Week 10 : Induction with Induction

YSC3236: Functional Programming and Proving

"Je ne sais pas le reste."
— Évariste Galois

November 21, 2023

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

The big concepts this week were about a eureka lemma creator for resetting accumulators and the introduction of the concepts of weak and strong induction, the former being first-order structural induction (where you only have one induction hypothesis; also called mathematical induction) and the latter being induction with many induction hypotheses.

To appreciate the beauty of the eureka lemma maker, we work on the familiar factorial function and observe how it does the heavy lifting we have done manually many times previously. After getting sea-legs working with the factorial function, we generalise our understanding of the eureka lemma maker by working on another familiar data structure, the binary tree, and see that it is just as useful there as we would expect.

We then move on to a program of proofs that help us understand the design of higher-order induction functions, ending with a comparison between using higher-order and first-order induction to prove a coin puzzle. For the last bit of this week, we have a returnee from Intro CS: mutual recursion! Here like before with the eureka lemmas, we apply what we have learnt about first and higher order induction to a different scenario, this time mutually defined odd and even predicates.

2 Exercise 00

We begin this week's exercises by checking the index of concepts and updates of the lecture notes. In particular, we are introduced to the various types of induction over Peano numbers.

The `first-order` structural recursion over Peano numbers is also known as mathematical induction. It has only one induction hypothesis and is sometimes referred to as `weak induction`.

The `second-order` structural recursion over Peano numbers has two induction hypotheses, which suits the structure of the Fibonacci function.

Similarly, the `third-order` and `fourth-order` structural recursion over Peano numbers has three and four induction hypotheses respectively. More generally, in the `course-of-value` induction principle, k induction hypotheses are involved. The induction principle involving many induction hypotheses is also known as `strong induction`.

In the exercises this week, we will see how these additional induction principles can help us prove seemingly intractable propositions in a relatively straightforward manner.

3 Exercise 01

3.1 Introduction

In this exercise, we are asked with formulating an eureka lemma for `length_alt_aux` using `make_Eureka_lemma`.

3.2 Answer

Following the lecture notes, we have the following definition for `make_Eureka_lemma`:

```
Definition make_Eureka_lemma (A : Type) (id_A : A) (combine_A : A -> A -> A) (c : A -> A) (a : A) :
    Prop :=
  c a = combine_A (c id_A) a.
```

The first argument is the type for the accumulator, then an identity element for this type, and a function to combine two elements of this type. Then we have the auxiliary function and the accumulator.

Supplying these arguments for the `length` function, we have:

```
Check (forall (V : Type) (vs : list V) (a : nat), make_Eureka_lemma nat 0 Nat.add (length_alt_aux V
    vs) a).
(*
forall (V : Type) (vs : list V) (a : nat), make_Eureka_lemma nat 0 Nat.add (length_alt_aux V vs) a
    : Prop
*)
```

We can them formulate this property as a lemma:

```
Lemma about_length_alt_aux :
  forall (V : Type) (vs : list V) (a : nat),
    make_Eureka_lemma nat 0 Nat.add (length_alt_aux V vs) a.

(*
1 goal (ID 18)

============================
forall (V : Type) (vs : list V) (a : nat),
make_Eureka_lemma nat 0 Nat.add (length_alt_aux V vs) a
*)
```

To prove this, let us first unfold the definition of `make_Eureka_lemma`. Proceeding from this, we can follow the goal and introduce the variables `V` and `vs` into the environment. Note that since the lemma has an accumulator, we should perform induction using Light of Inductil.

Inducting on `vs`, we have the following base case:

```
1 goal (ID 30)

V : Type
a : nat
============================
length_alt_aux V nil a = length_alt_aux V nil 0 + a
```

Which is a double rewrite of the fold-unfold lemma `fold_unfold_length_alt_aux_nil` and a single rewrite of `Nat.add_0_l`.

Proceeding to the inductive case, we have:

```
1 goal (ID 35)

V : Type
v : V
vs' : list V
IHvs' : forall a : nat, length_alt_aux V vs' a = length_alt_aux V vs' 0 + a
a : nat
============================
length_alt_aux V (v :: vs') a = length_alt_aux V (v :: vs') 0 + a
```

Which is a double rewrite of the fold-unfold lemma `fold_unfold_length_alt_aux_cons`. Afterwards, we can apply the induction hypothesis `IHvs'` on `(S a)`.

```
1 goal (ID 38)

V : Type
v : V
vs' : list V
IHvs' : forall a : nat, length_alt_aux V vs' a = length_alt_aux V vs' 0 + a
a : nat
============================
length_alt_aux V vs' 0 + S a = length_alt_aux V vs' 1 + a
```

Observing the goal, we can still apply the induction hypothesis `IHvs'` on `1`. Finally, we only need to rewrite the goal using `Nat.add_succ_l` and `Nat.add_assoc` to complete the proof.

Here is the full proof:

```
Lemma about_length_alt_aux :
  forall (V : Type) (vs : list V) (a : nat),
    make_Eureka_lemma nat 0 Nat.add (length_alt_aux V vs) a.
Proof.
  unfold make_Eureka_lemma.
  intros V vs.
  induction vs as [n | v vs' IHvs'].
  + intro a.
    rewrite -> 2 fold_unfold_length_alt_aux_nil.
    rewrite -> Nat.add_0_l.
    reflexivity.
  + intro a.
    rewrite -> 2 fold_unfold_length_alt_aux_cons.
    rewrite -> (IHvs' (S a)).
    rewrite -> (IHvs' 1).
    rewrite <- Nat.add_1_l.
    rewrite -> Nat.add_assoc.
    reflexivity.
Qed.
```

3.3 Conclusion

In this exercise, we have shown that the Eureka lemma we postulated for the length function is an instance of `make_Eureka_lemma`. The importance of this is to realize that all of the Eureka lemmas we have been writing so far, including this one, is part of the same elephant. In other words, they are an instance of the same idea and possess the same general structure.

4 Exercise 02

4.1 Introduction

In this exercise, we are tasked to formulate an Eureka lemma for list reversal using `make_Eureka_lemma`.

4.2 Answer

In formulating an Eureka lemma for list reversal, we must first understand the structure of the list reversal function. In particular, we must understand that the type of the accumulator, which carries the reversed list, is the same as the type of the list itself (`list V`), and that the identity element is the tail of the list (`vs`), the function to combine two elements is the list append function (`list_append V`), and the auxiliary function is the list reversal function (`reverse_alt_aux V`), and finally the accumulator (`a`).

Note that for this exercise, we are going to import the `list_append` function from the midterm exercise and its respective fold-unfold lemmas.

Using the `Check` tactic on our Eureka lemma, we have:

```
Check (forall (V : Type) (vs a : list V), make_Eureka_lemma (list V) vs (list_append V) (
    reverse_alt_aux V vs) a).
(*
forall (V : Type) (vs a : list V),
make_Eureka_lemma (list V) vs (list_append V) (reverse_alt_aux V vs) a
    : Prop
*)
```

Formulating this as a lemma, we have the following goal:

```
1 goal (ID 157)

============================
forall (V : Type) (vs a : list V),
make_Eureka_lemma (list V) nil (list_append V) (reverse_alt_aux V vs) a
```

We can proceed initially as routine by unfolding the definition of `make_Eureka_lemma` and introducing the variables `V` and `vs` into the environment.

For the base case, we have:

```
1 goal (ID 169)

V : Type
a : list V
============================
reverse_alt_aux V nil a = list_append V (reverse_alt_aux V nil nil) a
```

Which are a double rewrite of the fold-unfold lemma `fold_unfold_reverse_alt_aux_nil` and a single rewrite of the fold-unfold lemma `fold_unfold_list_append_nil`.

Proceeding to the inductive case, we have:

```
1 goal (ID 174)

V : Type
v : V
vs' : list V
IHvs' : forall a : list V, reverse_alt_aux V vs' a = list_append V (reverse_alt_aux V vs' nil) a
a : list V
============================
reverse_alt_aux V (v :: vs') a = list_append V (reverse_alt_aux V (v :: vs') nil) a
```

Which is a double rewrite of the fold-unfold lemma `fold_unfold_reverse_alt_aux_cons` and we can proceed by supplying the induction hypothesis `IHvs'` on `(v :: a)`.

However, the goal then reads

```
1 goal (ID 178)

V : Type
v : V
vs' : list V
IHvs' : forall a : list V, reverse_alt_aux V vs' a = list_append V (reverse_alt_aux V vs' nil) a
a : list V
============================
list_append V (reverse_alt_aux V vs' nil) (v :: a) =
list_append V (reverse_alt_aux V vs' (v :: nil)) a
```

And we require an additional Eureka lemma to prove this. In particular, we require an Eureka lemma that states that the list append of `reverse_alt_aux V vs' nil` and `(v :: a)` is equal to the list append of `reverse_alt_aux V vs' (v :: nil)` and `a`. Let us name this lemma `about_list_append_cons`.

```
Lemma about_list_append_cons :
  forall (V : Type)
         (v2 : V)
         (v1s v2s' : list V),
    list_append V v1s (v2 :: v2s') = list_append V (list_append V v1s (v2 :: nil)) v2s'.
```

Admitting this lemma for now, we can proceed with the proof. We can rewrite the goal using `about_list_append_cons` and `IHvs'` on `(v :: nil)` to complete the proof.

With that, let us return to proving `about_list_append_cons`. We can proceed by induction on `vs'`. The base case is as follows:

```
1 goal (ID 172)

V : Type
v2s' : list V
v2 : V
============================
list_append V nil (v2 :: v2s') = list_append V (list_append V nil (v2 :: nil)) v2s'
```

However, we quickly realize that we need an additional Eureka lemma to prove this. In particular, we require an Eureka lemma that states that in the list append operation nil is left neutral. Let us name this lemma `nil_is_left_neutral_wrt_list_append`.

```
Proposition nil_is_left_neutral_wrt_list_append :
  forall (V : Type)
          (vs : list V),
    list_append V nil vs = vs.
```

Admitting this lemma for now, we can proceed with the proof. We can perform a double rewrite using our newly defined lemma `nil_is_left_neutral_wrt_list_append` and then rewrite using `fold_unfold_list_append_cons`. Finally, we can rewrite again using `nil_is_left_neutral_wrt_list_append` to complete the proof.

Let us proceed to the inductive case. We have:

```
1 goal (ID 178)

V : Type
v1 : V
v1s', v2s' : list V
```

```
IHv1s' : forall v2 : V,
          list_append V v1s' (v2 :: v2s') = list_append V (list_append V v1s' (v2 :: nil)) v2s'
v2 : V
============================
list_append V (v1 :: v1s') (v2 :: v2s') =
list_append V (list_append V (v1 :: v1s') (v2 :: nil)) v2s'
```

Which we can proceed by rewriting using `fold_unfold_list_append_cons` and then applying the induction hypothesis `IHv1s'` on `v2`. Lastly, we can perform two more rewrites using `fold_unfold_list_append_cons` to complete the proof.

With that, we have completed the proof for `about_list_append_cons`.

Let us now return to the proof of `nil_is_left_neutral_wrt_list_append`.

```
1 goal (ID 150)

============================
forall (V : Type) (vs : list V), list_append V nil vs = vs
```

We can introduce the variables `V` and `vs` into the environment. We can then proceed by rewriting using our fold-unfold lemma for list append on the nil case `fold_unfold_list_append_nil` on `V` and `vs` to complete the proof.

Here is the full proof:

```
Proposition nil_is_left_neutral_wrt_list_append :
  forall (V : Type)
         (vs : list V),
    list_append V nil vs = vs.
Proof.
  intros V vs.
  rewrite -> (fold_unfold_list_append_nil V vs).
  reflexivity.
Qed.

Lemma about_list_append_cons :
  forall (V : Type)
         (v2 : V)
         (v1s v2s' : list V),
    list_append V v1s (v2 :: v2s') = list_append V (list_append V v1s (v2 :: nil)) v2s'.
Proof.
  intros V v2 v1s v2s'.
  revert v2.
  induction v1s as [ | v1 v1s' IHv1s'].
  + intro v2.
    rewrite -> 2 nil_is_left_neutral_wrt_list_append.
    rewrite -> fold_unfold_list_append_cons.
    rewrite -> nil_is_left_neutral_wrt_list_append.
    reflexivity.
  + intro v2.
    rewrite -> fold_unfold_list_append_cons.
    rewrite -> (IHv1s' v2).
    rewrite -> 2 fold_unfold_list_append_cons.
    reflexivity.
Qed.

Lemma about_reverse_alt_aux :
  forall (V : Type) (vs a : list V),
    make_Eureka_lemma (list V) nil (list_append V) (reverse_alt_aux V vs) a.
Proof.
  unfold make_Eureka_lemma.
  intros V vs.
  induction vs as [n | v vs' IHvs'].
  + intro a.
    rewrite -> 2 fold_unfold_reverse_alt_aux_nil.
    rewrite -> fold_unfold_list_append_nil.
    reflexivity.
  + intro a.
    rewrite -> 2 fold_unfold_reverse_alt_aux_cons.
    rewrite -> (IHvs' (v :: a)).
    rewrite -> about_list_append_cons.
    rewrite -> (IHvs' (v :: nil)).
    reflexivity.
Qed.
```

4.3 Conclusion

In this exercise, we arrive at the same conclusion as Exercise 01. However, it is just that the proof is more involved. In particular, we had to prove an additional Eureka lemma to prove the original Eureka lemma, and then another additional Eureka lemma to prove our second Eureka lemma.

5 Exercise 04

5.1 Introduction

In this exercise, we revisit the accumulator-passing tail-recursive factorial function and state its Eureka lemma using `make_Eureka_lemma`.

As part of the exercise, we have been provided with a tail-recursive implementation of the factorial function, its respective fold-unfold lemmas, and the factorial function definition which makes use of the accumulator. I will omit this here for brevity.

5.2 Answer

Our first task is to implement an Eureka lemma for the factorial function `about_fac_alt_aux`. Here, if we step back for a moment, we notice that in the tail-recursive auxiliary function `fac_alt_aux`, the accumulator holds the value of the factorial we are computing. As such, I postulated an Eureka lemma that states that the factorial of `fac_alt_aux n a` is equal to the factorial of `fac_alt_aux n 1` multiplied by `a`.

Hence, we can formalize the lemmas as follows:

```
Lemma about_fac_alt_aux :
  forall n a : nat,
    fac_alt_aux n a = fac_alt_aux n 1 * a.
```

Since we have an accumulator, we can proceed our proof using the Light of Inductil. In particular, we can proceed by induction on `n`.

Observing the base case we have:

```
1 goal (ID 25)

a : nat
============================
fac_alt_aux 0 a = fac_alt_aux 0 1 * a
```

Which is a double rewrite of the fold-unfold lemma `fold_unfold_fac_alt_aux_O`.

```
1 goal (ID 27)

a : nat
============================
a = 1 * a
```

Which we can rewrite using the lemma `Nat.mul_1_r`. With that, we can use the `reflexivity` tactic to complete the base case.

For the inductive case, we have:

```
1 goal (ID 30)

n' : nat
IHn' : forall a : nat, fac_alt_aux n' a = fac_alt_aux n' 1 * a
a : nat
============================
fac_alt_aux (S n') a = fac_alt_aux (S n') 1 * a
```

Which is a double rewrite of the fold-unfold lemma `fold_unfold_fac_alt_aux_S`.

```
1 goal (ID 32)

n' : nat
IHn' : forall a : nat, fac_alt_aux n' a = fac_alt_aux n' 1 * a
a : nat
============================
fac_alt_aux n' (S n' * a) = fac_alt_aux n' (S n' * 1) * a
```

Observing the LHS and RHS, we can apply the induction hypothesis on `S n' * a` and `S n' * 1` respectively.

```
1 goal (ID 34)

n' : nat
IHn' : forall a : nat, fac_alt_aux n' a = fac_alt_aux n' 1 * a
a : nat
============================
fac_alt_aux n' 1 * (S n' * a) = fac_alt_aux n' 1 * (S n' * 1) * a
```

We can then rewrite the goal using `Nat.mul_1_r` and `Nat.mul_assoc` to prove the inductive case.

With that our proof is complete.

```
Lemma about_fac_alt_aux :
  forall n a : nat,
    fac_alt_aux n a = fac_alt_aux n 1 * a.
Proof.
  intro n.
  induction n as [ | n' IHn'].
    + intro a.
    rewrite -> 2 fold_unfold_fac_alt_aux_O.
    rewrite -> Nat.mul_1_l.
    reflexivity.
  + intro a.
    rewrite -> 2 fold_unfold_fac_alt_aux_S.
    rewrite -> (IHn' (S n' * a)).
    rewrite -> (IHn' (S n' * 1)).
    rewrite -> Nat.mul_1_r.
    rewrite -> Nat.mul_assoc.
    reflexivity.
Qed.
```

Next, we proceed with formulating an Eureka lemma using `make_Eureka_lemma`.

Firstly, let us understand the what it does:

```
Definition make_Eureka_lemma (A : Type) (id_A : A) (combine_A : A -> A -> A) (c : A -> A) (a : A) :
    Prop :=
  c a = combine_A (c id_A) a.
```

The first argument is the type for the accumulator, then an identity element for this type, and a function to combine two elements of this type. Then we have the auxiliary function and the accumulator.

Supplying these arguments for the tail-recursive factorial function, we have:

```
Check (forall x n a : nat, make_Eureka_lemma nat 1 Nat.mul (power_alt_aux x n) a).

forall n a : nat, make_Eureka_lemma nat 1 Nat.mul (fac_alt_aux n) a
     : Prop
```

Which seems to be what we need. Formalizing this as an Euler lemma, we have:

```
Lemma about_fac_alt_aux' :
  forall n a : nat,
    make_Eureka_lemma nat 1 Nat.mul (fac_alt_aux n) a.
```

To prove this, let us first unfold the definition of `make_Eureka_lemma`.

```
1 goal (ID 18)

============================
forall n a : nat, fac_alt_aux n a = fac_alt_aux n 1 * a
```

Notice that the goal is precisely the same as the Eureka lemma we proved earlier. Hence, the proof is identical. Furthermore, we can use the `exact` tactic on the lemma we proved earlier to complete the proof.

```
Lemma about_fac_alt_aux' :
  forall n a : nat,
    make_Eureka_lemma nat 1 Nat.mul (fac_alt_aux n) a.
Proof.
  unfold make_Eureka_lemma.
  intro n.
  exact (about_fac_alt_aux n).
Qed.
```

Hence, we showed that the first Eureka lemma we postulated is expressible as an instance of `make_Eureka_lemma`.

5.3 Conclusion

In this exercise, we have shown that the Eureka lemma we postulated for the tail-recursive factorial function is an instance of `make_Eureka_lemma`. The importance of this is to realize that all of the Eureka lemmas we have been writing so far, including this one, is an part of the same elephant. In other words, they are an instance of the same idea and possess the same general structure.

All Eureka lemmas have the same general structure.

6 Exercise 05

6.1 Introduction

Binary trees are a fundamental data structure in computer science. Often, we are interested in computing properties of binary trees. In this document, we will examine the problem of computing the sum of the values stored in the leaves of a binary tree of natural numbers. We present two implementations: a direct recursive method and a tail-recursive method. The goal is to prove that these two methods are equivalent in terms of their results.

```
Inductive binary_tree : Type :=
| Leaf : nat -> binary_tree
| Node : binary_tree -> binary_tree -> binary_tree.
```

```
Fixpoint weight (t : binary_tree) : nat :=
  match t with
  | Leaf n => n
  | Node t1 t2 => weight t1 + weight t2
  end.
```

```
Fixpoint weight_alt_aux (t : binary_tree) (a : nat) : nat :=
  match t with
  | Leaf n => n + a
  | Node t1 t2 => weight_alt_aux t1 (weight_alt_aux t2 a)
  end.

Definition weight_alt (t : binary_tree) : nat :=
  weight_alt_aux t 0.
```

6.2 Answer

Firstly, I am tasked to postulate and prove an Eureka lemma for the `weight` function.

Similar to the factorial function, the computation of the weight is stored in the accumulator. As such, I postulate that the weight of `weight_alt_aux t a` is equal to the weight of `weight_alt_aux t 0` added to `a`.

Formalizing the lemma is as follows:

```
Lemma about_weight_alt_aux :
  forall (t : binary_tree)
         (a : nat),
    weight_alt_aux t a = weight_alt_aux t 0 + a.
```

We can proceed with the proof using the Light of Inductil. In particular, we can proceed by induction on `t`.

The base case is as follows:

```
1 goal (ID 40)

n, a : nat
============================
weight_alt_aux (Leaf n) a = weight_alt_aux (Leaf n) 0 + a
```

Which is a double rewrite of the fold-unfold lemma `fold_unfold_weight_alt_aux_Leaf`.

```
1 goal (ID 42)

n, a : nat
============================
n + a = n + 0 + a
```

Next, we must reorder the goal, which is an instance of `Nat.add_shuffle0`.

```
1 goal (ID 56)

n, a : nat
============================
n + a = n + a + 0
```

Finally we can rewrite the goal using `Nat.add_0_r` on `(n + a)` to complete the base case.

For the inductive case, we have:

```
1 goal (ID 59)

t1, t2 : binary_tree
IHt1 : forall a : nat, weight_alt_aux t1 a = weight_alt_aux t1 0 + a
IHt2 : forall a : nat, weight_alt_aux t2 a = weight_alt_aux t2 0 + a
a : nat
============================
weight_alt_aux (Node t1 t2) a = weight_alt_aux (Node t1 t2) 0 + a
```

Which is a double rewrite of the fold-unfold lemma `fold_unfold_weight_alt_aux_Node`.

```
1 goal (ID 61)

t1, t2 : binary_tree
IHt1 : forall a : nat, weight_alt_aux t1 a = weight_alt_aux t1 0 + a
IHt2 : forall a : nat, weight_alt_aux t2 a = weight_alt_aux t2 0 + a
a : nat
============================
weight_alt_aux t1 (weight_alt_aux t2 a) = weight_alt_aux t1 (weight_alt_aux t2 0) + a
```

We can then apply the induction hypothesis `IHt1` twice on `weight_alt_aux t2 a` and `weight_alt_aux t2 0` respectively.

```
1 goal (ID 63)

t1, t2 : binary_tree
IHt1 : forall a : nat, weight_alt_aux t1 a = weight_alt_aux t1 0 + a
IHt2 : forall a : nat, weight_alt_aux t2 a = weight_alt_aux t2 0 + a
a : nat
============================
weight_alt_aux t1 0 + weight_alt_aux t2 a = weight_alt_aux t1 0 + weight_alt_aux t2 0 + a
```

We can then apply the induction hypothesis `IHt2` on `a`.

```
1 goal (ID 64)

t1, t2 : binary_tree
IHt1 : forall a : nat, weight_alt_aux t1 a = weight_alt_aux t1 0 + a
IHt2 : forall a : nat, weight_alt_aux t2 a = weight_alt_aux t2 0 + a
a : nat
============================
weight_alt_aux t1 0 + (weight_alt_aux t2 0 + a) = weight_alt_aux t1 0 + weight_alt_aux t2 0 + a
```

Finally we can rewrite the goal using `Nat.add_assoc` to complete the inductive case.

With that, our proof is complete.

```
Lemma about_weight_alt_aux :
  forall (t : binary_tree)
         (a : nat),
    weight_alt_aux t a = weight_alt_aux t 0 + a.
Proof.
  intro t.
  induction t as [n | t1 IHt1 t2 IHt2].
  + intro a.
    rewrite -> (fold_unfold_weight_alt_aux_Leaf n a).
    rewrite -> (fold_unfold_weight_alt_aux_Leaf n 0).
    Search (_ + _ + _ = _ + _ + _).
    rewrite -> Nat.add_shuffle0.
    rewrite -> (Nat.add_0_r (n + a)).
    reflexivity.
  + intro a.
    rewrite -> 2 fold_unfold_weight_alt_aux_Node.
    rewrite -> (IHt1 (weight_alt_aux t2 a)).
    rewrite -> (IHt1 (weight_alt_aux t2 0)).
    rewrite -> (IHt2 a).
    rewrite -> Nat.add_assoc.
    reflexivity.
Qed.
```

Next, we must determine if our Eureka lemma is expressible using `make_Eureka_lemma`.

Similar to the factorial function, we supply the arguments for the binary tree type, the identity element, the function to combine two elements, the auxiliary function, and the accumulator.

```
Check (forall (t : binary_tree) (a : nat), make_Eureka_lemma nat 0 Nat.add (weight_alt_aux t) a).
forall (t : binary_tree) (a : nat), make_Eureka_lemma nat 0 Nat.add (weight_alt_aux t) a
     : Prop
```

Formalizing this as an Eureka lemma, we have:

```
Lemma about_weight_alt_aux' :
  forall (t : binary_tree)
         (a : nat),
    make_Eureka_lemma nat 0 Nat.add (weight_alt_aux t) a.
```

To prove this, let us first unfold the definition of `make_Eureka_lemma`.

```
1 goal (ID 29)

============================
forall (t : binary_tree) (a : nat), weight_alt_aux t a = weight_alt_aux t 0 + a
```

Which is precisely the same as the Eureka lemma we proved earlier. Hence, the proof is identical. Furthermore, we can use the `exact` tactic on the lemma we proved earlier to complete the proof.

```
Lemma about_weight_alt_aux' :
  forall (t : binary_tree)
         (a : nat),
    make_Eureka_lemma nat 0 Nat.add (weight_alt_aux t) a.
Proof.
 intros t.
 exact (about_weight_alt_aux t).
Qed.
```

Hence, we have shown that the Eureka lemma we postulated for the `weight` function is expressible using the `make_Eureka_lemma`.

Lastly, we must show that the two functions `weight` and `weight_alt` are equivalent in terms of their results.

The theorem states:

```
Theorem weight_and_weight_alt_are_equivalent :
  forall t : binary_tree,
    weight t = weight_alt t.
```

Firstly, we must unfold the definition of `weight_alt`.

```
1 goal (ID 34)

t : binary_tree
============================
weight t = weight_alt_aux t 0
```

However, with that our proof no longer follows the structure for the program as we are no longer proving the equality of `weight t` and `weight_alt t` but instead `weight t` and `weight_alt_aux t 0`. As such, let us postulate the following lemma:

```
Lemma about_weight_and_weight_alt_aux :
  forall (t : binary_tree),
    weight t = weight_alt_aux t 0.
```

To prove this lemma, we can proceed by induction on `t`.

Note that we can proceed with induction because both the LHS and RHS are recursive. So naturally it is reasonable to reason about them using induction.

The base case is as follows:

```
1 goal (ID 37)

n : nat
============================
weight (Leaf n) = weight_alt_aux (Leaf n) 0
```

Solving this is a simple rewrite of the fold-unfold lemma `fold_unfold_weight_alt_aux_Leaf` and a rewrite using `Nat.add_0_r`.

Proceeding with the inductive case, we have:

```
1 goal (ID 42)

t1, t2 : binary_tree
IHt1 : weight t1 = weight_alt_aux t1 0
IHt2 : weight t2 = weight_alt_aux t2 0
============================
weight (Node t1 t2) = weight_alt_aux (Node t1 t2) 0
```

We can then rewrite the goal using the fold-unfold lemmas we have been provided with `fold_unfold_weight_Node` and `fold_unfold_weight_alt_aux_Node`.

```
1 goal (ID 48)

t1, t2 : binary_tree
IHt1 : weight t1 = weight_alt_aux t1 0
IHt2 : weight t2 = weight_alt_aux t2 0
============================
weight t1 + weight t2 = weight_alt_aux t1 (weight_alt_aux t2 0)
```

Here, observe that we have an instance of the Eureka lemma we proved earlier `about_weight_alt_aux`. As such, we can rewrite the goal using the lemma.

```
1 goal (ID 49)

t1, t2 : binary_tree
IHt1 : weight t1 = weight_alt_aux t1 0
IHt2 : weight t2 = weight_alt_aux t2 0
============================
weight t1 + weight t2 = weight_alt_aux t1 0 + weight_alt_aux t2 0
```

We can conclude the proof by applying the induction hypothesis `IHt1` and `IHt2` respectively.

As we used our Eureka lemma from earlier, we notice that there is a connection with what we did prior.

Here is the full proof:

```
Lemma about_weight_and_weight_alt_aux :
  forall (t : binary_tree),
    weight t = weight_alt_aux t 0.
Proof.
  intro t.
  Check (about_weight_alt_aux t 0).
  induction t as [n | t1 IHt1 t2 IHt2].
  + rewrite -> fold_unfold_weight_Leaf.
    rewrite -> fold_unfold_weight_alt_aux_Leaf.
    rewrite -> Nat.add_0_r.
    reflexivity.
  + rewrite -> fold_unfold_weight_Node.
    rewrite -> fold_unfold_weight_alt_aux_Node.
    rewrite -> about_weight_alt_aux.
    rewrite <- IHt1.
    rewrite <- IHt2.
    reflexivity.
Qed.
```

We can then use this lemma to prove the theorem `weight_and_weight_alt_are_equivalent` using the `exact` tactic.

```
Theorem weight_and_weight_alt_are_equivalent :
  forall t : binary_tree,
    weight t = weight_alt t.
Proof.
  unfold weight_alt.
  intro t.
  exact (about_weight_and_weight_alt_aux t).
Qed.
```

6.3 Conclusion

In this exercise, we have shown that the Eureka lemma we postulated for the `weight` function is expressible using the `make_Eureka_lemma`. Furthermore, we have shown that the two functions `weight` and `weight_alt` are equivalent in terms of their results.

My main takeaway was structuring proofs in a way that follows the structure of the program and writing programs that follow the structure of the proof. This is because it makes the proof easier to understand and follow.

Furthermore, we also solidify the notion that Eureka lemmas are an instance of the same general idea. In particular, we can express all Eureka lemmas using the `make_Eureka_lemma`.

7 Exercise 12

7.1 Introduction

Since we have already seen how an induction principle with 2 base cases and induction hypotheses `nat_ind2` and one with 3 base cases and induction hypotheses `nat_ind3` and their applications, in this exercise, we design and implement the next induction principle `nat_ind4`, that has 4 base cases and 4 induction hypotheses, and prove its correctness.

7.2 Answer

We know that `nat_ind4` has 4 base cases and 4 induction hypotheses, so it is stated in the following way:

```
Lemma nat_ind4 :
  forall P : nat -> Prop,
    P 0 ->
    P 1 ->
    P 2 ->
    P 3 ->
    (forall n : nat, P n -> P (S n) -> P (S (S n)) -> P (S (S (S n))) -> P (S (S (S (S n))))) ->
    forall n : nat, P n.
```

We prove it using the standard induction principle first.

```
Proof.
  intros P P_0 P_1 P_2 P_3 P_SSSS.
  assert (all :
           forall m : nat,
             P m /\ P (S m) /\ P (S (S m)) /\ P (S (S (S m)))).
  { intro m.
    induction m as [ | m' [IHm' [IHSm' [IHSSm' IHSSSm']]]].
    - exact (conj P_0 (conj P_1 (conj P_2 P_3))).
    - exact (conj IHSm' (conj IHSSm' (conj IHSSSm' (P_SSSS m' IHm' IHSm' IHSSm' IHSSSm')))). }
  intro n.
  destruct (all n) as [ly _].
  exact ly.
```

In this case, in the `all` lemma, since there are 4 induction hypotheses and we are doing first-order induction, we need 3 conjunctions, and the rest is straightforward.

Next, we try proving using `nat_ind2`.

```
  Restart.

  intros P P_0 P_1 P_2 P_3 P_SSSS.
  assert (all :
           forall m : nat,
             P m /\ P (S m) /\ P (S (S m))).
  { intro m.
    induction m as [ | | m' [IHm' [IHSm' IHSSm']] [_ [_ IHSSSm']]] using nat_ind2.
    - exact (conj P_0 (conj P_1 P_2)).
    - exact (conj P_1 (conj P_2 P_3)).
    - exact (conj IHSSm' (conj IHSSSm' (P_SSSS m' IHm' IHSm' IHSSm' IHSSSm'))). }
  intro n.
  destruct (all n) as [ly _].
  exact ly.
```

Since `nat_ind2` has 2 base cases and 2 induction hypotheses, we only need 2 conjunctions for the `all` lemma, and the rest is similar to the proof using `nat_ind2`.

Finally, we complete the proof using `nat_ind3`.

```
  Restart.

  intros P P_0 P_1 P_2 P_3 P_SSSS.
  assert (all :
           forall m : nat,
             P m /\ P (S m)).
  { intro m.
    induction m as [ | | | m' [IHm' IHSm'] [_ IHSSm'] [_ IHSSSm']] using nat_ind3.
    - exact (conj P_0 P_1).
    - exact (conj P_1 P_2).
    - exact (conj P_2 P_3).
    - exact (conj IHSSSm' (P_SSSS m' IHm' IHSm' IHSSm' IHSSSm')). }
  intro n.
  destruct (all n) as [ly _].
  exact ly.
Qed.
```

Since `nat_ind3` has 3 base cases and 3 induction hypotheses, we only need 1 conjunction for the `all` lemma, and the proof is short and concise using conjunctions of the induction hypotheses.

7.3 Conclusion

In this exercise we implement and prove `nat_ind4` using the three induction principles with 1, 2 and 3 base cases and induction hypotheses respectively, and we can see that how we construct the `all` lemma matches the induction principle we use.

8 Exercise 13

8.1 Introduction

In this exercise we see an application of the fourth-order structural induction over Peano numbers which we implemented in the previous exercise. More specifically, we prove the following property using `nat_ind4`.

```
Property fours_and_fives :
  forall n : nat,
  exists a b : nat,
    12 + n = 4 * a + 5 * b.
```

8.2 Answer

We begin the induction proof using `nat_ind4`.

```
Proof.
  intro n.
  induction n as [ | | | | n' [a [b IHn']] _ _ _] using nat_ind4.
  - exists 3, 0.
    compute; reflexivity.
  - exists 2, 1.
    compute; reflexivity.
  - exists 1, 2.
    compute; reflexivity.
  - exists 0, 3.
    compute; reflexivity.
  - exists (S a), b.
```

We observe that in the induction hypotheses, we have the cases for `n'`, `S n'`, `S (S n')` and `S (S (S n'))`, so in the last induction step, we need to prove the property for `S (S (S (S n')))`. Therefore, compared to the case for `n'`, we only need one more 4 on the right-hand side and keep the number of fives constant, which implies that we will only be using the first induction hypothesis in the induction step. Hence, we do not need to name the rest of the induction hypotheses when doing the induction.

For the 4 base cases, we can just figure out the values of `a` and `b` and compute to complete the proofs. For the induction step, since we instantiate `a'` with `S a`, we have `4 * S a` on the right-hand side, so the following lemma will come in handy.

```
Lemma four_times_succ :
  forall n : nat,
    S (S (S (S (4 * n)))) = 4 * S n.
Proof.
  intro n.
  rewrite -> (plus_n_O (4 * n)).
  rewrite ->4 plus_n_Sm.
  rewrite -> (plus_n_O n) at 2.
  rewrite -> plus_n_Sm.
  rewrite -> Nat.mul_add_distr_l.
  simpl (4 * 1).
  reflexivity.
Qed.
```

The proof is completed simply using properties of addition, multiplication and successor. With the help of this lemma, we can proceed with the proof of our property.

```
  - exists (S a), b.
    rewrite <- four_times_succ.
    rewrite -> (plus_n_O n').
    rewrite ->4 plus_n_Sm.
    rewrite -> (plus_n_O (4 * a)).
    rewrite ->4 plus_n_Sm.
    rewrite -> Nat.add_assoc.
    rewrite <- (Nat.add_assoc (4 * a) 4 (5 * b)).
    rewrite -> (Nat.add_comm 4 (5 * b)).
    rewrite -> Nat.add_assoc.
    destruct (Nat.add_cancel_r (12 + n') (4 * a + 5 * b) 4) as [_ H_tmp].
    exact (H_tmp IHn').
Qed.
```

The rest of the proof is completed by using properties of addition and successors.

Here is the full proof:

```
Property fours_and_fives :
  forall n : nat,
  exists a b : nat,
    12 + n = 4 * a + 5 * b.
Proof.
  intro n.
  induction n as [ | | | | n' [a [b IHn']] _ _ _] using nat_ind4.
  - exists 3, 0.
    compute; reflexivity.
  - exists 2, 1.
    compute; reflexivity.
  - exists 1, 2.
    compute; reflexivity.
  - exists 0, 3.
    compute; reflexivity.
  - exists (S a), b.
    rewrite <- four_times_succ.
    rewrite -> (plus_n_O n').
    rewrite ->4 plus_n_Sm.
    rewrite -> (plus_n_O (4 * a)).
    rewrite ->4 plus_n_Sm.
    rewrite -> Nat.add_assoc.
    rewrite <- (Nat.add_assoc (4 * a) 4 (5 * b)).
    rewrite -> (Nat.add_comm 4 (5 * b)).
    rewrite -> Nat.add_assoc.
    destruct (Nat.add_cancel_r (12 + n') (4 * a + 5 * b) 4) as [_ H_tmp].
    exact (H_tmp IHn').
Qed.
```

8.3 Conclusion

This exercise gives us an application of `nat_ind4`, which is because if we had proven the cases for `n'`, `S n'`, `S (S n')` and `S (S (S n'))`, then it's easy to prove the induction step since we can just increment `a` by 1 and the pattern repeats. The important thing to note here is that to prove the case for `S (S (S (S n')))`, we only need to know the case for `n'`, since we only need to increase the value of `a` by 1 and keep `b` constant, so only the first induction hypothesis is needed. Additionally, we see how choosing the induction principle which aligns with the recursion pattern makes the proof simple and elegant.

9 Exercise 14

9.1 Introduction

Whereas in Exercise 13 we used higher-order induction via nat_ind4, in this exercise we prove fours_and_fives via mathematical induction (i.e., first-order induction).

9.2 Answer

To solve this exercise, let us begin as instructed by the exercise and induct the property on n.

Observing the base case, we have:

```
1 goal (ID 273)

============================
exists a b : nat, 12 + 0 = 4 * a + 5 * b
```

which requires us to find values for a and b such that 12 + 0 = 4 * a + 5 * b. We can easily see that a = 3 and b = 0 satisfies this equation. Hence, we can use the `exists` tactic to instantiate the existential variables and use the `reflexivity` tactic to complete the proof of the base case.

Moving on to the inductive case, we have:

```
1 goal (ID 276)

n' : nat
IHn' : exists a b : nat, 12 + n' = 4 * a + 5 * b
============================
exists a b : nat, 12 + S n' = 4 * a + 5 * b
```

We observe that the inductive hypothesis is an existential statement. Hence, we can use the `destruct` tactic to name the two conjuncts of the existential statement accordingly.

```
1 goal (ID 291)

n', a, b : nat
Hn' : 12 + n' = 4 * a + 5 * b
============================
exists a0 b0 : nat, 12 + S n' = 4 * a0 + 5 * b0
```

Afterwards, we can proceed the proof by splitting into cases.

```
4 goals (ID 316)

n' : nat
Hn' : 12 + n' = 5 * 0
============================
exists a b : nat, 12 + S n' = 4 * a + 5 * b

goal 2 (ID 322) is:
exists a b : nat, 12 + S n' = 4 * a + 5 * b
goal 3 (ID 328) is:
exists a b : nat, 12 + S n' = 4 * a + 5 * b
goal 4 (ID 330) is:
exists a b : nat, 12 + S n' = 4 * a + 5 * b
```

Proceeding to the first case, we have:

```
1 goal (ID 316)

n' : nat
Hn' : 12 + n' = 5 * 0
============================
exists a b : nat, 12 + S n' = 4 * a + 5 * b
```

Which we solve using the `discriminate` tactic as follows:

```
rewrite -> (Nat.mul_0_r 5) in Hn'.
Search (_ + _ = 0).
Check (Plus.plus_is_O_stt 12 n').
Check (Plus.plus_is_O_stt 12 n' Hn').
assert (H_absurd := (Plus.plus_is_O_stt 12 n' Hn')).
destruct H_absurd as [H_twelve_equals_zero H_n'_equals_zero].
Search (S _ = _).
discriminate H_twelve_equals_zero.
```

Likewise the next two cases can be solved similarly:

```
1 goal (ID 322)

n' : nat
Hn' : 12 + n' = 5 * 1
============================
exists a b : nat, 12 + S n' = 4 * a + 5 * b

(*
rewrite -> (Nat.mul_1_r 5) in Hn'.
Search (_ + _ =  _).
rewrite -> 5 plus_Sn_m in Hn'.
Search (S _ = S _ -> _ = _).
Check (Nat.succ_inj ((11 + n')) 4 Hn').
assert (H1 := Nat.succ_inj ((S (S (S (S (7 + n')))))) 4 Hn').
assert (H2 := Nat.succ_inj ((S (S (S (7 + n'))))) 3 H1).
assert (H3 := Nat.succ_inj ((S (S (7 + n')))) 2 H2).
assert (H4 := Nat.succ_inj ((S (7 + n'))) 1 H3).
assert (H5 := Nat.succ_inj ((7 + n')) 0 H4).
Check (Plus.plus_is_O_stt 7 n').
assert (H_absurd := (Plus.plus_is_O_stt 7 n' H5)).
destruct H_absurd as [H_seven_equals_zero H_n'_equals_zero].
discriminate  H_seven_equals_zero.
*)
```

And also

```
1 goal (ID 328)

n' : nat
Hn' : 12 + n' = 5 * 2
============================
exists a b : nat, 12 + S n' = 4 * a + 5 * b

(*
rewrite -> (Nat.mul_comm 5 2) in Hn'.
unfold Nat.mul in Hn'.
rewrite -> 10 plus_Sn_m in Hn'.
rewrite -> (Nat.add_0_r 5) in Hn'.
simpl (5 + 5) in Hn'.
assert (H1 := Nat.succ_inj (S (S (S (S (S (S (S (S (S (2 + n')))))))))) 9 Hn').
assert (H2 := Nat.succ_inj (S (S (S (S (S (S (S (S (2 + n'))))))))) 8 H1).
assert (H3 := Nat.succ_inj (S (S (S (S (S (S (S (2 + n')))))))) 7 H2).
assert (H4 := Nat.succ_inj (S (S (S (S (S (S (2 + n'))))))) 6 H3).
assert (H5 := Nat.succ_inj (S (S (S (S (S (2 + n')))))) 5 H4).
assert (H6 := Nat.succ_inj (S (S (S (S (2 + n'))))) 4 H5).
assert (H7 := Nat.succ_inj (S (S (S (2 + n')))) 3 H6).
assert (H8 := Nat.succ_inj (S (S (2 + n'))) 2 H7).
assert (H9 := Nat.succ_inj (S (2 + n')) 1 H8).
assert (H10 := Nat.succ_inj (2 + n') 0 H9).
assert (H_absurd := (Plus.plus_is_O_stt 2 n' H10)).
destruct H_absurd as [H_two_equals_zero H_n'_equals_zero].
discriminate H_two_equals_zero.
*)
```

However, this method is not very elegant. We realize we can use the lemma `five_times_succ`:

```
Lemma five_times_succ :
  forall n : nat,
    S (S (S (S (S (5 * n))))) = 5 * S n.
Proof.
  intro n.
  rewrite -> (plus_n_O (5 * n)).
  rewrite ->5 plus_n_Sm.
  rewrite -> (plus_n_O n) at 2.
  rewrite -> plus_n_Sm.
  rewrite -> Nat.mul_add_distr_l.
  simpl (5 * 1).
  reflexivity.
Qed.
```

Which is similar to the lemma `four_times_succ` we used in Exercise 13.

Using this lemma, we can solve the remaining proof as follows:

```
1 goal (ID 330)

n', b''' : nat
Hn' : 12 + n' = 5 * S (S (S b'''))
============================
exists a b : nat, 12 + S n' = 4 * a + 5 * b

(*
exists 4.
exists b'''.
rewrite <-  Nat.add_succ_comm.
rewrite -> plus_Sn_m.
rewrite -> Hn'.
Search (S _ + _ = S (_ + _)).
rewrite <- 3 five_times_succ.
rewrite <- (Nat.add_1_l (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (5 * b'''))))))))))))))))).
rewrite <- 15 Nat.add_succ_comm.
reflexivity.
*)
```

And finally

```
1 goal (ID 303)

n', a', b : nat
Hn' : 12 + n' = 4 * S a' + 5 * b
============================
exists a b0 : nat, 12 + S n' = 4 * a + 5 * b0

(*
exists a'.
exists (S b).
Search (S _ = _).
rewrite <- (Nat.add_1_r n').
rewrite -> Nat.add_assoc.
rewrite -> Hn'.
rewrite <- five_times_succ.
rewrite <- four_times_succ.
rewrite <- (Nat.add_assoc (S (S (S (S (4 * a'))))) (5 * b) 1).
rewrite -> (Nat.add_comm (5 * b) 1).
rewrite -> (Nat.add_assoc (S (S (S (S (4 * a'))))) 1 (5 * b)).
rewrite -> (Nat.add_succ_comm (S (S (S (4 * a')))) 1).
rewrite -> (Nat.add_succ_comm (S (S (4 * a'))) 2).
rewrite -> (Nat.add_succ_comm (S (4 * a')) 3).
rewrite -> (Nat.add_succ_comm (4 * a') 4).
rewrite <- (Nat.add_1_l (S (S (S (S (5 * b)))))).
rewrite <- (Nat.add_succ_comm 1 (S (S (S (5 * b))))).
rewrite <- (Nat.add_succ_comm 2 (S (S (5 * b)))).
rewrite <- (Nat.add_succ_comm 3 (S (5 * b))).
rewrite <- (Nat.add_succ_comm 4 (5 * b)).
rewrite -> (Nat.add_assoc (4 * a') 5 (5 * b)).
reflexivity.
*)
```

Here is the full proof for reference:

```
Lemma five_times_succ :
  forall n : nat,
    S (S (S (S (S (5 * n))))) = 5 * S n.
```

```
Proof.
  intro n.
  rewrite -> (plus_n_O (5 * n)).
  rewrite ->5 plus_n_Sm.
  rewrite -> (plus_n_O n) at 2.
  rewrite -> plus_n_Sm.
  rewrite -> Nat.mul_add_distr_l.
  simpl (5 * 1).
  reflexivity.
Qed.

Property fours_and_fives_via_mathematical_induction :
  forall n : nat,
  exists a b : nat,
    12 + n = 4 * a + 5 * b.
Proof.
  intro n.
  induction n as [ | n' IHn'].
  + exists 3.
    exists 0.
    compute. (* the remaining goals are only computational steps *)
    reflexivity.
  + destruct IHn' as [a [b Hn']].
    case a as [ | a'].
    ++ rewrite -> (Nat.mul_0_r 4) in Hn'.
       rewrite -> (Nat.add_0_l (5 * b)) in Hn'.
       case b as [ | [ | [ | b''']]].
       +++ rewrite -> (Nat.mul_0_r 5) in Hn'.
           Search (_ + _ = 0).
           Check (Plus.plus_is_O_stt 12 n').
           Check (Plus.plus_is_O_stt 12 n' Hn').
           assert (H_absurd := (Plus.plus_is_O_stt 12 n' Hn')).
           destruct H_absurd as [H_twelve_equals_zero H_n'_equals_zero].
           Search (S _ = _).
           discriminate H_twelve_equals_zero.
       +++ rewrite -> (Nat.mul_1_r 5) in Hn'.
           Search (_ + _ =  _).
           rewrite -> 5 plus_Sn_m in Hn'.
           Search (S _ = S _ -> _ = _).
           Check (Nat.succ_inj ((11 + n')) 4 Hn').
           assert (H1 := Nat.succ_inj ((S (S (S (S (7 + n')))))) 4 Hn').
           assert (H2 := Nat.succ_inj ((S (S (S (7 + n'))))) 3 H1).
           assert (H3 := Nat.succ_inj ((S (S (7 + n')))) 2 H2).
           assert (H4 := Nat.succ_inj ((S (7 + n'))) 1 H3).
           assert (H5 := Nat.succ_inj ((7 + n')) 0 H4).
           Check (Plus.plus_is_O_stt 7 n').
           assert (H_absurd := (Plus.plus_is_O_stt 7 n' H5)).
           destruct H_absurd as [H_seven_equals_zero H_n'_equals_zero].
           discriminate  H_seven_equals_zero.
       +++ rewrite -> (Nat.mul_comm 5 2) in Hn'.
           unfold Nat.mul in Hn'.
           rewrite -> 10 plus_Sn_m in Hn'.
           rewrite -> (Nat.add_0_r 5) in Hn'.
           simpl (5 + 5) in Hn'.
           assert (H1 := Nat.succ_inj (S (S (S (S (S (S (S (S (S (2 + n')))))))))) 9 Hn').
           assert (H2 := Nat.succ_inj (S (S (S (S (S (S (S (S (2 + n'))))))))) 8 H1).
           assert (H3 := Nat.succ_inj (S (S (S (S (S (S (S (2 + n')))))))) 7 H2).
           assert (H4 := Nat.succ_inj (S (S (S (S (S (S (2 + n'))))))) 6 H3).
           assert (H5 := Nat.succ_inj (S (S (S (S (S (2 + n')))))) 5 H4).
           assert (H6 := Nat.succ_inj (S (S (S (S (2 + n'))))) 4 H5).
           assert (H7 := Nat.succ_inj (S (S (S (2 + n')))) 3 H6).
           assert (H8 := Nat.succ_inj (S (S (2 + n'))) 2 H7).
           assert (H9 := Nat.succ_inj (S (2 + n')) 1 H8).
           assert (H10 := Nat.succ_inj (2 + n') 0 H9).
           assert (H_absurd := (Plus.plus_is_O_stt 2 n' H10)).
           destruct H_absurd as [H_two_equals_zero H_n'_equals_zero].
           discriminate H_two_equals_zero.
       +++ exists 4.
           exists b'''.
           rewrite <-  Nat.add_succ_comm.
           rewrite -> plus_Sn_m.
           rewrite -> Hn'.
```

```
            Search (S _ + _ = S (_ + _)).
            rewrite <- 3 five_times_succ.
            rewrite <- (Nat.add_1_l (S (S (S (S (S (S (S (S (S (S (S (S (S (S (5 * b''')))))))))))
                ))))))).
            rewrite <- 15 Nat.add_succ_comm.
            reflexivity.
    ++ exists a'.
       exists (S b).
       Search (S _ = _).
       rewrite <- (Nat.add_1_r n').
       rewrite -> Nat.add_assoc.
       rewrite -> Hn'.
       rewrite <- five_times_succ.
       rewrite <- four_times_succ.
       rewrite <- (Nat.add_assoc (S (S (S (S (4 * a'))))) (5 * b) 1).
       rewrite -> (Nat.add_comm (5 * b) 1).
       rewrite -> (Nat.add_assoc (S (S (S (S (4 * a'))))) 1 (5 * b)).
       rewrite -> (Nat.add_succ_comm (S (S (S (4 * a')))) 1).
       rewrite -> (Nat.add_succ_comm (S (S (4 * a'))) 2).
       rewrite -> (Nat.add_succ_comm (S (4 * a')) 3).
       rewrite -> (Nat.add_succ_comm (4 * a') 4).
       rewrite <- (Nat.add_1_l (S (S (S (S (5 * b)))))).
       rewrite <- (Nat.add_succ_comm 1 (S (S (S (5 * b))))).
       rewrite <- (Nat.add_succ_comm 2 (S (S (5 * b)))).
       rewrite <- (Nat.add_succ_comm 3 (S (5 * b))).
       rewrite <- (Nat.add_succ_comm 4 (5 * b)).
       rewrite -> (Nat.add_assoc (4 * a') 5 (5 * b)).
       reflexivity.
Qed.
```

9.3 Conclusion

Here, the proof using mathematical induction is quite more involved. We realize the importance of choosing the right induction principle to simplify our proofs.

10 Exercise 25

10.1 Introduction

The two predicates `evenp` and `oddp` are so intertwined that they are often defined together. In this exercise, we are asked to prove the soundness and completeness of the two predicates. In particular, we first prove them separately, in the comfort of `nat_ind2`. After that, we prove them together in the comfort of mathematical induction.

We do this because it is important to use the suitable induction principle in a proof. Choosing an induction principle not suited to the problem will not take us very far.

10.2 Answer

We begin by proving the soundness and completeness of `evenp` using `nat_ind2`. While the proof may seem complex at first glance, it is logical and actionable as the induction principle follows the structure of the specification.

Another key takeaway from this proof is that when one of the assumptions contains `exists`, we can use the `destruct` tactic and name the two conjuncts accordingly. We can then search for functions in the library that has the corresponding structure and use the `rewrite` tactic. When the left-hand side and right-hand side of the Leibniz equality have different constructors, we can use the `discriminate` tactic.

The entire proof is shown below:

```
Theorem soundness_and_completeness_of_evenp :
  forall n : nat,
    evenp n = true <-> exists m : nat, n = 2 * m.
Proof.
  intro n.
  induction n as [ | | n' [IHn'_sound IHn'_complete] [IHSn'_sound IHSn'_complete]] using nat_ind2.
  - split.
    + intro H_true.
      exists 0.
      Search (_ * 0 = 0).
      rewrite -> (Nat.mul_0_r 2).
      reflexivity.
    + intro H_true.
      rewrite -> (fold_unfold_evenp_O).
      reflexivity.
  - split.
    + intro H_absurd.
      discriminate H_absurd.
    + rewrite -> (fold_unfold_evenp_S 0).
      rewrite -> (fold_unfold_oddp_O).
      intro H_absurd.
      destruct H_absurd as [m H_m].
      case m as [ | m'].
      ++ rewrite -> (Nat.mul_0_r 2) in H_m.
         discriminate H_m.
      ++ Search (_ * S _).
         Check (Nat.mul_succ_r 2 m').
         rewrite -> (Nat.mul_succ_r 2 m') in H_m.
         Search (_ + S _ = S _).
         rewrite ->2 Nat.add_succ_r in H_m.
         discriminate H_m.
  - split.
    + rewrite -> (fold_unfold_evenp_S (S n')).
      rewrite -> (fold_unfold_oddp_S n').
      intro H_true.
      Check (IHn'_sound H_true).
      destruct (IHn'_sound H_true) as [m H_m].
      rewrite -> H_m.
      Check (twice_S m).
      rewrite -> (twice_S m).
      exists (S m).
      reflexivity.
    + rewrite -> (fold_unfold_evenp_S (S n')).
      rewrite -> (fold_unfold_oddp_S n').
      intros [m H_m].
      apply IHn'_complete.
      case m as [ | m'].
      ++ rewrite -> (Nat.mul_0_r 2) in H_m.
         discriminate H_m.
      ++ rewrite <- (twice_S m') in H_m.
         rewrite -> Nat.mul_comm in H_m.
         injection H_m as H_n'.
         rewrite -> Nat.mul_comm in H_n'.
         rewrite -> H_n'.
         exists m'.
         reflexivity.
Qed.
```

It is also worth noting in the comments that tCPA might oversimplify our assumptions when we use the `injection` tactic. To overcome this, we can use the `remember` tactic to add Leibniz equalities in our assumptions and name them using `eqn`.

The rest of the proof is routine, using the fold-unfold lemmas of both `evenp` and `oddp`.

On the other hand, if we attempt to prove the soundness and completeness of `evenp` without using `nat_ind2`, we will soon discover that it is easy to get stuck.

```
Theorem soundness_and_completeness_of_evenp_messy :
  forall n : nat,
    evenp n = true <-> exists m : nat, n = 2 * m.
Proof.
  intro n.
  induction n as [ | n' [IHn'_sound IHn'_complete]].
  - split.
    + intro H_true.
      exists 0.
      Search (_ * 0 = 0).
      rewrite -> (Nat.mul_0_r 2).
      reflexivity.
    + intro H_true.
      rewrite -> (fold_unfold_evenp_0).
      reflexivity.
  - split.
    + rewrite -> (fold_unfold_evenp_S n').
      intro H_n'.
      case n' as [ | n''].
      * rewrite -> fold_unfold_oddp_0 in H_n'.
        discriminate H_n'.
      * rewrite -> fold_unfold_oddp_S in H_n'.
        rewrite -> fold_unfold_evenp_S in IHn'_sound.
        rewrite -> fold_unfold_evenp_S in IHn'_complete.
Abort.
```

At this point of the proof, the *goals* window reads:

```
1 goal (ID 88)

  n'' : nat
  IHn'_sound : oddp n'' = true ->
               exists m : nat, S n'' = 2 * m
  IHn'_complete : (exists m : nat, S n'' = 2 * m) ->
                  oddp n'' = true
  H_n' : evenp n'' = true
  ============================
  exists m : nat, S (S n'') = 2 * m
```

Here we discover that it is impossible to make use of the assumptions we have to simplify our goal. This again shows that using induction principles that befit the structure of programs is key to a successful proof.

Luckily, we have the help of `nat_ind2` at our disposal, and the soundness and completeness of `oddp` can be proved using the induction principle.

```
Theorem soundness_and_completeness_of_oddp :
  forall n : nat,
    oddp n = true <-> exists m : nat, n = S (2 * m).
Proof.
  intro n.
  induction n as [ | | n' [IHn'_sound IHn'_complete] [IHSn'_sound IHSn'_complete]] using nat_ind2.
  - split.
    + intro H_absurd.
      exists 0.
      Search (_ * 0 = 0).
      rewrite -> (Nat.mul_0_r 2).
      discriminate H_absurd.
    + intro H_absurd.
      rewrite -> (fold_unfold_oddp_O).
      destruct H_absurd as [m H_m].
      case m as [ | m'].
      ++ discriminate H_m.
      ++ discriminate H_m.
  - split.
    + intro H_true.
      exists 0.
      Search (_ * 0 = _).
      rewrite -> (Nat.mul_0_r 2).
      reflexivity.
    + intro H_true.
      destruct H_true as [m H_m].
      case m as [ | m'].
      ++ exact (fold_unfold_oddp_S 0).
      ++ exact (fold_unfold_oddp_S 0).
  - split.
    + rewrite -> (fold_unfold_oddp_S (S n')).
      rewrite -> (fold_unfold_evenp_S n').
      intro H_true.
      Check (IHn'_sound H_true).
      destruct (IHn'_sound H_true) as [m H_m].
      rewrite -> H_m.
      Check (twice_S m).
      rewrite -> (twice_S m).
      exists (S m).
      reflexivity.
    + rewrite -> (fold_unfold_oddp_S (S n')).
      rewrite -> (fold_unfold_evenp_S n').
      intros [m H_m].
      apply IHn'_complete.
      case m as [ | m'].
      ++ rewrite -> (Nat.mul_0_r 2) in H_m.
         discriminate H_m.
      ++ rewrite <- (twice_S m') in H_m.
         rewrite -> Nat.mul_comm in H_m.
         injection H_m as H_n'.
         rewrite -> Nat.mul_comm in H_n'.
         rewrite -> H_n'.
         exists m'.
         reflexivity.
Qed.
```

Notice that the proof is highly similar to the proof for `evenp`. This again shows that the two predicates are highly intertwined with each other.

Taking a step further, we can also prove the soundness and completeness of the two predicates together. After all, they are defined together in the first place. Here we can use the first order mathematical induction, and the proof goes through.

```
Theorem soundness_and_completeness_of_evenp_and_of_oddp :
  forall n : nat,
    (evenp n = true <-> exists m : nat, n = 2 * m)
    /\
      (oddp n = true <-> exists m : nat, n = S (2 * m)).
Proof.
  intro n.
  induction n as [ | n' [[IHn'_esound IHn'_ecomplete] [IHn'_osound IHn'_ocomplete]]].
  - split.
    +  split.
       ++ intro H_true.
          exists 0.
          rewrite -> (Nat.mul_0_r 2).
          reflexivity.
       ++ intro H_true.
```

```
          rewrite -> (fold_unfold_evenp_O).
          reflexivity.
    + split.
      ++ intro H_absurd.
         discriminate H_absurd.
      ++ intro H_absurd.
         rewrite -> (fold_unfold_oddp_O).
         destruct H_absurd as [m H_m].
         case m as [ | m'].
         +++ discriminate H_m.
         +++ discriminate H_m.
  - split.
    + split.
      ++ intro H_true.
         rewrite -> (fold_unfold_evenp_S n') in H_true.
         apply IHn'_osound in H_true.
         destruct H_true as [m H_m].
         rewrite -> H_m.
         rewrite -> (twice_S m).
         exists (S m).
         reflexivity.
      ++ rewrite -> (fold_unfold_evenp_S n').
         intros [m H_m].
         apply IHn'_ocomplete.
         case m as [ | m'].
         +++ rewrite -> (Nat.mul_0_r 2) in H_m.
             discriminate H_m.
         +++ rewrite <- (twice_S m') in H_m.
             rewrite -> Nat.mul_comm in H_m.
             injection H_m as H_n'.
             rewrite -> Nat.mul_comm in H_n'.
             rewrite -> H_n'.
             exists m'.
             reflexivity.
    + split.
      ++ intro H_true.
         rewrite -> (fold_unfold_oddp_S n') in H_true.
         apply IHn'_esound in H_true.
         destruct H_true as [m H_m].
         rewrite -> H_m.
         exists m.
         reflexivity.
      ++ rewrite -> (fold_unfold_oddp_S n').
         intros [m H_m].
         apply IHn'_ecomplete.
         case m as [ | m'].
         +++ exists 0.
             injection H_m.
             intro H_n.
             Search (_ * 0 = 0).
             rewrite -> (Nat.mul_0_r 2).
             exact H_n.
         +++ exists (S m').
             injection H_m.
             intro H_n.
             rewrite -> (Nat.add_0_r m') in H_n.
             Search (S (_ + _)).
             rewrite <- (plus_Sn_m m' (S m')) in H_n.
             rewrite -> H_n.
             exact (twice (S m')).
Qed.
```

This again highlights the importance of choosing the suitable induction principle when doing a proof by induction. Here since the two predicates are defined together, there is no need for an additional induction hypothesis when proving their soundness and completeness together.

10.3 Conclusion

To conclude, this exercise has been a telling example that demonstrates the importance of choosing the suitable induction principle for proofs by induction. Instead of starting the proof straightaway with excitement, one should pause and consider the structure of the program beforehand. Once we are in the right direction, the rest of the proof can follow in a relatively smooth manner.

Apart from that, one should always reflect on how the predicates are defined, as it may inform our way of approaching the proof. In this case, since the two predicates are defined together, their soundness and completeness can also be proved together in the comfort of mathematical induction.

11 Conclusion

To conclude, this week's exercises first ask us to reflect on the nature of Eureka lemmas written so far in the course. The exercises after have been an invitation to understand and reason about induction at a more "meta" level. All these allow us to zoom out and see the big picture in the induction proofs we have written. Our main summaries for the exercises this week are as follows:

- All Eureka lemmas are an instance of the same general idea. In particular, we can express all Eureka lemmas using the `make_Eureka_lemma`. This is because all Eureka lemmas have the same general structure.
- Structuring proofs in a way that follows the structure of the program and writing programs that follow the structure of the proof makes the proof easier to understand and follow.
- Choosing the suitable induction principle based on the structure of the program can of great help in induction proofs.
- When doing higher-order induction, we should recognize which induction hypothesis will be of use to prove the induction step.