Induction practice problems
COMP 302

Jacob Thomas Errington

Fall 2018

This document contains a number of theorems to prove by structural induction. Recall the general structure of a proof by induction:

1. Sometimes, it is necessary to generalize the theorem to prove, in order to obtain a stronger induction hypothesis.
2. Begin the proof (of the generalized theorem) by stating on what it is a proof by induction, e.g. "proof: by structural induction on `l`" where `l` refers to a list.
3. State the base case, e.g. "**case** `l = []`." Then prove the theorem under the assumption that `l` is the empty list.
4. State the step case, e.g. "**case** `l = x :: xs`." State the induction hypothesis: since `xs` is the sublist, the IH should specifically involve `xs`. Prove the theorem under the assumption that `l = x :: xs`, likely making use of the induction hypothesis.
5. Finally, once all cases are covered (for lists there are just two) restate the theorem, saying that it is proven, e.g. "Therefore, `P` holds by induction" where `P` is statement of the generalized theorem.
6. If you had to generalize a specific theorem (e.g. because you needed to abstract for all accumulators or something like that) then use the generalized theorem to prove the specific theorem. State the specific theorem, and prove it, e.g. "proof: by the previous theorem, choose `acc = 0`, then ..." Usually the derivation in this proof will be very short because the generalized theorem is doing all the work.

The structure of this document is the following.

- On page 2 we give the definitions of the functions that will be involved in the exercises.
- On page 3 we give the specific theorems to prove. These are *not* the generalized statements. Try to come up with the generalized version (if necessary) by trying to directly prove the specific theorem and seeing where the proof breaks down. Maybe the proof won't break down and you can prove the theorem directly!
- On page 4 we give the generalized theorems. Look here if you get stuck trying to generalize a specific theorem, but still want to give the proof of the generalized theorem a shot.
- Starting on page 5 we give the complete solution proofs.

1 Definitions

The theorems focus mainly on lists, so recall a few definitions of functions that will be important.

```
let rec map (f : 'a -> 'b) (l : 'a list) : 'b list =
  match l with
  | [] -> []
  | x :: xs -> f x :: map f xs

let rec length (l : 'a list) : int =
  match l with
  | [] -> 0
  | _ :: xs -> 1 + length xs

let rec sum_tr (l : int list) (a : int) : int =
  match l with
  | [] -> a
  | x :: xs -> sum_tr xs (a + x)

let rec fold_left (f : 'b -> 'a -> 'b) (l : 'a list) (a : 'b) : 'b =
  match l with
  | [] -> a
  | x :: xs -> fold_left f xs (f a x)

let rec fold_right (f : 'a -> 'b -> 'b) (l : 'a list) (e : 'b) : 'b =
  match l with
  | [] -> e
  | x :: xs -> f x (fold_right f xs e)
```

2 Specific theorems

Theorem 1 (Mapping identity is identity)**.** *Let* `id = fun x -> x`*. Then, for any* `l : 'a list`*, we have* `map id l = l`*.*

Theorem 2 (Map preserves length)**.**
For any `l : 'a list` *and any function* `f : 'a -> 'b`*, we have* `length l = length (map f l)`*.*

Theorem 3 (Map distributes over composition)**.**
Suppose `f : 'b -> 'c`*,* `g : 'a -> 'b`*, and* `l : 'a list`*.*
Then `map (fun x -> f (g x)) l = map f (map g l)`*.*

Theorem 4. *For any* `l : int list`*, we have that* `sum_tr l 0 = fold_right (+) l 0`*.*

The following theorem is a generalization of theorem 4. After proving theorem 5, see how you would use it to write an alternate proof for theorem 4. *Hint: you may need to prove that* `sum_tr` *is equal to a left fold.*

Theorem 5.
Suppose that `f : 'a -> 'a -> 'a` *is associative, i.e. for any* `x, y, z : 'a` *we have*
`f x (f y z) = f (f x y) z`
(For example, addition is such an operator.)

Suppose further that there is `e : 'a` *such that for any* `x : 'a`*, we have*
`f e x = f x e = x`
(In the case of addition, `e` *is* `0`*.)*

Then, for any list `l : 'a list`*, we have that*

```
fold_right f l e = fold_left f l e
```

3 Generalized theorems

Theorems 1, 2, and 3 do not require generalization.

Generalized theorem 1 (Of theorem 4)**.**
For any `l : int list` *and any* `a : int`*, we have* `sum_tr l a = a + fold_right (+) l 0`.

Generalized theorem 2 (Of theorem 5)**.**
For any accumulator `a : 'a` *and any list* `l : 'a list`*, we have that*

```
f a (fold_right f l e) = fold_left f l a
```

4 Solutions

Theorem. *Let* `id = fun x -> x`. *Then, for any* `l : 'a list`, *we have* `map id l = l`.

Proof. By structural induction on `l`.

Case `l = []`.

Then `map id [] = []` by definition of `map`.

Case `l = x :: xs`.

To show: `map id (x :: xs) = x :: xs`.

IH: `map id xs = xs`

`LHS = map id (x :: xs)`	
`= id x :: map id xs`	by definition of `map`.
`= x :: map id xs`	by definition of `id`.
`= x :: xs = RHS`	by induction hypothesis.

We transformed the LHS into the RHS, so this case is proven.

Therefore, by induction, for any `l`, we have `map id l = l`. □

Theorem (Map preserves length.).
For any `l : 'a list` *and any function* `f : 'a -> 'b`, *we have* `length l = length (map f l)`.

Proof. By structural induction on `l`.

Case `l = []`.

To show: `length [] = length (map f [])`.

`LHS = length [] = 0`	by definition of `length`.
`RHS = length (map f []) = length [] = 0.`	by definitions of `map` and `length`.

The LHS and RHS agree, so this case is proven.

Case `l = x :: xs`.

To show: `length (x :: xs) = length (map f (x :: xs))`.

IH: `length xs = length (map f xs)`.

`LHS = length (x :: xs) = 1 + length xs`	by definition of `length`.
`RHS = length (map f (x :: xs))`	
`= length (f x :: map f xs)`	by definition of `map`.
`= 1 + length (map f xs)`	by definition of `length`.
`= 1 + length xs`	by induction hypothesis.

Therefore the LHS and RHS agree, so this case is proven.

Therefore by induction, for any `l` and `f`, `length l = length (map f l)`. □

Theorem (Map distributes over composition.).
Suppose `f : 'b -> 'c`, `g : 'a -> 'b`, *and* `l : 'a list`.
Then `map (fun x -> f (g x)) l = map f (map g l)`.

Proof. By structural induction on `l`.

Case `l = []`.

To show: `map (fun x -> f (g x)) [] = map f (map g [])`.

`LHS = map (fun x -> f (g x)) [] = []`	by definiton of `map`.
`RHS = map f (map g []) = map f [] = []`	by definition of `map` twice.

The LHS and RHS agree, so this case is proven.

Case `l = x :: xs`.

To show: `map (fun x -> f (g x)) (x :: xs) = map f (map g (x :: xs))`.

IH: `map (fun x -> f (g x)) xs = map f (map g xs)`.

`LHS = map (fun x -> f (g x)) (x :: xs)`	
`= (fun x -> f (g x)) x :: map (f . g) xs`	by definition of `map`.
`= f (g x) :: map (fun x -> f (g x)) xs`	by function application.
`RHS = map f (map g (x :: xs))`	
`= map f (g x :: map g xs)`	by definition of `map` (inside).
`= f (g x) :: map f (map g xs)`	by definition of `map` (outside).
`= f (g x) :: map (fun x -> f (g x)) xs`	by induction hypothesis.

The RHS and LHS agree, so this case is proven.

Therefore by induction for any `l`, `f`, and `g`, `map (fun x -> f (g x)) l = map f (map g l)`. □

Theorem. *For any* `l : int list`*, we have that* `sum_tr l 0 = fold_right (+) l 0`*.*

We can't prove this theorem directly, we need to generalize it.

Generalized theorem (Generalization of theorem 4)**.**
For any `l : int list` *and any* `a : int`*, we have* `sum_tr l a = a + fold_right (+) l 0`*.*

Proof. By structural induction on `l`.

Case `l = []`.

To show: for any `a`, we have `sum_tr [] a = a + fold_right (+) [] 0`.

`LHS = sum_tr [] a = a`	by definition of `sum_tr`.
`RHS = a + fold_right (+) [] 0`	
`= a + 0`	by definition of `fold_right`.
`= a`	by property of addition.

The LHS and RHS agree, so this case is proven.

Case `l = x :: xs`.

To show: for any `a`, we have `sum_tr (x :: xs) a = a + fold_right (+) (x :: xs) 0`.

IH: for any `a`, `sum_tr xs a = a + fold_right (+) xs 0`.

`LHS = sum_tr (x :: xs) a`	
`= sum_tr xs (a + x)`	by definition of `sum_tr`.
`RHS = a + fold_right (+) (x :: xs) 0`	
`= a + (x + fold_right (+) xs 0)`	by definition of `fold_right`.
`= (a + x) + fold_right (+) xs 0`	by associativity of `+`.
`= sum_tr xs (a + x)`	by induction hypothesis.

The RHS and LHS agree, so this case is proven.

Therefore by induction, for any `l` and `a`, we have `sum_tr l a = a + fold_right (+) l 0`. □

Theorem.
Suppose that `f : 'a -> 'a -> 'a` *is associative, i.e. for any* `x, y, z : 'a` *we have*
`f x (f y z) = f (f x y) z`*.*
(For example, addition is such an operator.)

Suppose further that there is `e : 'a` *such that for any* `x : 'a`*, we have*
`f e x = f x e = x`
(In the case of addition, `e` *is* `0`*.)*

Then, for any list `l : 'a list`*, we have that*

```
fold_right f l e = fold_left f l e
```

We can't prove this theorem directly; we need to generalize it over all accumulators.

Generalized theorem (Generalization of theorem 5)**.**
For any accumulator `a : 'a` *and any list* `l : 'a list`*, we have that*

```
f a (fold_right f l e) = fold_left f l a
```

Proof. By structural induction on `l`.

Case `l = []`.

To show: for any `a`, `f a (fold_right f [] e) = fold_left f [] a`

Take arbitrary `a`.

`LHS = f a (fold_right f [] e) = f a e`	by definition of `fold_right`.
`= a`	by property of `f`.
`RHS = fold_left f [] a = a`	by definition of `fold_left`.

The LHS and RHS agree, so this case is proven.

Case `l = x :: xs`.

To show: for any `a`, `f a (fold_right f (x :: xs) e) = fold_left f (x :: xs) a`.

IH: for any `a`, `f a (fold_right f xs e) = fold_left f xs a`

`LHS = f a (fold_right f (x :: xs) e)`	
`= f a (f x (fold_right f xs e))`	by definition of `fold_right`.
`= f (f a x) (fold_right f xs e)`	by associativity of `f`.
`= fold_left f xs (f a x)`	by induction hypothesis.
`RHS = fold_left f (x :: xs) a`	
`= fold_left f xs (f a x)`	by definition of `fold_left`.

The LHS and RHS agree, so this case is proven.

Therefore by induction, for any `a` and `l`, we have `f a (fold_right f l e) = fold_left f l a` provided that `f` is associative and `e` is a neutral element for `f`. □

Proof of theorem 5. To show: for any `l`, `fold_right f l e = fold_left f l e`.

for any `l` and `a`,

`f a (fold_right f l e) = fold_left f l a`	by generalized theorem 2
`f e (fold_right f l e) = fold_left f l e`	by choosing `e` for `a`.
`fold_right f l e = fold_left f l e`	by property of `e` and `f`.

This is precisely what we wanted to show. □