2 acc)`, but the `acc` of (IH2) is just instantiated to the `acc` of the proof.

Note: the proof involves steps which use associativity or commutativity of +; these steps are not explicitly shown. Additionally, the formal generalisation step is omitted.

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`)`

Do this one by induction on `xs`, need to use

`P(xs) = ∀ys. slinky (slinky xs ys) zs = slinky ys (xs ++ zs)`

Base Case

```
slinky (slinky [] ys) zs      = slinky ys ([] ++ zs)

slinky (slinky [] ys) zs      = slinky ys zs              -- by S1
                              = slinky ys ([] ++ zs)      -- by A1
```

Step Case

Assume `∀ys.  slinky (slinky as ys) zs = slinky ys (as ++ zs) -- (IH)`

Prove `∀ys.  slinky (slinky (a:as) ys) zs = slinky ys ((a:as) ++ zs)`

```
slinky (slinky (a:as) ys) zs  = slinky (slinky as (a:ys)) zs  -- by S2
                              = slinky (a:ys) (as ++ zs)      -- by IH (*)
                              = slinky ys (a:(as ++ zs))      -- by S2
                              = slinky ys ((a:as) ++ zs)      -- by A2
```

`(*)` Note, ys in the IH is instantiated to $a : ys$ when it is used in the proof

3.2 Do one yourself

Prove lemma (b).

Do this one by induction on `ys`, need to use

`P(ys) = ∀zs.  slinky xs (slinky ys zs) = slinky (ys ++ xs) zs`

Base Case

```
slinky xs (slinky [] zs)          = slinky ([] ++ xs) zs

slinky xs (slinky [] zs)          = slinky xs zs            -- by S1
                                  = slinky ([] ++ xs) zs    -- by A1
```

Step Case

Assume `∀zs.  slinky xs (slinky as zs) = slinky (as ++ xs) zs -- (IH)`

Prove `∀zs.  slinky xs (slinky (a:as) zs) = slinky ((a:as) ++ xs) zs`

```
slinky xs (slinky (a:as) zs) = slinky xs (slinky as (a:zs)) -- by S2
                             = slinky (as ++ xs) (a:zs)     -- by IH (*)
                             = slinky (a:(as ++ xs)) zs     -- by S2
                             = slinky ((a:as) ++ xs) zs     -- by A2
```

(*) Note, zs in the IH is instantiated to $a : zs$ when it is used in the proof

Prove lemma (c).

Do this one by induction on `xs`, need to use

`P(xs) = ∀ys.  slinky xs (ys ++ zs) = slinky xs ys ++ zs`

Base Case

```
slinky [] (ys ++ zs)          = slinky [] ys ++ zs

slinky [] (ys ++ zs)          = ys ++ zs                 -- by S1
                              = slinky [] ys ++ zs       -- by S1
```

Step Case

Assume `∀ys.  slinky as (ys ++ zs) = slinky as ys ++ zs -- (IH)`

Prove `∀ys.  slinky (a:as) (ys ++ zs) = slinky (a:as) ys ++ zs`

```
slinky (a:as) (ys ++ zs)   = slinky as (a:(ys ++ zs))   -- by S2
                           = slinky as ((a:ys) ++ zs)   -- by A2
                           = slinky as (a:ys) ++ zs     -- by IH (*)
                           = slinky (a:as) ys ++ zs     -- by S2
```

(*) Note, ys in the IH is instantiated to $a : ys$ when it is used in the proof

3.3 Reverse, again

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

Take:

```
reverse xs  = slinky xs []                                -- (Slinkify)
```

Rewrite:

```
reverse (reverse xs)          = slinky (slinky xs []) []   -- by Slinkify
                              = slinky [] (xs ++ [])       -- by 3.1a
                              = xs ++ []                   -- by S1
                              = xs                         -- easy lemma
```

4 Appendix: Function definitions

```
count :: Tree a -> Int
count Nul              = 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
```