The Australian National University
School of Computing
Dirk Pattinson and Victor Rivera

Semester 2, 2021
Practical Session 4

Foundations of Computation

The practical contains a number of exercises designed for the students to practice the course content. During the practical session, the tutor will work through some of these exercises while students will be responsible for completing the remaining exercises in their own time. There is no expectation that all the exercises will be covered in the practical session.

Covers: Lecture Material Week 4

At the end of this tutorial, you will be able to prove programs by induction on lists and trees.

Exercise 1 — Associativity of List Concatenation

We know that list concatenation is associative, i.e that

$$\texttt{xs ++ (ys ++ zs) = (xs ++ ys) ++ zs}$$

holds for all lists `xs`, `ys` and `zs`. Prove this using list induction. Precisely state

- The property that you are proving, including all quantifiers
- What you need to show for the base case and in the inductive step
- What you are assuming as induction hypothesis.

Solution.

We show that

$$\forall \texttt{xs}.\forall \texttt{ys}.\forall \texttt{zs}.\texttt{xs ++ (ys ++ zs) = (xs ++ ys) ++ zs}$$

by list induction. That is, we use the induction rule to show

$$\forall \texttt{xs}.P(\texttt{xs})$$

where

$$P(\texttt{xs}) = \forall \texttt{ys}.\forall \texttt{zs}.\texttt{xs ++ (ys ++ zs) = (xs ++ ys) ++ zs}$$

An alternative (easier) way would be to treat ys and zs as constants and show that

$$\forall \texttt{xs}.\texttt{xs ++ (ys ++ zs) = (xs ++ ys) ++ zs}$$

and then argue that – since ys and zs were arbitrary, this holds for all ys and zs.

Base Case. We show that

$$P([\,]) = \forall \texttt{ys}.\forall \texttt{zs}.\texttt{[] ++ (ys ++ zs) = ([] ++ ys) ++ zs}$$

Let zs and ys be arbitrary lists. Then

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

Step Case. We show that $\forall \texttt{x}.\forall \texttt{xs}.P(\texttt{xs}) \to P(\texttt{x} : \texttt{xs})$. So let x and xs be arbitrary and assume the inductive hypothesis

$$\forall \texttt{ys}.\forall \texttt{zs}.\texttt{xs ++ (ys ++ zs) = (xs ++ ys) ++ zs} \qquad (IH)$$

For arbitrary ys and zs we then need to show that

```
(x:xs) ++ (ys ++ zs)     = ((x:xs) ++ ys) ++ zs
```

We argue as follows:

```
(x:xs) ++ (ys ++ zs)     = x : (xs ++ (ys ++ zs))        -- by A2
                         = x : ((xs ++ ys) ++ zs)        -- by IH
                         = (x : (xs ++ ys)) ++ zs        -- by A2
                         = ((x:xs) ++ ys) ++ zs          -- by A2
```

which finishes the proof.

Exercise 2 **Tree Induction**

Consider the definition of binary trees given in the lectures

```
  data Tree a = Nul | Node (Tree a) a (Tree a)
```

and the following two functions:

```
size :: Tree a -> Int
size Nul          = 0                                    -- C1
size (Node l x r) = 1 + size l + size r                  -- C2

mirror :: Tree a -> Tree a
mirror Nul          = Nul                                -- M1
mirror (Node l x r) = Node (mirror r) x (mirror l)       -- M2
```

`size t` counts the number of nodes in `t` and `mirror t` obtains the mirrored tree `t`.

Establish, using structural induction, that `mirror` preserves the size of the tree.

Solution.

We establish that the following

```
        P(t) = (size t = size (mirror t))
```

holds for all trees `t` by structural induction.

Base case: `P(Nul)`. We show that


```
                size Nul = size (mirror Nul)
```

by means of the following calculation:


```
        size (mirror Nul)
= size Nul                              -- by M1
```

Step case: We show that

$$\forall \mathtt{t1}.\forall \mathtt{x}.\forall \mathtt{t2}.P(\mathtt{t1}) \wedge P(\mathtt{t2}) \rightarrow P(\mathtt{Node\ t1\ x\ t2})$$

We separate the induction hypothesis for the left and the right subtree:

```
  size (mirror t1) = size t1            -- (IH1)
  size (mirror t2) = size t2            -- (IH2)
```

Proof Goal. For arbitrary `a`, show that `P (Node t1 a t2)`, i.e

```
        size (mirror (Node t1 x t2)) = size (Node t1 x t2)
```

The proof is as follows:

```
        size (mirror (Node t1 x t2))
= size (Node (mirror t2) x (mirror t1))         -- by M2
= 1 + size (mirror t2) + size (mirror t1)       -- by C2
= 1 + size (mirror t1) + size (mirror t2)       -- (we assume integers are commutative)
= 1 + size t1 + size (mirror t2)                -- by IH1
= 1 + size t1 + size t2                         -- by IH2
= size (Node t1 x t2)                           -- by C2
```

which finishes the proof.

Exercise 3 **Euclid's Algorithm**

We can formulate Euclid's Algorithm in Haskell as follows:

```
euclid :: Int -> Int -> Int
euclid n 0 = n
euclid 0 m = m
euclid n m = euclid (max n m - min n m) (min n m)
```

Our aim is to show that `euclid` terminates for all inputs $n, m > 0$.

1. Define a *termination measure*, that is, a function $t : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $t(n', m') < t(n, m)$ where n' and m' are the arguments of a recursive call to `euclid` in the definition of `euclid n m`, whenever the recursive call would be evaluated.
2. Prove that your termination measure t, defined above, indeed has this property.

Solution. We put $t(n, m) = n + m$. For a recursive call, we then need to show that

$$t(\max(n, m) - \min(n, m), \min(n, m)) < t(n, m)$$

and we do this by distinguishing cases.

Case 1: $n \geq m$. Then

$$t(\max(n, m) - \min(n, m), \min(n, m)) = t(n - m, m) = n - m + m = n < n + m = t(n, m)$$

as $m > 0$ (as otherwise we would be in the base cse).

Case 2: $n < m$. Then similarly

$$t(\max(n, m) - \min(n, m), \min(n, m)) = t(m - n, n) = m - n + n = m < m + n = t(n, m)$$

as $n > 0$ (as otherwise we would be in the other base case).

Exercise 4 **A simple but useful fact**

Show that `xs ++ [] = xs` holds for all lists `xs`, using list induction.

Solution.

We have to show that $\forall$`xs.xs ++ [] = xs`.

Base Case. We show `[] ++ [] = []`. But this is precisely the definition of `A1` for `ys = []`.

Step Case. We show that $\forall$`xs.`$\forall$`x.xs ++ [] = xs` $\to$ `(x:xs) ++ [] = x:xs`. So let `x` and `xs` be arbitrary and assume the induction hypothesis

`xs ++ [] = xs.` $\quad (IH)$

We show that

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

This follows, as

```
 (x:xs) ++ [] = x:(xs ++ [])    -- by A2
 = x:xs              -- by IH
```

Exercise 5 **List Reversal and Concatenation**

Consider the following definition of list reversal:

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

The aim of this exercise is to show that list reversal interacts with concatenation in the following way:

$\forall$xs.$\forall$ys.`reverse (xs ++ ys) = reverse ys ++ reverse xs.`

Use list induction to establish that `reverse (xs ++ ys) = reverse ys ++ reverse xs` for all lists `xs` and `ys`. You may find it helpful to use associativity of concatenation as well as other properties that we have proved in earlier exercises. Precisely state

- The property that you are proving, including all quantifiers
- What you need to show for the base case and in the inductive step
- What you are assuming as induction hypothesis.

Solution.

We show that $\forall$xs.$\forall$ys.`reverse (xs ++ ys) = reverse ys ++ reverse xs`

Base Case. We show that $\forall$ys.`reverse ([] ++ ys) = reverse ys ++ reverse []`. Let ys be arbitrary. Then

```
reverse ([] ++ ys) = reverse (ys)               -- by A1
= reverse ys ++ []                              -- Ex4
= reverse ys ++ reverse []                      -- by R1
```

as we had to show.

Step Case. We let x and xs be arbitrary and assume that

$\forall$ys.`reverse (xs ++ ys) = reverse ys ++ reverse xs` (IH)

and show that

$\forall$ys.`reverse ((x:xs) ++ ys) = reverse ys ++ reverse (x:xs)`

So let ys be arbitrary. We have


```
reverse ((x:xs) ++ ys) = reverse (x:(xs ++ ys))         -- by A2
= reverse (xs ++ ys) ++ [x]                             -- by R2
= (reverse ys ++ reverse xs) ++ [x]                     -- by IH
= reverse ys ++ (reverse xs ++ [x])                     -- assoc (Ex1)
= reverse ys ++ reverse (x:xs)                          -- by R2
```


and the equality is proven.

Exercise 6 **Arguing by Cases**

Show that

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

holds for all lists xs and ys and all z. Precisely state

- The property that you are proving, including all quantifiers
- What you need to show for the base case and in the inductive step
- What you are assuming as induction hypothesis.

You will want to argue by cases.

Solution.

We use list induction to show

$$\forall \texttt{xs}.\underbrace{\forall \texttt{ys}.\forall \texttt{z}.\texttt{elem z (xs ++ ys) = elem z xs || elem z ys}}_{P(\texttt{xs})}.$$

Base Case. We show $P([])$, that is

$\forall$ys.$\forall$z.`elem z ([] ++ ys) = elem z [] || elem z ys`

We let ys and z be arbitrary to obtain:

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

Step Case. We assume that

$$P(\texttt{xs}) = \forall \texttt{ys}.\forall \texttt{z}.\texttt{elem z (xs ++ ys) = elem z xs || elem z ys} \qquad (IH)$$

and show $P(\texttt{x : xs})$, that is,

$$\forall \texttt{ys}.\forall \texttt{z}.\texttt{elem z ((x:xs) ++ ys) = elem z (x:xs) || elem z ys.}$$

We distinguish the following cases for arbitrary ys and z:

- Case `z == x`

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

- Case `z /= x`

```
elem z ((x:xs) ++ ys)   = elem z (x : (xs ++ ys))        -- by A2
                        = elem z (xs ++ ys)              -- by E3
                        = elem z xs || elem z ys         -- by IH
                        = elem z (x:xs) || elem z ys     -- by E3
```


Exercise 7 **More Properties**

Consider the following function `slinky` that you may recognise from an earlier exercise:


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


Prove each of the following equations

```
(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
```

by list induction, and state explicitly

- The property that you are proving, including all quantifiers
- What you need to show for the base case and in the inductive step
- What you are assuming as induction hypothesis.

Hint. Think about precisely which variable is the variable you should induct on.

Solution.

1. We show that

$$\forall \texttt{xs}.\forall \texttt{ys}.\forall \texttt{zs}.\texttt{slinky (slinky xs ys) zs = slinky ys (xs ++ zs).}$$

Base Case. Show that

$$\forall \texttt{ys}.\forall \texttt{zs}.\texttt{slinky (slinky [] ys) zs = slinky ys ([] ++ zs).}$$

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

Step Case. Let `x` and `xs` be arbitrary and assume the inductive hypothesis

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

We have to show that

```
∀ys.∀zs.slinky (slinky (x:xs) ys) zs = slinky ys ((x:xs) ++ zs).
```

```
slinky (slinky (x:xs) ys) zs
= slinky (slinky xs (x:ys)) zs     -- S2
= slinky (x:ys) (xs ++ zs)         -- IH
= slinky ys (x:(xs ++ zs))         -- S2
= slinky ys ((x:xs) ++ zs)         -- A2
```

2. Here, we consider `xs` as a constant, and show that the property

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

holds for all `ys`.

Base Case. Show that

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

This is just a matter of unfolding equations:

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


Step Case.

Assume the inductive hypothesis for an arbitrary list `as` in place of `ys`:

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


Show that

```
∀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

3. Here, we treat `zs` as a constant and establish that the property

$P(\texttt{xs}) = \forall\texttt{ys}.$ `slinky xs (ys ++ zs) = slinky xs ys ++ zs`

holds for all lists `xs`. We do this by induction on `xs`.

Base Case. Show that

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

We prove this by unfolding equations:

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

Step Case.

Assume the inductive hypothesis

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

Prove that, for any `a`,

```
∀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
```

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

Exercise 8 **More Efficient List Reversal**

Consider the following, more efficient, version of list reversal:

```
rev_a [] ys = ys                      -- RA1
rev_a (x:xs) ys = rev_a xs (x:ys)     -- RA2

rev2 xs = rev_a xs []                 -- RR
```

The aim of this exercise is to show that ∀xs.`reverse xs = rev2 xs`. Notice that `rev2` is defined in terms of `rev_a` and that the second (accumulating) argument of `rev_a` changes in the recursive call.

1. Find a property that describes the relationship between `reverse` and `rev_a` where the second argument is an explicit variable. This property will have the form

   ∀xs.∀ys.`... reverse xs ... ys ... = ... rev_a xs ys ...` .

2. Establish this property by list induction.

3. Use the validity of this property to establish the original goal, that is, ∀xs.`reverse xs = rev2 xs`.

   Hint. In the proof, you may (and probably want to) use some of the equations that have been established in earlier exercises. Also note that `[x]` is just a notation for the list `x:[]`.

Solution.

1. We find the property

   ∀xs∀ys.`reverse xs ++ ys = rev_a xs ys`.

2. We show the above property by list induction.

   Base Case. Show that ∀ys.`rev_a [] ys = reverse [] ++ ys`.

   Let ys be arbitrary. Then we have

```
rev_a [] ys
= ys                   -- RA1
= [] ++ ys             -- A1
= reverse [] ++ ys     -- R1
```

   Step Case. We show that

   ∀xs.∀x(∀ys.`rev_a xs ys = reverse xs ++ ys`) → (∀ys.`rev_a (x:xs) ys = reverse (x:xs) ++ ys`)

   So let x and xs be arbitrary and assume that

   ∀ys.`rev_a xs ys = reverse xs ++ ys` (IH).

   We show that this property also holds for `x:xs` in place of `xs`. Let `ys` be arbitrary.

```
rev_a (x:xs) ys
= rev_a xs (x:ys)           -- RA2
= reverse xs ++ (x:ys)      -- IH
= reverse xs ++ x:([] ++ ys) -- A1
= reverse xs ++ ([x] ++ ys)  -- A2
= (reverse xs ++ [x]) ++ ys  -- assoc (Ex1)
= reverse (x:xs) ++ ys       -- R2
```

which finishes the proof.

3. We now show that `reverse xs = rev2 xs` for all lists `xs`.

```
rev2 xs
= rev_a xs []         -- RR
= reverse xs ++ []    -- as we have shown in part 2
= reverse xs          -- Ex4
```

Exercise 9 **Double List Reversal**

Consider the definition of list reversal given in the previous exercise, i.e. the function

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

The aim of this exercise is to show that

`reverse (reverse xs) = xs.`

As for the other exercises, precisely state

- The property that you are proving, including all quantifiers
- What you need to show for the base case and in the inductive step
- What you are assuming as induction hypothesis.

Hint. In the proof, you may (and probably want to) use some of the equations that have been established in earlier exercises. Also note that `[x]` is just a notation for the list `x:[]`.

Solution.

We prove $\forall$**xs**.`reverse (reverse xs) = xs`.

Base Case. We show that `reverse (reverse []) = []`:

```
  reverse (reverse [])
= reverse []              -- R1
= []                      -- R1
```


Step Case. We show that

$\forall$**xs**.$\forall$**x**.((`reverse (reverse xs) = xs`) $\rightarrow$ (`reverse (reverse (x:xs)) = x:xs`))

So let `x` and `xs` be arbitrary. We assume that

`reverse (reverse xs) = xs` (IH)

and show that the same property also holds with `x:xs` in place of `xs`. The argument is the following:

```
  reverse (reverse (x:xs))
= reverse ((reverse xs) ++ [x])          -- R2
= reverse [x] ++ reverse (reverse xs)    -- Ex5
= reverse (x:[]) ++ xs                   -- IH
= (reverse [] ++ [x]) ++ xs              -- R2
= ([] ++ [x]) ++ xs                      -- R1
= (x:[]) ++ xs                           -- A1
= x:([] ++ xs)                           -- A2
= x:xs                                   -- A1
```

Appendix: Function definitions

```
(++) :: [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

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

pref :: a -> [a] -> [a]
pref x l = x:l                                          -- P
```