YSC3236: Functional Programming and Proving

Midterm: A Study of Polymorphic Lists and Generic Programming

Zhang Liu, A0190879J

[Created 3 Oct 2020]

Harald (o_o): Fold? What? How?
Mimer: Use what you know.

characters from a website
words from an email thread

Contents

1 Introduction

1.1 The big picture

In this midterm project, we studied the polymorphic lists and some functions of interest that involve lists, namely,

- function computing the length of a list,
- function indexing the list,
- function copying the list,
- function appending a list to another, and
- function reversing the list.

We also studied generic programming over polymorphic lists using `fold_left` and `fold_right`. In this process we discovered many connections (both within the project and to the previous Intro to CS project). We realized the importance of accumulators in relating the structural recursion and tail recursion. In this report I will document the key insights in the program solving process and the bag of tricks I've learned along the way.

1.2 Structure of the midterm project

In each of the exercises 1 to 5, we follow these steps:

- Implement the unit-test function
- Implement the respective function
- State the fold-unfold lemmas
- Prove specification is satisfied
- Prove several algebraic properties associated with the function

In exercise 7, we focus our attention on implementing these functions of interest with generic programming, in particular using the `list_fold_right` and `list_fold_left`.

Lastly, we look back on exercise 2 with the perspectives gained while attempting exercise 7.

2 Exercise 0: Soundness and Completeness of the equality predicate for lists.

2.1 Prove soundness

In week 6 we learned that for an equality predicate to be sound, it means that if applying the predicate to two values yields true, then these two values are Leibniz-equal.

As such, the soundness of equality over lists is stated as follows:

```
Theorem soundness_of_equality_over_lists :
  forall (V : Type)
         (eqb_V : V -> V -> bool),
    (forall v1 v2 : V,
        eqb_V v1 v2 = true -> v1 = v2) ->
    forall v1s v2s : list V,
      eqb_list V eqb_V v1s v2s = true ->
      v1s = v2s.
```

The theorem is proved by induction on v1s:

```
Proof.
  intros V eqb_V S_eqb_V v1s.
  induction v1s as [ | v1 v1s' IHv1s'].
```

We first focus on the base case and use the `intros` tactic to name the assumptions about `v2` and the equality of the two lists.

```
 - intros [ | v2 v2s'] H_eqb.
```

We then use the fold-unfold lemma and apply `H_eqb` to v1,v2, and `S_eqb` yields a Leibniz equality, `v1 = v2`. Using this to rewrite the goal and both sides of the equal sign become the same. And so we can use the `reflexivity` tactic to complete this subgoal.

Next, we use the fold-unfold lemma again and the goal now reads `nil = v2 :: v2s'`. This calls for the use of the `discriminate` tactic.

After this, we start proving the induction step. In the first subgoal, we apply the fold_unfold lemma and then the `discriminate` tactic. In the second subgoal, after `rewrite` using the fold_unfold lemma, the goal reads `v1 :: v1s' = v2 :: v2s'`. After checking the Coq libraies, we apply `H_eqb` as the premise of `andb_prop` and name the components with `destruct`. In the final step, we apply the induction hypothesis and use `reflexivity` to complete the proof.

2.2 Prove completeness

For an equality predicate to be sound, it means that if two values are Leibniz-equal, then applying the predicate to these two values yields true. The theorem is thus stated as follows:

Similar to the proof for soundness, the proof for completeness is also by induction on `v1s`. The only extra step is to use the `injection` tactic to distill the Leibniz equality of two lists into the Leibniz equality of their respective terms. With that, we complete this exercise.

3 Exercise 1: Length Function

3.1 Implement: length with accumulator

The first exercise is to implement the length function using an accumulator. Given the length function in direct style, we simply introduce the accumulator into the base case:

```
| nil =>
  a
```

and the induction step:

```
| v :: vs' =>
  length_v1_aux V vs' (S a)
```

3.2 State the fold-unfold lemmas

Mirroring what has been done in the fold-unfold lemmas for length function in direct style, we state the fold-unfold lemmas for length function using an accumulator using `fold_unfold_tactic`.

3.3 Prove: specification

Now, we are in a position to prove that the implementation of length function with an accumulator satisfies the specification of length function. Since this requires an inductive proof, we define an auxiliary lemma:

```
Lemma length_v1_satisfies_the_specification_of_length_aux :
  forall (V : Type)
         (vs : list V)
         (a : nat),
    a + length_v1_aux V vs 0 = length_v1_aux V vs a.
```

We introduce `a` the accumulator in the base case and induction step and prove the two separately. The rest proof for this lemma is routine induction.

With the help of this auxiliary lemma and unfolding the specification, we can prove the theorem that the accumulator version of length function satisfies the specification of length. The same explanations apply to proofs for specification of other functions in this project.

3.4 Conclusion

We implemented the accumulator version of length function and proved that it satisfies the specification just as well as the direct version. This is a warm-up and in fact foreshadows the importance of accumulators in this project.

4 Exercise 2: Indexing Function

4.1 Unit-test function

We added a few more unit tests. To make them cover more general cases, we use tests that involve `bool` variables. We tested and veried that the indexing functions passed the unit-tests:

```
Compute (test_list_nth (fun V vs n => nat_nth V n vs)).
Compute (test_nat_nth (fun V n vs => list_nth V vs n)).
```

4.2 Prove:list-nth implies nat-nth

The key point in solving this problem is the light of inductil (credit to Halcyon's dream). At first, we intros all the variables, but realized that reverting `vs` and `H_ov` and only introducing them within the base case and induction step gives us a stronger induction hypothesis. With that, we prove by induction on n using the rewrite tactics and respective fold-unfold lemmas.

4.3 Prove: nat-nth implies list-nth

The proof for this proposition is similar to that for the previous proposition, except that the order of the rewrite tactics is exchanged between nat-nth and list-nth.

4.4 Conclusion

An immediate consequence of the previous two propositions is that nat-nth and list-nth are equivalent. The proof is simply to invoke the previous two results using the exact tactic.

5 Exercise 3: Copy Function

5.1 Unit-test function

(similar to what is done in the exercise for length function)

5.2 Implement: copy function

In this part, we implement the copy function with accumulator, which follows the same structure as the length function with accumulator.

5.3 Fold-unfold lemmas

In this part, we state the fold-unfold lemmas for copy function, which follows the same structure as those for the length function.

5.4 Prove: specifications

Like what is done in the exercise for length function, we use the fold-unfold lemmas just stated to prove that specification of copy is satisfied.

5.5 Prove: copy is idempotent

This is a routine induction proof on vs.

5.6 Prove: copy preserves length

This is a routine induction proof on vs.

5.7 Conclusion

In this exercise, we implemented the accumulator version of copy function and proved two properties of copy: (1) copy is idempotent; (2) copy preserves length.

6 Exercise 4: Append Function

6.1 Unit-test function

(similar to what is done in the exercise for length/copy function)

6.2 Implement: append function

In this part, we implement the append function in a direct style. This is defined with reference to the specification of append. In the base case:

```
| nil =>
    v2s
```

and in the induction step:

```
| v1 :: v1s' =>
    v1 :: append_v0_aux V v1s' v2s
```

6.3 Fold-unfold lemmas

In this part, we state the fold-unfold lemmas for append function, which follows the same structure as those for the length/copy function.

6.4 Prove: specifications

Like what is done in the exercise for length function, we use the fold-unfold lemmas just stated to prove that specification of append is satisfied. The only tricky part for append function is: now that we have two lists to work with, we need to write with extra care the arguments for `fold_unfold_append_v0_aux`. Concretely:

```
  - intros V v2s.
    exact (fold_unfold_append_v0_aux_nil V v2s).
  - intros V v1 v1s' v2s.
    exact (fold_unfold_append_v0_aux_cons V v1 v1s' v2s).
```

6.5 Prove: nil is left neutral

This is a direct proof using `exact (fold_unfold_append_v0_aux_nil V v2s).`

6.6 Prove: nil is right neutral

This proof is harder than the previous one given how the `fold_unfold_append_v0_aux` is structured. We cannot use direct proof any more, and so we turn to proof by induction for help. In this case, we can use induction on `v1s` using `induction v1s as [ ` v1 v1s' IHv1s'].— Now we have two subgoals to prove, the base case and the induction step. And this is a routine induction proof.

6.7 Prove: append is not commutative

We first show this for the specific case of lists of natural numbers. To prove that a property does not hold, we want to show that there exists a counter example. This motivates the use of exists tactic:

```
exists (1 :: nil).
exists (2 :: nil).
```

Also worthy of note is that in this proof we need to unfold `not` at the beginning. Additionally, we use `injection` tactic to name the equalities of the components of pairs (e.g. H12 now represents ”1::2::nil” and H21 represents ”2::1::nil”). At the last step, the goal read “false,” which then calls for the `discriminate` tactic to complete the proof.

We then generalize this proposition to polymorphic lists. The structure of the proof is exactly the same except that we now replace the numbers 1, 2 with v1, v2 of type V.

6.8 Prove: append is associative

This is a routine induction proof on v1s.

6.9 Prove: append preserves length

This is a routine induction proof on v1s.

6.10 Prove: append and copy commute

In this part, we show that both implementations of copy function commute with append. The proof for append commutes with `copy_v0` is a routine induction proof on v1s. The proof for append commutes with `copy_v1` is a one-step proof after `unfold`.

6.11 Conclusion

This exercise is a study of append function for polymorphic lists. Some of the algebraic properties associated with append turn out to be particularly important for later parts of the project.

7 Exercise 5: Reverse Function

7.1 Unit-test function

(similar to what is done in the exercise for length/copy/append function)

7.2 Implement: reverse function

In this part, we implement the append function in a direct style. This is defined with reference to the specification of append. In the base case:

```
| nil =>
    nil
```

and in the induction step:

```
| v :: vs' =>
    append_v0 V (reverse_v0_aux V vs') (v :: nil)
```

7.3 Fold-unfold lemmas

In this part, we state the fold-unfold lemmas for reverse function, which follows the same structure as those for the length/copy/append function.

7.4 Prove: specifications

One thing worth noting during the problem solving process: since the specification of reverse contains "append" but not "append_v0," we need the theorem that there is at most one function satisfying the specification of append.

7.5 Prove: reverse is involutory

This is a routine induction proof on v1s.

7.6 Prove: reverse preserves length

This is a routine induction proof on v1s.

7.7 Prove: append and reverse commute

In this part, we prove that append and reverse commute with each other. This is done by induction on v1s and using the `fold_unfold` to rewrite repeatedly. There were a lot of trials and errors in this process, but what helped make it more systematic include:

- Use `unfold` for non-recursive functions.
- Use `fold_unfold` for recursive functions. Additionally, to make this process more structured, we write the induction part of the proof as a separate auxiliary lemma, e.g., `reverse_v0_aux` and `append_v0_aux`.

- Look through the previously proved lemmas about associated algebraic properties, e.g., in the proof for this part of the exercise, we realized that `nil_is_right_neutral_wrt_append_aux` and `append_aux_is_associative` can help us get closer to the goal.
- If all else fails, think of a Eureka lemma.

7.8 Implement: reverse with an accumulator

We implemented the reverse function with an accumulator. In this part, the key steps are outlined:

(i) Implement the accumulator version and state the associated fold-unfold lemmas, the same way we have done before.

(ii) Prove that reverse_v2 satisfies the specification of reverse. However, we realized the need for a Eureka lemma when we reach this goal window:

```
============================
  reverse_v1_aux V vs'
    (v :: nil) =
  append V
    (reverse_v1_aux V vs'
       nil) (v :: nil)
```

(iii) Thus, we state and prove the Eureka lemma using induction on vs and introducing the accumulator a in both the base case and induction step.

```
Lemma about_reverse_v1_aux :
forall (V : Type)
       (vs a : list V),
  reverse_v1_aux V vs a = append_v0_aux V (reverse_v1_aux V vs nil) a.
```

(iv) Prove that the accumulator version is equivalent to reverse in direct style.

(v) Use the theorem about equivalence to prove all the associated propositions that we have done for reverse in direct style.

7.9 Conclusion

In this exercise, we study the reverse function and its accumulator version in details. This will prove to provide important clues for many parts in exercise 7, in particular the structure of the Eureka lemma used in proving specification.

8 Exercise 7: Fold-right and Fold-left Function

8.1 7a and 7b fold right and left in direct style

(already given)

8.2 7c fold-unfold lemmas

(done in the same way as previous exercises)

8.3 7d prove specification for each

(proved respectively in each sub-exercise)

8.4 7e foo and bar

- `foo` is essentially the copy function
- `bar` is essentially the reverse function

The outline of the proof is as follows:

(i) We proved that the foo function is equivalent to copy. This is done by using routine induction on vs.

(ii) In proving that bar function is equivalent to reverse, we realize we need another Eureka lemma when we are stuck at this goal:

```
     list_fold_left V
    (list V) (v' :: nil)
    (fun (v : V)
       (vs : list V) =>
     v :: vs) vs' =
  append_v0_aux V
    (list_fold_left V
       (list V) nil
       (fun (v : V)
          (vs : list V) =>
        v :: vs) vs')
    (v' :: nil)
```

(iii) Thus, we state and prove a lemma which generalizes the accumulator from `(v :: nil)` to `a`. The Eureka lemma is thus as follows:

```
Lemma about_list_fold_left_for_lists :
    forall (V : Type)
           (vs a : list V),
      list_fold_left V (list V) a (fun (v : V) (vs' : list V) => v :: vs') vs =
      append_v0_aux V (list_fold_left V (list V) nil (fun (v : V) (vs' : list V) => v :: vs')
```

This is proved using induction on vs and introducing a in the base case and induction step. We also use `append_aux_is_associative` to help us in proving the induction step.

(iv) With the help of this Eureka lemma, we can now easily prove that bar is equivalent to reverse.

8.5 7f length list fold left

This exercise is to express the **length** function generically with list_fold_left and prove that it satisfies the specification.

The outline of the solution is as follows:

(i) implement the length function generically, using the predefined `list_fold_left` function.

(ii) In trying to prove that this implementation satisfies the specification, the key insight (on hindsight) is to use the final eureka lemma, i.e., if a function is left permutative then `list_fold_left` is equivalent to `list_fold_right`. The inspiration comes from observing the pattern of the expression in the goal window where we got stuck:

```
list_fold_left V nat 1
  (fun (_ : V) (l : nat)
   => S l) vs' =
S
  (list_fold_left V nat
     0
     (fun (_ : V)
        (l : nat) =>
      S l) vs')
```

and compare it with the structure of the finale eureka lemma:

```
 finale_eureka V nat
(fun (_ : V) (l : nat)
 => S l)
   : is_left_permutative
       V nat
       (fun (_ : V)
          (l : nat) =>
        S l) ->
     forall
       nil_case : nat,
     V ->
     forall vs : list V,
     list_fold_left V
       nat (S nil_case)
       (fun (_ : V)
          (l : nat) =>
        S l) vs =
     S
```

```
              (list_fold_left V
                 nat nil_case
                 (fun (_ : V)
                    (l : nat)
                  => S l) vs)
```

(iii) To use the finale eureka, we first need to prove that the successor function is left permutative, which is a direct proof:

```
    Lemma S_is_left_permutative :
  forall (V: Type),
    forall (n : nat),
    is_left_permutative V nat (fun _ n => S n).
Proof.
  unfold is_left_permutative.
  intro n.
  reflexivity.
Qed.
```

(iv) Finally, with the lemma that successor function is left permutative and the finale eureka lemma, we can prove that our implementation of length function using fold left indeed satisfies the specification.

8.6 7g copy list fold right

This exercise is to express the **copy** function generically with list_fold_right and prove that it satisfies the specification. The proof is a direct proof and mirrors that of exercise 3:

```
Theorem copy_fold_right_satisfies_its_specification :
  specification_of_copy copy_fold_right.
Proof.
  unfold specification_of_copy, copy_fold_right.
  split.
  - intro V.
    reflexivity.
  - intros V v vs'.
    reflexivity.
Qed.
```

8.7 7i append list fold right

This exercise is to express the **append** function generically with list_fold_right and prove that it satisfies the specification. The proof is a direct proof and mirrors that of exercise 4.

8.8 7h map list fold right

(map function is optional)

8.9 7i reverse list fold left

This exercise is to express the **reverse** function generically with list_fold_left and prove that it satisfies the specification.

The outline of solution is as follows:

(i) Implement reverse with `list_fold_left`.

(ii) In the specification of reverse, `append` is used instead of as `append_v0`. Thus, we first need to rewrite using the lemma `there_is_at_most_one_function_satisfying_the_specification_of_append`. This allows us to use `fold_unfold_append_v0` in our proof.

(iii) The second key insight is to use the eureka lemma that we proved in trying to show that bar is equivalent to reverse. We arrived at this by observing the goal window below and realizing its similarities to the statement of eureka lemma `about_list_fold_left_for_lists`:

```
============================
  list_fold_left V
    (list V) (v :: nil)
    (fun (v0 : V)
       (vs : list V) =>
     v0 :: vs) vs' =
  append_v0 V
    (list_fold_left V
       (list V) nil
       (fun (v0 : V)
          (vs : list V) =>
        v0 :: vs) vs')
    (v :: nil)
```

8.10 7k relate list fold left and right with reverse

In this exercise, we show that list fold left can be related to list fold right using reverse, vice versa. The problem solving process is outlined below:

(i) First, we formalize what we want to show:

```
For all (V : Type) (vs : list V),
LFL (reverse (vs)) = LFR(vs)
LFR (reverse (vs)) = LFL(vs).
```

(ii) Given the above equations, we can state the two theorems that we would like to prove, one of which is a corollary of the other:

```
Theorem list_fold_right_in_terms_of_list_fold_left_and_reverse_v0 :
  forall (V W : Type)
         (nil_case : W)
         (cons_case : V -> W -> W)
         (vs : list V),
    list_fold_left V W nil_case cons_case (reverse_v0 V vs) =
    list_fold_right V W nil_case cons_case vs.

Theorem list_fold_left_in_terms_of_list_fold_right_and_reverse_v0 :
  forall (V W : Type)
         (v : V)
         (vs : list V),
  forall  (nil_case : W)
          (cons_case : V -> W -> W),
    list_fold_right V W nil_case cons_case (reverse_v0 V (vs)) =
    list_fold_left V W nil_case cons_case (vs).
```

(iii) We start by trying to prove the first theorem by induction on vs. The base case is easier to prove as it involves just rewriting with fold=unfold. We get stuck at the induction step with the following goal:

```
  list_fold_left V W
    nil_case cons_case
    (reverse_v0_aux V
       (v :: vs')) =
  list_fold_right V W
    nil_case cons_case
    (v :: vs')
```

Getting stuck is a big hint to state a Eureka lemma. With the generous help from Prof Danvy, the Eureka lemma we need is:

```
Lemma about_list_fold_left_and_reverse_v0_aux :
  forall (V W : Type)
         (nil_case : W)
         (cons_case : V -> W -> W)
         (v : V)
         (vs : list V),
    list_fold_left V W nil_case cons_case (reverse_v0_aux V (v :: vs)) =
    cons_case v (list_fold_left V W nil_case cons_case (reverse_v0_aux V vs)).
```

(iv) Now, we proceed to prove the first Eureka lemma for this exercise. This is done by using induction on vs. An **enormously crucial** point in this induction proof is the omnipotent Light of Inductil. Here, we make our life much easier with just doing `revert v nil_case.` before the proving by induction. The rationale behind this move is to make our induction hypothesis stronger. This allows us to get over the mental block and proceed with the proof smoothly thereafter.

(v) We are, however, stuck again at this goal window:

```
  ============================
  list_fold_left V W
    nil_case cons_case
```

```
         (append_v0_aux V
            (reverse_v0_aux V
               (v' :: vs'))
            (v :: nil)) =
       cons_case v
         (list_fold_left V W
            nil_case cons_case
            (reverse_v0_aux V
               (v' :: vs')))
```

We again are at lost about what to do next, which is again a big hint that it is time for another Eureka lemma! And indeed, what we need is this:

```
Lemma about_list_fold_left_and_append_v0_aux :
  forall (V W : Type)
         (nil_case : W)
         (cons_case : V -> W -> W)
         (v1s v2s : list V),
    list_fold_left V W
                   nil_case
                   cons_case
                   (append_v0_aux V v1s v2s) =
    list_fold_left V W
                   (list_fold_left V W nil_case cons_case v1s)
                   cons_case
                   v2s.
```

(vi) This second Eureka lemma is in turn proved with induction on v1s, and similarly guided by Light of Inductil. Note that we only need to revert nil_case before the induction (and re-introducing within the base case and induction step) because cons_case remains fixed and thus does not need to be made more general.

(vii) The rest of the proof is routine. And we only need to be careful with choosing what arguments to take in fold-unfold when rewriting to match the patterns.

(viii) After proving the first theorem, we proceed with proving the corollary. This is a direct proof and is done with the help of the previously proved theorem that reverse is involutory. By showing both directions (left in terms of right with reverse and right in terms of left with reverse), we complete this exercise.

8.11 7l list fold right in terms of list fold left

This exercise requires us to implement list fold right in terms of list fold left without using reverse.

8.12 7m list fold left with list fold right

An important realization here is that list fold left is essentially the accumulator version of list fold right. This is exactly analogous to what we have studied in intro to CS. Thus we take inspiration from the process of rewriting power function from the accumulator version to the `fold_right_nat` version in Intro to CS:

```
let power_v0 x n =
  let rec visit n a =
    match n with
    | 0 ->
       a
    | _ ->
       let n' = pred n
       in visit n' (x * a)
  in visit n 1;;
----------------------------
let power_v0 x n =
  let rec visit n a =
    match n with
    | 0 ->
       a
    | _ ->
       let n' = pred n
       in visit n' (x * a)
  in visit n 1;;
---------------------------
let power_v2 x n =
  let rec visit n =
    match n with
    | 0 ->
       (fun a -> a)
    | _ ->
       (fun a -> let n' = pred n
                 in visit n' (x * a))
  in visit n 1;;
---------------------------
let fold_right_nat z s n_given =
  let rec visit n =
    match n with
    | 0 ->
       z
    | _ ->
       let n' = pred n
       in s (visit n')
  in visit n_given;;
------------------------------
let power_v3 x n =
  fold_right_nat (fun a -> a) (fun ih a -> ih (x * a)) n 1;;
```

In parallel with the above process, we have:

```
Fixpoint list_fold_left_aux (V W : Type) (nil_case : W) (cons_case : V -> W -> W) (vs : list V) : W
  match vs with
  | nil =>
    nil_case
  | v :: vs' =>
    list_fold_left_aux V W (cons_case v nil_case) cons_case vs'
  end.
--------------------------
  match vs with
  | nil =>
    (fun nil_case => nil_case)
  | v :: vs' =>
    (fun nil_case => (cons_case v nil_case))
--------------------------
 list_fold_right
    (fun nil_case => nil_case)
    (fun ih nil_case => ih (cons_case v nil_case))
    vs nil_case.
```

The implementation is thus:

```
Definition  list_fold_left_in_terms_of_list_fold_right (V W : Type) (nil_case : W) (cons_case : V -

 list_fold_right V (W -> W)
                 (fun nil_case => nil_case)
                 (fun v ih => (fun nil_case => ih (cons_case v nil_case)))
                 vs nil_case.
```

After the implementation, we further state the fold-unfold lemmas and proved that this implementation satisfies the specification of list fold left.

8.13 7n The Finale Eureka

This eureka lemma is proved with induction on vs. A step worthy of note is:

`unfold is_left_permutative in H_cons_leftp.` where we unfold the definition of left permutative in the hypothesis `H_cons_leftp` so that we can apply this later in our proof as an argument for the finale eureka. (note that being left permutative is the predicate in the finale eureka lemma).

8.14 7o some corollaries

In this exercise, we have found some more examples that have the algebraic property of being left permutative, including addition and multiplication. These are proved respectively by making use of the Coq libraries for natural numbers.

9 Exercise 8: Indexing Function using fold

This exercise is a revisit to of exercise 2 the indexing function. After implementing this it is hard to miss the striking similarity to exercise 7l and 7m. And thus, I attempted to match the patterns in the following function to implement list fold left in terms of list fold right and vice versa. (only manage to pass the specification for list fold left in terms of list fold right).

```
Definition list_nth_rightfold (V : Type) (vs : list V) (n : nat) : option V :=
  list_fold_right (V)
                (nat -> option V)
                (fun n => None)
                (fun v c n =>
                   match n with
                   | O =>
                     Some v
                   | S n' =>
                     c n'
                   end)
                vs
                n.

Definition list_nth_leftfold (V : Type) (vs : list V) (n : nat) : option V :=
  list_fold_left (V)
                (nat -> option V)
                (fun n => None)
                (fun v c n =>
                   match n with
                   | O =>
                     Some v
                   | S n' =>
                     c n'
                   end)
                vs
                n.
```

10 Conclusion

Through the exercises in this midterm project, we have become more confident in working with polymorphic lists. Also of particular importance is that we have now gained more understanding that the key difference between folding right and folding left (and in turn between structural induction and tail recursion) is due to the accumulator. Through the iterative process of being stuck and being inspired, I have summarized some problem-solving strategies that I encountered through this journey:

- It is a good habit to be very structured and systematic, especially when there are a lot of proofs. Write induction proof in an associated auxiliary lemma. Use `fold_unfold` for recursive functions and `unfold` for non-recursive functions.
- Insights sometimes come on hindsight. Often we realized in the later part of the project something that could help with a previous part where we got stuck. Thus problem solving is a iterative process.
- When stuck, use the Light of inductil to guide the way. (or attempt to find a Eureka lemma if it is not already attempted).