Week 11 : Programs from Proofs and Bisimilarity

YSC3236: Functional Programming and Proving

November 6, 2023

"You will be surprised to find out how many things in FPP are of the same elephant."

— Alan Matthew

Group Member Details

- Alan Matthew
  Email: alan.matthew@u.yale-nus.edu.sg
  Matriculation ID: A0224197B
- Jingyi Hou
  Email: jingyi.hou@u.yale-nus.edu.sg
  Matriculation ID: A0242429E
- Sean Lim
  Email: sean.lim@u.yale-nus.edu.sg
  Matriculation ID: A0230369E
- Zhu Wentao
  Email: zhu.wentao@u.yale-nus.edu.sg
  Matriculation ID: A0224190N

Contents

1 Introduction

2 Exercise 00

2.1 Introduction

As usual, we begin this week's exercises by checking the index of concepts and the lecture notes' updates.

In terms of the index of concepts, we are first introduced to the notion of `bisimilarity`. More specifically, according to the lecture notes, structural equality is an inductive notion that only applies to data that are constructed inductively. For streams, the corresponding `coinductive` notion is known as `bisimilarity`.

While lists are constructed inductively, `streams` are constructed `co-inductively` from beginning to the end. Moreover, the end is constructed on demand. The corresponding Gallina keyword in tCPA is `CoInductive`. Similar to `streams`, `lazy lists` are also constructed `co-inductively`.

Last but not least, a key concept in this week's lecture is `programs from proofs`. This means that we can extract the implementation of a program from a relevant proof. We will see this in greater detail in Exercise 01 and Exercise 03 of this week.

For the lecture notes' updates, we see that the lecturer has added "a family of programming puzzles about lists". We will treat this section in the optional exercises this week.

3 Exercise 01

3.1 Introduction

In this exercise, we extract from the proof of even_or_odd a program that divides its input by two. We are provided with a skeleton which we flesh out.

Here is the proof:

```
Lemma even_or_odd :
  forall n : nat,
    exists q : nat,
      n = 2 * q \/ n = S (2 * q).
Proof.
  intro n.
  induction n as [ | n' IHn'].
  - exists 0.
    rewrite -> Nat.mul_0_r.
    left.
    reflexivity.
  - destruct IHn' as [q [H_q | H_q]].
    + rewrite -> H_q.
      exists q.
      right.
      reflexivity.
    + rewrite -> H_q.
      rewrite -> twice_S.
      exists (S q).
      left.
      reflexivity.
  Show Proof.
Qed.
```

3.2 Answer

In the base case we wrote exists 0 and left. In the induction step we wrote exists q when n is even and exists S q when n is odd. So the corresponding program reads:

```
Definition half_rect (n : nat) : nat :=
  match nat_rect
          (fun _ : nat => left_or_right)
          (Left 0)
          (fun _ ih =>
             match ih with
             | Left q =>
```

```
                Right q
            | Right q =>
                Left (S q)
            end)
         n
  with
  | Left q =>
      q
  | Right q =>
      q
  end.
```

As in the lecture notes, we can observe that since the first argument of the succ_case parameter in the call to nat_rect is unused, the definition of half_rect is primitive iterative. That means we can rewrite half_rect with nat_fold_right:

```
Definition half_fold_right (n : nat) : nat :=
  match nat_fold_right
         left_or_right
         (Left 0)
         (fun ih =>
            match ih with
            | Left q =>
                Right q
            | Right q =>
                Left (S q)
            end)
         n
  with
  | Left q =>
      q
  | Right q =>
      q
  end.
```

And of course, since fold_left and fold_right are functionally equal, we can also rewrite half_fold_right with nat_fold_left:

```
Definition half_fold_left (n : nat) : nat :=
  match nat_fold_left
         left_or_right
         (Left 0)
         (fun ih =>
            match ih with
            | Left q =>
                Right q
            | Right q =>
                Left (S q)
            end)
         n
  with
  | Left q =>
      q
  | Right q =>
      q
  end.
```

And following the lecture notes, we can also inline our call to nat_fold_left to get a tail-recursive implementation:

```
Definition half_iterative (n : nat) : nat :=
  let fix visit n acc :=
    match n, acc with
    | 0, _ =>
        acc
    | S n', Left q =>
        visit n' (Right q)
    | S n', Right q =>
        visit n' (Left (S q))
    end
  in match visit n (Left 0) with
     | Left q =>
         q
```

```
    | Right q =>
        q
    end.
```

3.3 Conclusion

In this exercise we get our feet wet with extracting programs are proofs, echoing the mantra of FPP that proofs are programs and programs are proofs. We also see that there is a methodical way to do so through studying the structure of our proof and mapping that onto primitive iterative and recursive fold functions. We also saw how to convert between them or in and out of them, i.e., via in-lining the call.

4 Exercise 03

4.1 Introduction

In this exercise, we prove the lemma about the square-root function and extract a program that computes it from the proof. The lemma states that for any natural number `n`, there exists natural numbers `x` and `r` such that the square root of `n` is `x` with the remainder `r`, and `r` is less than `2x + 1`.

4.2 Answer

We first prove the lemma by doing an induction on `n`.

```
Lemma square_root :
  forall n : nat,
    exists x r : nat,
      n = x * x + r /\ r < S (2 * x).
Proof.
  intro n.
  induction n as [ | n' [x [r [H_n H_r]]]].
  - exists 0, 0.
    split.
    + compute; reflexivity.
    + rewrite -> Nat.mul_0_r.
      exact Nat.lt_0_1.
```

For the base case, both `x` and `r` are equal to 0. For the induction step, since we have the assumption that `r < S (2 * x)`, we have 2 cases: either `r = 2 * x` or `r < 2 * x`, and we look at them separately.

```
  - case (le_lt_or_eq r (2 * x) (Arith_prebase.lt_n_Sm_le r (2 * x) H_r)) as [lt_2x_r | eq_2x_r].
    + exists x, (S r).
      split.
      * rewrite -> H_n.
        ring.
      * Search (_ < _ -> S _ < S _).
        exact (Arith_prebase.lt_n_S_stt r (2 * x) lt_2x_r).
    + exists (S x), 0.
      split.
      * rewrite -> H_n.
        rewrite -> eq_2x_r.
        ring.
      * exact (Nat.lt_0_succ (2 * S x)).
Qed.
```

If `r < 2 * x`, `x` remains unchanged and the remainder increases by 1, so we have `exists x, (S r)`. If `r = 2 * x`, `S n'` becomes a square number, so `x` increases by 1 while the remainder becomes 0, hence `exists (S x), 0`.

Therefore, looking at the proof of the square-root function, we have `exists 0, 0` in the base case, and in the induction step, we have `exists x, (S r)` if `r` is less than `2 * x`, and `exists (S x), 0` if `r` is equal to `2 * x`. Hence, we can write the corresponding program as follows:

```
Definition square_root_and_remainder_rect (n : nat) : nat * nat :=
  nat_rect
    (fun _ : nat => (nat * nat)%type)
    (0, 0)
    (fun _ ih =>
       let (x, r) := ih
```

```
      in if r <? (2 * x)
         then (x, S r)
         else (S x, 0))
    n.
```

Since the first argument of the `succ_case` parameter in the call to `nat_rect` is unused, the definition of `square_root_and_remainder` is not primitive recursive, but primitive iterative, so we can express it in terms of `nat_fold_right`:

```
Definition square_root_and_remainder_right' (n : nat) : nat * nat :=
  nat_fold_right
    (nat * nat)
    (0, 0)
    (fun ih =>
       let (x, r) := ih
       in if r <? (2 * x)
          then (x, S r)
          else (S x, 0))
    n.
```

And since `nat_fold_right` and `nat_fold_left` are functionally equal, we can replace one by the other in the definition of `square_root_and_remainder_right`:

```
Definition square_root_and_remainder_left (n : nat) : nat * nat :=
  nat_fold_left
    (nat * nat)
    (0, 0)
    (fun ih =>
       let (x, r) := ih
       in if r <? (2 * x)
          then (x, S r)
          else (S x, 0))
    n.
```

If we inline the call to `nat_fold_left`, we end up with a tail-recursive implementation of the square-root function with accumulator.

```
Definition square_root_and_remainder_iterative (n : nat) : (nat * nat) :=
  let fix visit n x r :=
    match n with
    | 0 =>
        (x, r)
    | S n' =>
        if r <? (2 * x)
        then visit n' x (S r)
        else visit n' (S x) 0
    end
  in visit n 0 0.
```

4.3 Conclusion

Similar to what we did in Exercise 01, we extracted the program of the square-root function from its proof, reflecting the correspondence between induction proofs and recursive programs through the expression of the implementation as an instance of a fold function. If the first argument of the successor case parameter in the call to `nat_rect` is unused, we can then express the implementation using `nat_fold_right` and `nat_fold_left` instead since the definition is primitive iterative, and we can then inline the call to `nat_fold_left` to write the tail-recursive implementation with accumulators.

5 Exercise 04

5.1 Introduction

In this exercise, we are introduced to streams which are constructed "coinductively", i.e. from beginning to end where the end is constructed on demand. When working with a new recursive data structure, we need a way to compare whether two instances of the data structure are structurally the same.

For streams, the corresponding notion for structural equality of two data structures is called `bisimilarity`. We are asked to prove that `bisimilarity` is an equivalence relation.

```
CoInductive bisimilar : forall V : Type, (V -> V -> Prop) -> stream V -> stream V -> Prop :=
| Bisimilar :
    forall (V : Type)
           (eq_V : V -> V -> Prop)
           (v1 v2 : V)
           (v1s v2s : stream V),
    eq_V v1 v2 ->
    bisimilar V v1s v2s ->
    bisimilar V (Cons V v1 v1s) (Cons V v2 v2s).
```

Observing the definition of bisimilarity above, we see that it is defined inductively. Furthermore, it takes a type `V`, a relation `eq_V`, and two streams of type `V` as arguments. Note that we use `eq_V` to compare the heads of the two stream. So, the definition of `bisimilarity` is as follows: two streams `v1s` and `v2s` are `bisimilar` if they have the same head and their tails are `bisimilar`.

Note that to prove that `bisimilarity` is an equivalence relation, we need to prove that it is reflexive, symmetric, and transitive.

Further note that we are provided with the proof for reflexivity and symmetry. We are asked to prove transitivity.

5.2 Answer

5.2.1 Reflexivity

For a relation to be reflexive, we need to prove that for all `v1s`, `bisimilar v1s v1s` holds. We can prove this by induction on `v1s`.

```
Proposition bisimilar_refl :
  forall (V : Type)
         (eq_V : V -> V -> Prop),
    (forall v : V,
        eq_V v v) ->
    forall vs : stream V,
      bisimilar V eq_V vs vs.
```

Observing the proposition above, we see that it takes a type `V`, a relation `eq_V`, and a proof that `eq_V` is reflexive as arguments. It then returns a proof that `bisimilar` is reflexive.

Let us introduce them into the environment.

```
1 goal (ID 16)

V : Type
eq_V : V -> V -> Prop
eq_V_refl : forall v : V, eq_V v v
============================
forall vs : stream V, bisimilar V eq_V vs vs
```

We can use the `cofix` tactic to introduce the goal we are trying to prove as a coinductive hypothesis. We do this because the goal we are trying to prove, which is a statement about the `bisimilar` relation is reflexive on streams, is a coininductive statement.

```
1 goal (ID 17)

V : Type
eq_V : V -> V -> Prop
eq_V_refl : forall v : V, eq_V v v
coIH : forall vs : stream V, bisimilar V eq_V vs vs
============================
forall vs : stream V, bisimilar V eq_V vs vs
```

Observe that the goal takes in `vs` which is a stream. Since a stream is defined by its head and tail, we can introduce `vs` into the environment as follows.

```
1 goal (ID 22)

V : Type
eq_V : V -> V -> Prop
eq_V_refl : forall v : V, eq_V v v
```

```
coIH : forall vs : stream V, bisimilar V eq_V vs vs
v : V
vs' : stream V
============================
bisimilar V eq_V (Cons V v vs') (Cons V v vs')
```

Observing our goal, we can see that it is an application of the coinductive `BiSimilar` definition we saw earlier. We need to supply the type `V`, the relation `eq_V`, the head `v`, and the tail `vs'`. Furthermore, we also need to supply a proof that `eq_V v v` and a proof that `bisimilar V eq_V vs' vs'` holds. For the latter two, we can use `eq_V_refl` and `coIH` from the environment respectively.

Using the `Check` command, we can see that the goal is indeed an application of the `BiSimilar` definition.

```
Check (Bisimilar V eq_V v v vs' vs' (eq_V_refl v) (coIH vs')).
(*
Bisimilar V eq_V v v vs' vs' (eq_V_refl v) (coIH vs')
     : bisimilar V eq_V (Cons V v vs') (Cons V v vs')
*)
```

We can complete the proof by using the `exact` tactic.

Here is the full proof for reference.

```
Proposition bisimilar_refl :
  forall (V : Type)
         (eq_V : V -> V -> Prop),
    (forall v : V,
        eq_V v v) ->
    forall vs : stream V,
      bisimilar V eq_V vs vs.
Proof.
  intros V eq_V eq_V_refl.
  cofix coIH.
  intros [v vs'].
  Check (Bisimilar V eq_V v v vs' vs' (eq_V_refl v) (coIH vs')).
  exact (Bisimilar V eq_V v v vs' vs' (eq_V_refl v) (coIH vs')).
Qed.
```

5.2.2 Symmetry

For a relation to be symmetric, we need to prove that for all `v1s` and `v2s`, if `bisimilar v1s v2s` holds, then `bisimilar v2s v1s` holds.

```
Proposition bisimilar_sym :
  forall (V : Type)
         (eq_V : V -> V -> Prop),
    (forall v1 v2 : V,
        eq_V v1 v2 ->
        eq_V v2 v1) ->
    forall v1s v2s : stream V,
      bisimilar V eq_V v1s v2s ->
      bisimilar V eq_V v2s v1s.
```

Observing the proposition above, we see that it takes a type `V`, and a relation `eq_V`.

Similar as above, let us introduce the arguments into the environment and use the `cofix` tactic to introduce the goal we are trying to prove as a coinductive hypothesis.

```
1 goal (ID 18)

V : Type
eq_V : V -> V -> Prop
eq_V_sym : forall v1 v2 : V, eq_V v1 v2 -> eq_V v2 v1
coIH : forall v1s v2s : stream V, bisimilar V eq_V v1s v2s -> bisimilar V eq_V v2s v1s
============================
forall v1s v2s : stream V, bisimilar V eq_V v1s v2s -> bisimilar V eq_V v2s v1s
```

Observe that the goal takes in `v1s` and `v2s` which are streams. Since a stream is defined by its head and tail, we can introduce `v1s` and `v2s` in the same way as above. Furthermore, we also need to introduce the implication in the goal, i.e. `bisimilar V eq_V v1s v2s`.

```
1 goal (ID 29)

V : Type
eq_V : V -> V -> Prop
eq_V_sym : forall v1 v2 : V, eq_V v1 v2 -> eq_V v2 v1
coIH : forall v1s v2s : stream V, bisimilar V eq_V v1s v2s -> bisimilar V eq_V v2s v1s
v1 : V
v1s' : stream V
v2 : V
v2s' : stream V
bs_v1s_v2s : bisimilar V eq_V (Cons V v1 v1s') (Cons V v2 v2s')
============================
bisimilar V eq_V (Cons V v2 v2s') (Cons V v1 v1s')
```

The goal is now an instance of the `Bisimilar` definition. However, we also need to supply the proofs that `eq_V v2 v1` and `bisimilar V eq_V v2s' v1s'` holds. However, observing the environment, we see that we have `eq_V_sym` and `coIH` which are proofs of the two statements respectively, but require us to supply the proof of `eq_V v1 v2` and `bisimilar V eq_V v1s' v2s'` respectively.

So the next step in our proof is to obtain the proofs for `eq_V v1 v2` and `bisimilar V eq_V v1s' v2s'`. We can use this using the lemma for the converse of bisimilarity which is provided to us as `Bisimilar_3_12` which given the proof for `bisimilar V eq_V (Cons V v1 v1s') (Cons V v2 v2s')`, returns the conjunction of `eq_V v1 v2 / bisimilar V eq_V v1s' v2s'`.

Let us use this to our advantage to prove the goal and use the `destruct` tactic. Since we can already prove the implication of `Bisimilar_3_12` using `b_v1s_v2s` in our environment and obtainL

```
Check (Bisimilar_3_12 V eq_V v1 v2 v1s' v2s' bs_v1s_v2s).
(*
Bisimilar_3_12 V eq_V v1 v2 v1s' v2s' bs_v1s_v2s
    : eq_V v1 v2 /\ bisimilar V eq_V v1s' v2s'
*)

1 goal (ID 35)

V : Type
eq_V : V -> V -> Prop
eq_V_sym : forall v1 v2 : V, eq_V v1 v2 -> eq_V v2 v1
coIH : forall v1s v2s : stream V, bisimilar V eq_V v1s v2s -> bisimilar V eq_V v2s v1s
v1 : V
v1s' : stream V
v2 : V
v2s' : stream V
bs_v1s_v2s : bisimilar V eq_V (Cons V v1 v1s') (Cons V v2 v2s')
eq_v1_v2 : eq_V v1 v2
bs_v1s'_v2s' : bisimilar V eq_V v1s' v2s'
============================
bisimilar V eq_V (Cons V v2 v2s') (Cons V v1 v1s')
```

Now we have everything we need to use the `Bisimilar` definition to prove our goal, just like we did for reflexivity.

```
Check (Bisimilar V eq_V v2 v1 v2s' v1s' (eq_V_sym v1 v2 eq_v1_v2) (coIH v1s' v2s' bs_v1s'_v2s')).
(*
Bisimilar V eq_V v2 v1 v2s' v1s' (eq_V_sym v1 v2 eq_v1_v2) (coIH v1s' v2s' bs_v1s'_v2s')
    : bisimilar V eq_V (Cons V v2 v2s') (Cons V v1 v1s')
*)
```

Here is the full proof for reference.

```
Proposition bisimilar_sym :
  forall (V : Type)
         (eq_V : V -> V -> Prop),
    (forall v1 v2 : V,
        eq_V v1 v2 ->
        eq_V v2 v1) ->
    forall v1s v2s : stream V,
      bisimilar V eq_V v1s v2s ->
      bisimilar V eq_V v2s v1s.
Proof.
  intros V eq_V eq_V_sym.
  cofix coIH.
```

```
  intros [v1 v1s'] [v2 v2s'] bs_v1s_v2s.
  Check (Bisimilar_3_12 V eq_V v1 v2 v1s' v2s' bs_v1s_v2s).
  destruct (Bisimilar_3_12 V eq_V v1 v2 v1s' v2s' bs_v1s_v2s) as [eq_v1_v2 bs_v1s'_v2s'].
  Check (Bisimilar V eq_V v2 v1 v2s' v1s').
  Check (Bisimilar V eq_V v2 v1 v2s' v1s' (eq_V_sym v1 v2 eq_v1_v2)).
  Check (Bisimilar V eq_V v2 v1 v2s' v1s' (eq_V_sym v1 v2 eq_v1_v2) (coIH v1s' v2s' bs_v1s'_v2s')).
  exact (Bisimilar V eq_V v2 v1 v2s' v1s' (eq_V_sym v1 v2 eq_v1_v2) (coIH v1s' v2s' bs_v1s'_v2s')).
Qed.
```

5.2.3 Transitivity

For a relation to be transitive, we need to prove that for all `v1s`, `v2s`, and `v3s`, if `bisimilar v1s v2s` and `bisimilar v2s v3s` hold, then `bisimilar v1s v3s` holds.

```
Proposition bisimilar_trans :
  forall (V : Type)
         (eq_V : V -> V -> Prop),
    (forall v1 v2 v3: V,
        eq_V v1 v2 ->
        eq_V v2 v3 ->
        eq_V v1 v3) ->
    forall v1s v2s v3s: stream V,
      bisimilar V eq_V v1s v2s ->
      bisimilar V eq_V v2s v3s ->
      bisimilar V eq_V v1s v3s.
```

This proof for this is similar to the proofs for reflexivity and symmetry, but with some extra steps since we are proving the relation holds for three streams instead of two.

Let us proceed as routine and introduce the arguments into the environment and use the `cofix` tactic to introduce the goal we are trying to prove as a coinductive hypothesis.

```
1 goal (ID 19)

V : Type
eq_V : V -> V -> Prop
eq_V_trans : forall v1 v2 v3 : V, eq_V v1 v2 -> eq_V v2 v3 -> eq_V v1 v3
coIH : forall v1s v2s v3s : stream V,
         bisimilar V eq_V v1s v2s -> bisimilar V eq_V v2s v3s -> bisimilar V eq_V v1s v3s
============================
forall v1s v2s v3s : stream V,
bisimilar V eq_V v1s v2s -> bisimilar V eq_V v2s v3s -> bisimilar V eq_V v1s v3s
```

Furthermore, we can introduce `v1s`, `v2s`, and `v3s` as well as the implication in the goal, i.e. `bisimilar V eq_V v1s v2s` and `bisimilar V eq_V v2s v3s`.

```
1 goal (ID 36)

V : Type
eq_V : V -> V -> Prop
eq_V_trans : forall v1 v2 v3 : V, eq_V v1 v2 -> eq_V v2 v3 -> eq_V v1 v3
coIH : forall v1s v2s v3s : stream V,
         bisimilar V eq_V v1s v2s -> bisimilar V eq_V v2s v3s -> bisimilar V eq_V v1s v3s
v1 : V
v1s' : stream V
v2 : V
v2s' : stream V
v3 : V
v3s' : stream V
bs_v1s_v2s : bisimilar V eq_V (Cons V v1 v1s') (Cons V v2 v2s')
bs_v2s_v3s : bisimilar V eq_V (Cons V v2 v2s') (Cons V v3 v3s')
============================
bisimilar V eq_V (Cons V v1 v1s') (Cons V v3 v3s')
```

We can see that, similar as before, we need to supply the proof for `eq_V v1 v3` and `bisimilar V eq_V v1s v3s` to the coininductive `BiSimilar` definition.

Observing our environment, we can obtain the necessary proofs from `eq_V_trans` and `coIH`. However, we also need to supply the proofs for their implications. We can notice a pattern here which is that as we increase the number of

streams we are comparing in our relation, the number of implications we need to prove increases by one. This makes sense since the relation is structural equality.

Going back to the main proof, we can use the `Bisimilar_3_12` lemma to obtain the proofs for the implications, similar to what we did for symmetry. However, since we have two implications, we need to use the `destruct` tactic twice, one for each implication.

```
1 goal (ID 48)

V : Type
eq_V : V -> V -> Prop
eq_V_trans : forall v1 v2 v3 : V, eq_V v1 v2 -> eq_V v2 v3 -> eq_V v1 v3
coIH : forall v1s v2s v3s : stream V,
       bisimilar V eq_V v1s v2s -> bisimilar V eq_V v2s v3s -> bisimilar V eq_V v1s v3s
v1 : V
v1s' : stream V
v2 : V
v2s' : stream V
v3 : V
v3s' : stream V
bs_v1s_v2s : bisimilar V eq_V (Cons V v1 v1s') (Cons V v2 v2s')
bs_v2s_v3s : bisimilar V eq_V (Cons V v2 v2s') (Cons V v3 v3s')
eq_v1_v2 : eq_V v1 v2
bs_v1s'_v2s' : bisimilar V eq_V v1s' v2s'
eq_v2_v3 : eq_V v2 v3
bs_v2s'_v3s' : bisimilar V eq_V v2s' v3s'
============================
bisimilar V eq_V (Cons V v1 v1s') (Cons V v3 v3s')
```

Now we have everything we need to use the `Bisimilar` definition and `exact` tactic to prove our goal, just like we did for reflexivity and symmetry.

Here is the full proof for reference.

```
Proposition bisimilar_trans :
  forall (V : Type)
         (eq_V : V -> V -> Prop),
    (forall v1 v2 v3: V,
        eq_V v1 v2 ->
        eq_V v2 v3 ->
        eq_V v1 v3) ->
    forall v1s v2s v3s: stream V,
      bisimilar V eq_V v1s v2s ->
      bisimilar V eq_V v2s v3s ->
      bisimilar V eq_V v1s v3s.
Proof.
  intros V eq_V eq_V_trans.
  cofix coIH.
  intros [v1 v1s'] [v2 v2s'] [v3 v3s'] bs_v1s_v2s bs_v2s_v3s.
  Check (Bisimilar_3_12 V eq_V v2 v3 v2s' v3s' bs_v2s_v3s).
  destruct (Bisimilar_3_12 V eq_V v1 v2 v1s' v2s' bs_v1s_v2s) as [eq_v1_v2 bs_v1s'_v2s'].
  destruct (Bisimilar_3_12 V eq_V v2 v3 v2s' v3s' bs_v2s_v3s) as [eq_v2_v3 bs_v2s'_v3s'].
  Check (Bisimilar V eq_V v2 v3 v2s' v3s').
  Check (Bisimilar V eq_V v1 v3 v1s' v3s' (eq_V_trans v1 v2 v3 eq_v1_v2 eq_v2_v3)).
  Check (Bisimilar V eq_V v1 v3 v1s' v3s' (eq_V_trans v1 v2 v3 eq_v1_v2 eq_v2_v3)
           (coIH v1s' v2s' v3s' bs_v1s'_v2s' bs_v2s'_v3s')).
  exact (Bisimilar V eq_V v1 v3 v1s' v3s' (eq_V_trans v1 v2 v3 eq_v1_v2 eq_v2_v3)
           (coIH v1s' v2s' v3s' bs_v1s'_v2s' bs_v2s'_v3s')).
Qed.
```

5.3 Conclusion

In this exercise, we are introduced to streams which are constructed "coinductively", i.e. from beginning to end where the end is constructed on demand. Through this exercise, we become familiar with the notion of `bisimilarity`, coinduction, and the `cofix` tactic. Furthermore, by proving the transitivity of `bisimilarity`, we became familiar with how we can using the converse of a relation to prove properties about the relation.

6 Optional exercises

6.1 Introduction

In the optional exercises this week, we are invited to consider a family of programming puzzles about lists. Specifically, we are given a series of puzzles which use the same recursion pattern. As such, their solutions can be expressed using a fold function over lists. In particular, we consider the problem of indexing a list from right to left in this section. We are going to implement this using `list_fold_left`.

6.2 Answer

The implementation using `list_fold_left` is as follows:

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

To test our implementation, we can look at the test `test_list_index_rtl_nat` given above in the file.

```
Compute (test_list_index_rtl_nat (fun ns n => list_index_rtl_left nat ns n)).
```

This implementation passes the test.

To prove the correctness of our implementation, we need to look at how indexing a list from left to right is implemented in the same file.

```
Fixpoint list_index_ltr (V : Type) (vs : list V) (n : nat) : option V :=
  match vs with
  | nil =>
    None
  | v :: vs' =>
    match n with
    | O =>
      Some v
    | S n' =>
      list_index_ltr V vs' n'
    end
  end.
```

We then observe another test:

```
Compute (test_list_index_rtl_nat (fun ns n => list_index_ltr nat (list_reverse nat ns) n)).
```

Here we see that indexing a list from left to right and indexing a list from right to left can be related using the `list_reverse` function. As such, we can formalise the correctness of `list_index_rtl_left` in the following manner:

```
Theorem correctness_of_list_index_rtl_left :
  forall (V : Type)
         (vs : list V)
         (n : nat),
    list_index_rtl_left V vs n = list_index_ltr V (list_reverse V vs) n.
```

The theorem is stating that indexing a list from right to left should yield the same result as indexing the reverse of that list from left to right, given the same natural number `n`. This concurs with our intuitive understanding as well.

Before we proceed with the proof, it is critical to set up the relevant infrastructure. In particular, we need to investigate the recursive functions and write the relevant fold-unfold lemmas.

In the theorem, there are three recursive functions: `list_index_rtl_left`, `list_index_ltr` and `list_reverse`. Specifically, `list_index_ltr` and `list_reverse` are recursive functions with the `Fixpoint` notation. `list_index_rtl_left` is recursive as it is implemented with the recursive function `list_fold_left`.

The fold-unfold lemmas for `list_fold_left` and `list_reverse` can be found in the midterm project. We will also make use of the fold-unfold lemmas for `list_fold_right` and `list_append` in this proof. Here we will write the fold-unfold lemmas for `list_index_ltr`, carefully applying the structure we know:

```
Lemma fold_unfold_list_index_ltr_nil :
  forall (V : Type)
         (n : nat),
    list_index_ltr V nil n =
      None.
Proof.
  fold_unfold_tactic list_index_ltr.
Qed.

Lemma fold_unfold_list_index_ltr_cons :
    forall (V : Type)
        (v : V)
        (vs' : list V)
        (n : nat),
    list_index_ltr V (v :: vs') n =
    match n with
    | O =>
    Some v
    | S n' =>
    list_index_ltr V vs' n'
    end.
Proof.
    fold_unfold_tactic list_index_ltr.
Qed.
```

Now we are ready to begin our proof. We first attempted a direct proof of the theorem, like this :

```
Theorem correctness_of_list_index_rtl_left :
  forall (V : Type)
         (vs : list V)
         (n : nat),
    list_index_rtl_left V vs n = list_index_ltr V (list_reverse V vs) n.
Proof.
  Compute (let V := nat in
           let vs := 3 :: 2 :: 1 :: 0 :: nil in
           let n := 1 in
           list_index_rtl_left V vs n = list_index_ltr V (List.rev vs) n).
  intros V vs.
  induction vs as [ | v' vs' IHvs'].
  - intros [ | n'].
    + rewrite -> (fold_unfold_list_reverse_nil V).
      rewrite -> (fold_unfold_list_index_ltr_nil V).
      unfold list_index_rtl_left.
      rewrite -> (fold_unfold_list_fold_left_nil V
                    (nat -> option V)
                    (fun _ : nat => None)
                    (fun (a : V)
                         (ih : nat -> option V)
                         (v : nat) => match v with
                                      | 0 => Some a
                                      | S n' => ih n'
                                      end)).
      reflexivity.
    + rewrite -> (fold_unfold_list_reverse_nil V).
      rewrite -> (fold_unfold_list_index_ltr_nil V).
      unfold list_index_rtl_left.
      rewrite -> (fold_unfold_list_fold_left_nil V
                    (nat -> option V)
                    (fun _ : nat => None)
                    (fun (a : V)
                         (ih : nat -> option V)
                         (v : nat) => match v with
                                      | 0 => Some a
                                      | S n'0 => ih n'0
                                      end)).
      reflexivity.
  - intros n.
```

At this point, the *goals* window reads :

```
1 goal (ID 663)

  V : Type
  v' : V
  vs' : list V
  IHvs' : forall n : nat,
          list_index_rtl_left V vs' n =
          list_index_ltr V (list_reverse V vs') n
  n : nat
  ============================
  list_index_rtl_left V (v' :: vs') n =
  list_index_ltr V (list_reverse V (v' :: vs')) n
```

We then formulated the following Eureka lemma by combining our goal and the induction hypothesis:

```
Theorem about_the_correctness_of_list_index_rtl_left :
  forall (V : Type)
         (v' : V)
         (vs' : list V)
         (n : nat),
    list_index_rtl_left V vs' n =
      list_index_ltr V (list_reverse V vs') n ->
    list_index_rtl_left V (v' :: vs') n =
      list_index_ltr V (list_reverse V (v' :: vs')) n.
```

However, proving this turns out to be a big challenge. Indeed, it is quite difficult to come up with an actionable lemma that can be of help if we go down this route. We are then invited to consider an indirect proof by relating `list_fold_left` and `list_fold_right`, which is also something we investigated in the midterm project.

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

By using this upfront, we can hopefully come up with an Eureka lemma that can be proved routinely and applied. We attempted this by first using the `unfold` and `rewrite` tactics at the beginning of our proof :

```
unfold list_index_rtl_left.
rewrite -> (relating_list_fold_left_and_list_fold_right_using_list_reverse V
            (nat -> option V)
            (fun _ : nat => None)
            (fun (a : V)
                    (ih : nat -> option V)
                    (v : nat) =>
                match v with
                | O => Some a
                | S n' => ih n'
                end) vs).
```

Unfortunately, we ran out of the time allotted for FPP after some more failed attempts. But this is where we ended up with at the time of submission.

6.3 Conclusion

From this exercise, we have learnt to be mindful of what we have been introduced to so far in this course, in particular the midterm project. This is especially true as we need to learn the moves properly to make meaningful progress in our proof. Otherwise, it is easy to get stuck and move in random directions.

7 Conclusion

1. **Exercises 01 & 03** : In these 2 exercises, we learn how to extract programs from proofs through fold functions, which furthers our understanding of the correspondence between induction proofs and recursive programs. We also learn how to write tail-recursive functions with accumulators using the proofs. If the first argument of the

successor case parameter in the call to `nat_rect` is unused, i.e., the definition is primitive iterative, we can express the implementation using `nat_fold_right` and `nat_fold_left`, which is functionally equivalent, and the job is done by inlining the call to `nat_fold_left`.

2. **Exercise 04** : In this exercise, we are introduced to streams which are constructed "coinductively", in Coq. This is our first introduction to inductive data structures in Coq in which the end is constructed on demand. As we prove the property of transitivity of `bisimilarity`, we become familiar with how we can using the converse of a relation to prove properties about the relation. Furthermore, we also noticed that in the proof for transitivity, the number of implications we need to prove increases by one as we increase the number of streams we are comparing in our relation. This makes sense since the relation is structural equality.

3. **Optional exercises** : For the optional exercises this week, we are invited to consider a family of programming puzzles about lists. Here we reaffirm our appreciation of folding left and right, especially their role in expressing the recursion patterns in these puzzles listed. Perhaps more importantly, we are again reminded to keep in mind the previous lemmas proved and make use of them to make further progress. Indeed, we can only do more complex stuff building on top of what we already know.