YSC4217

Mechanised Reasoning

Week 1 Report

Extending the program processors to include multiplication and revisiting lists and binary trees

Alan Matthew (A024197B, alan.matthew@u.yale-nus.edu.sg)
Kim Young Il (A0207809Y, youngil.kim@u.yale-nus.edu.sg)
Vibilan Jayanth (A0242417L, vibilan@u.yale-nus.edu.sg)

September 3, 2024

Abstract

This report is divided into three sections in which we study the equivalence between compiling and running a source program and interpreting the source program. We first study this equivalence for the Plus operation, then examine the Times operation, and investigate a theorem relating the logical equivalence between flattening a mirrored binary tree and reversing a flattened tree. The report emphasizes the importance of evaluation order and its impact on observational equivalences.

Contents

0 Initial Introduction

This assignment explores a fundamental principle in the relationship between source arithmetic expressions and their mathematical counterparts: When quantified source arithmetic expressions are evaluated in the same order on both sides of an equivalence, the property holds as a corollary of the corresponding mathematical operation. Conversely, if the property doesn't hold, it is because the expressions are evaluated in different orders on each side. This principle serves as the cornerstone for our exploration of observational equivalences and properties of arithmetic expressions in our source language.

We begin by revisiting our implementation of stylized arithmetic expressions, now expanded to include multiplication. We then apply our key principle to study the observational equivalences of source arithmetic expressions for Plus, Times, and Literals. This demonstrates how properties like commutativity, associativity, and distributively either hold or fail based on the evaluation order of expressions. For properties that don't hold unconditionally, we examine the necessary assumptions to make them valid and prove their equivalence under these conditions. Furthermore, we applied the reasoning about order of evaluations to rtl evaluation and observed a mirrored structure in the rtl versions of the above properties.

We extend our exploration to binary trees, examining four versions of a theorem relating mirroring and flattening of binary trees to list reversal. This showcases how our key principle also applies to the equivalence between flattening a mirrored binary tree and reversing the contents of a flattened binary tree. Furthermore, each of these versions permutes between the accumulator and non-accumulator implementations for flattening a binary tree and reversing a list. Through these exercises, we revisit how to reason about accumulator-based functions.

1 Exercise 1a: About Plus

1.1 Introduction

Let us return to Plus and how it behaves when it is evaluated. We remember that by default, the operands are evaluated from left to right (unless one looks at the alternative rtl versions). Thus, referring to our `specification_of_evaluate`, if evaluating the first `arithmetic_expression` returns an `Expressible_msg`, the same message is returned, without evaluating the second `arithmetic_expression`. Only if the first expression evaluates to an `Expressible_nat` is the second expression evaluated. Again, if the second expression evaluates to an `Expressible_msg`, that message is thrown, but if it evaluates to an `Expressible_nat`, addition is performed on both numbers.

While the source language resembles arithmetic expressions, an important question arises: do source programs behave like arithmetic expressions and possess their properties? Thanks to our implementation of the interpreter, we can address this question. The interpreter is defined as:

```
Definition interpret (sp : source_program) : expressible_value :=
  match sp with
  | Source_program ae => evaluate ae
  end.
```

We see that when our source program is interpreted, it must be evaluated by our evaluate function. Unlike actual arithmetic expressions, which do not return errors, our evaluate function does return errors, in the form of strings. Therefore, traditional properties about arithmetic expressions may not hold for our program.

1.2 Is `(Literal 0)` neutral on the left and/or right of `Plus`?

We first begin by investigating if `Literal 0` is neutral for Plus on the left (when we evaluate Plus from left to right).

```
Proposition Literal_0_is_neutral_for_Plus_on_the_left :
  forall ae : arithmetic_expression,
    evaluate (Plus (Literal 0) ae) = evaluate ae.
Proof.
  intro ae.
  rewrite -> fold_unfold_evaluate_Plus.
  rewrite -> fold_unfold_evaluate_Literal.
  case (evaluate ae) as [n | s].
  - rewrite -> (Nat.add_0_l n).
    reflexivity.
  - reflexivity.
Qed.
```

This proof demonstrates that `(Literal 0)` is neutral on the left of `Plus`. Let's break down the proof step-by-step:

1. We start by introducing an arbitrary arithmetic expression: `ae`.
2. We then unfold the definition of `evaluate` for Plus and `Literal`. This reveals how `evaluate ae` can be evaluated as either a natural number or a message.
3. The proof proceeds by case analysis on the result of evaluating each expression:
   - We first consider the case for `Expressible_nat n`: we know that $0 + n = n$, and using `rewrite -> (Nat.add_0_l n).`, we clear the subcase.
   - We the consider the case for `Expressible_msg s`. Seeing an equality, we clear the subcase with `reflexivity`.

Apart from searching our brains for which tactics did what, that proof sailed smoothly. The proof is identical apart from changing `Nat.add_0_l n` to `Nat.add_0_r n` for neutrality for Plus on the right because the only difference between the two proofs is that in the subcase `Expressible_nat n`, `n` is on the RHS of the addition sign, not on the LHS of the addition sign.

This observation leads us to a key insight: the property that `Literal 0` is neutral on the left of `Plus` is proved thanks to two crucial factors. First, it relies on the property that 0 is neutral on the left of + (known as `Nat.add_0_l`). Second, and equally important, the error cases occur in the same way in both the LHS and the RHS of our expressions because there is only one thing ae to evaluate on both the LHS and RHS of the equality.

We also proved that `Literal 0` is neutral for Plus on the left and right when Plus evaluates from right to left. The proofs are identical except that we use the fold-unfold tactics to evaluate from right to left.

1.3 Is Plus Associative?

To investigate whether Plus is associative, we formulated and proved the following proposition:

```
Proposition Plus_is_associative :
  forall ae1 ae2 ae3 : arithmetic_expression,
    evaluate (Plus ae1 (Plus ae2 ae3)) = evaluate (Plus (Plus ae1 ae2) ae3).
Proof.
  intros ae1 ae2 ae3.
  rewrite ->4 fold_unfold_evaluate_Plus.
  case (evaluate ae1) as [n1 | s1].
  - case (evaluate ae2) as [n2 | s2].
    + case (evaluate ae3) as [n3 | s3].
      * rewrite -> Nat.add_assoc.
        reflexivity.
      * reflexivity.
    + reflexivity.
```

```
14    - reflexivity.
15  Qed.
```

This proof demonstrates that Plus is indeed associative for our arithmetic expressions. Let's break down the proof strategy:

1. We start by introducing three arbitrary arithmetic expressions: `ae1`, `ae2`, and `ae3`.
2. We then unfold the definition of `evaluate` for Plus four times using `rewrite ->4 fold_unfold_evaluate_Plus`. This exposes the internal structure of how Plus is evaluated.
3. The proof proceeds by case analysis on the result of evaluating each expression:
    - We first consider the case for `ae1`
    - Then, nested within that, we consider the case for `ae2`
    - Finally, we consider the case for `ae3`
4. The key step occurs when all three expressions evaluate to numbers (`Expressible_nat`). In this case, we use `rewrite -> Nat.add_assoc` to apply the associativity of addition for natural numbers.
5. In all other cases, where at least one expression evaluates to a string (`Expressible_msg`), the proof completes with `reflexivity` because the evaluation short-circuits and returns the first encountered message.

This proof reveals an important property of our Plus operation:

1. Associativity holds when all subexpressions evaluate to numbers, leveraging the associativity of natural number addition.
2. The short-circuiting behavior of Plus preserves associativity even when error messages are involved. If any subexpression evaluates to a message, both sides of the equality will produce the same result regardless of the grouping. This is because in both the LHS and the RHS, arithmetic expressions are evaluated in this order: `ae1 -> ae2 -> ae3`

This associativity property ensures that the grouping of operands does not affect the final result, allowing for flexibility in evaluation order without changing the overall behavior of the operation. This allows us to reason about evaluating the Plus operator from left to right in the same way when evaluating the Plus operator from right to left. This explains why our proof for the associativity of the Plus operator when we evaluate from right to left mirrors that of the proof for the associativity of the Plus operator when we evaluate from left to right, except that we use the fold-unfold tactics to evaluate from right to left.

1.4 Is Plus Commutative?

To investigate whether Plus is commutative, we formulated and attempted to prove the following proposition:

```
Proposition Plus_is_commutative :
  forall ae1 ae2 : arithmetic_expression,
    evaluate (Plus ae1 ae2) = evaluate (Plus ae2 ae1).
Proof.
  intros ae1 ae2.
  rewrite ->2 fold_unfold_evaluate_Plus.
  case (evaluate ae1) as [n1 | s1].
  - case (evaluate ae2) as [n2 | s2].
    + rewrite -> Nat.add_comm.
      reflexivity.
    + reflexivity.
  - case (evaluate ae2) as [n2 | s2].
    + reflexivity.
```

But, in `goals`, we run into the following problem:

```
1 subgoal (ID 543)

  ae1, ae2 : arithmetic_expression
  s1, s2 : string
  ============================
  Expressible_msg s1 = Expressible_msg s2
```

and we have no properties about either strings `s1` or `s2`, so plus is not commutative in general. Using `Compute`, we can also test with a basic case that Plus is not commutative.

```
Compute( let ae1 := (Minus (Literal 1) (Literal 3)) in
         let ae2 := (Minus (Literal 2) (Literal 3)) in
         evaluate (Plus ae1 ae2) = evaluate (Plus ae2 ae1)
).

(*
= Expressible_msg "numerical underflow: -2" =
       Expressible_msg "numerical underflow: -1"
     : Prop
*)
```

We use this test case to help us prove the following proposition.

```
Proposition Plus_is_not_commutative :
  exists ae1 ae2: arithmetic_expression,
    evaluate (Plus ae1 ae2) <> evaluate (Plus ae2 ae1).
Proof.
  exists (Minus (Literal 1) (Literal 3)).
  exists (Minus (Literal 2) (Literal 3)).
  compute.
  intro H_absurd.
  discriminate H_absurd.
Qed.
```

We can also prove the conditional property of commutativity. We see that if the error message from the first evaluation is different from the latter evaluation, the equality fails. We can capture the converse of this property in the following `Proposition`:

```
Proposition Plus_is_conditionally_commutative :
  forall ae1 ae2 : arithmetic_expression,
    ((exists n : nat,
        evaluate ae1 = Expressible_nat n)
     \/
       (exists n : nat,
           evaluate ae2 = Expressible_nat n)
     \/
       (exists s : string,
           evaluate ae1 = Expressible_msg s /\ evaluate ae2 = Expressible_msg
           ↪  s))
    <->
    evaluate (Plus ae1 ae2) = evaluate (Plus ae2 ae1).
Proof.
  intros ae1 ae2.
  split.
  - intros [[n1 H_n1] | [[n2 H_n2] | [s [H_s1 H_s2]]]].
    + rewrite ->2 fold_unfold_evaluate_Plus.
      rewrite -> H_n1.
      case (evaluate ae2) as [n2 | s2].
      * rewrite -> (Nat.add_comm n2 n1).
        reflexivity.
      * reflexivity.
    + rewrite ->2 fold_unfold_evaluate_Plus.
      rewrite -> H_n2.
      case (evaluate ae1) as [n1 | s1].
      * rewrite -> (Nat.add_comm n2 n1).
```

```
      reflexivity.
    * reflexivity.
  + rewrite ->2 fold_unfold_evaluate_Plus.
    rewrite -> H_s1.
    rewrite -> H_s2.
    reflexivity.
```

```
1 subgoal (ID 530)

  ae1, ae2 : arithmetic_expression
  ============================
  evaluate (Plus ae1 ae2) = evaluate (Plus ae2 ae1) ->
  (exists n : nat, evaluate ae1 = Expressible_nat n) \/
  (exists n : nat, evaluate ae2 = Expressible_nat n) \/
  (exists s : string,
     evaluate ae1 = Expressible_msg s /\
     evaluate ae2 = Expressible_msg s)
```

The forward direction is not difficult to prove. The backward direction requires reasoning with the following cases: When ae1 evaluates to n1, when ae1 evaluates to s1, but ae2 evaluates to n2, and when both ae1 and ae2 evaluate to a string (in this case s1 = s2). In total, 3 subcases are generated. We pick which branch of the disjunction to prove by looking at the type of the evaluation. We prove the subcases by using the following logic:

1. Show that there exists a natural number equal to `n1` or `n2`. (In the proof, we show that there exists such a number `n1`
2. Show that there exists a natural number equal to `n2`
3. Show that the message returned by the first evaluation is identical to that of the second evaluation.

Let us consider case 3 more closely:

```
1 subgoal (ID 667)

  ae1, ae2 : arithmetic_expression
  s1 : string
  H_ae1 : evaluate ae1 = Expressible_msg s1
  s2 : string
  H_ae2 : evaluate ae2 = Expressible_msg s2
  H_eq_s1_s2 : Expressible_msg s1 =
               Expressible_msg s2
```

```
  ============================
  (exists n : nat,
     Expressible_msg s1 = Expressible_nat n) \/
  (exists n : nat,
     Expressible_msg s2 = Expressible_nat n) \/
  (exists s : string,
     Expressible_msg s1 = Expressible_msg s /\
     Expressible_msg s2 = Expressible_msg s)
```

Because expressible messages cannot be expressible natural numbers, we have to prove the right-right branch. Focusing on it:

```
1 subgoal (ID 671)

  ae1, ae2 : arithmetic_expression
  s1 : string
  H_ae1 : evaluate ae1 = Expressible_msg s1
  s2 : string
  H_ae2 : evaluate ae2 = Expressible_msg s2
  H_eq_s1_s2 : Expressible_msg s1 =
               Expressible_msg s2
  ============================
  exists s : string,
    Expressible_msg s1 = Expressible_msg s /\
    Expressible_msg s2 = Expressible_msg s
```

We can prove one side of the conjunction using the `exists` tactic and the other using `H_eq_s1_s2`. We do exactly that:

```
      exists s1.
      split.
      * reflexivity.
      * exact (eq_sym H_eq_s1_s2).
Qed.
```

Furthermore, we repeated the same steps as above but this time using `evaluate_rtl` to show that the same property holds even when we evaluate Plus from right to left. The counterexample to show that evaluating Plus from right to left is not commutative, and the proof to show that evaluating Plus from right to left is conditionally commutative mirrors that of the evaluating Plus from left to right counterpart, except that we use the fold-unfold tactics to evaluate from right to left. The fundamental reason for the consistency in these properties across evaluation orders is the associativity of the Plus operator on our arithmetic expressions.

1.5 Conclusion

The first section allowed us to get our hands wet again with TCPA and all the key bindings we forgot over the semester. Furthermore, we learned how to re-express properties that did not always hold as conditional properties. That's something we did not do back in FPP. Lastly, with some guidance, we were able to prove the equivalence of the conditional properties. In doing so, we were able to observe how impossible goals we were trying to prove were actually the missing assumptions of conditional proofs we were trying to generate. Finally, we were able to observe that the assumptions for conditional properties were a result of the order of evaluation for our program. Assumptions for conditional statements were mirrored depending on the order of evaluation of the evaluator.

2 Exercise 1b: About Times

2.1 Introduction

After examining the properties of Plus, we now turn our attention to the Times operation in our arithmetic expressions. Similar to Plus, we expect Times to be associative, but not commutative (and thus conditionally commutative). Furthermore, we observe that even though times does not distribute over plus on the right, it actually is distributive over plus on the left. More on this later. We can apply the reasoning with order of evaluations we learned above when reasoning about properties of Times.

2.2 Is Times Associative/Commutative?

The proof for the associativity and commutativity properties of Times follows a similar structure to that of Plus, with some key distinctions. Times associativity is a result of two factors: first, the quantified three expressions are evaluated in the same order on both the left-hand side and right-hand side, and second, the natural number multiplication is associative (`Nat.mul_assoc`). Furthermore, Times is not generally commutative due to the quantified two expressions being evaluated in different orders on the LHS and RHS. However, it exhibits conditional commutativity when there is at most one error present. This conditional commutativity arises because the differing order of evaluation becomes irrelevant with at most one error (as opposed to two), and natural number multiplication is commutative (`Nat.mul_comm`).

2.3 Does Times Distribute on the Right of Plus?

After examining the associativity and commutativity of Times, we now turn our attention to another fundamental property: distributivity of Times over Plus. Specifically, we investigate whether Times distributes on the right of Plus.

We begin with an attempt to prove the following proposition:

```
Proposition Times_distributive_over_Plus_on_the_right :
  forall ae1 ae2 ae3 : arithmetic_expression,
    evaluate (Times (Plus ae1 ae2) ae3) =
    evaluate (Plus (Times ae1 ae3) (Times ae2 ae3)).
Proof.
  intros ae1 ae2 ae3.
  rewrite -> fold_unfold_evaluate_Times.
  rewrite ->2 fold_unfold_evaluate_Plus.
  rewrite ->2 fold_unfold_evaluate_Times.
  case (evaluate ae1) as [n1 | s1] eqn:Hae1.
  + case (evaluate ae2) as [n2 | s2] eqn:Hae2.
    ++ case (evaluate ae3) as [n3 | s3] eqn:Hae3.
       +++ rewrite -> Nat.mul_add_distr_r.
           reflexivity.
       +++ reflexivity.
```

```
    ++ case (evaluate ae3) as [n3 | s3] eqn:Hae3.
       +++ reflexivity.
+++ Abort. (* Expressible_msg s2 = Expressible_msg s3 *)
```

Just before aborting the proof, we encounter this subgoal:

```
1 subgoal

  ae1, ae2, ae3 : arithmetic_expression
  n1 : nat
  Hae1 : evaluate ae1 = Expressible_nat n1
  s2 : string
  Hae2 : evaluate ae2 = Expressible_msg s2
  s3 : string
  Hae3 : evaluate ae3 = Expressible_msg s3
  ======================== (1 / 1)
  Expressible_msg s2 = Expressible_msg s3
```

This subgoal reveals that Times is not universally distributive over Plus in our language. The issue arises when we encounter error messages, as the order of evaluation affects which error message is propagated. On the LHS, arithmetic expressions are evaluated in this following order: `ae1 -> ae2 -> ae3`. On the RHS, arithmetic expressions are evaluated in this following order: `ae1 -> ae3 -> ae2 -> ae3`, where evaluating ae3 again returns the same result as before.

We formally establish that Times is not distributive over Plus on the right, we prove the following proposition:

```
Proposition Times_is_not_distributive_over_Plus_on_the_right :
  exists ae1 ae2 ae3 : arithmetic_expression,
    evaluate (Times (Plus ae1 ae2) ae3) <>
    evaluate (Plus (Times ae1 ae3) (Times ae2 ae3)).
Proof.
  exists (Literal 0).
  exists (Minus (Literal 0) (Literal 5)).
  exists (Minus (Literal 2) (Literal 3)).
  compute.
  intro H_absurd.
  discriminate H_absurd.
Qed.
```

This proof provides a concrete counterexample where Times does not distribute over Plus on the right. The issue arises due to the left-to-right evaluation strategy and immediate error propagation, which leads to different error messages being produced and prop-

agated. In the left-hand expression, `(Times (Plus ae1 ae2) ae3)`, if `ae1` evaluates successfully but `ae2` contains an error, this error is encountered and propagated before `ae3` is ever evaluated. In the right-hand expression, `(Plus (Times ae1 ae3) (Times ae2 ae3))`, `ae1` and `ae3` are evaluated first in the left branch of the Plus. If both are successful, then `ae2` is evaluated, followed by `ae3` again in the right branch. If `ae3` contains an error, it would be propagated before reaching the potential error in `ae2`. This difference in evaluation order and error propagation results in different error messages being produced for certain inputs, thus breaking distributively.

However, Times does exhibit a form of conditional distributively over Plus. We can prove that Times is distributive over Plus on the right when all expressions evaluate to numbers:

```
Proposition Times_is_conditionally_distributive_over_Plus_on_the_right :
  forall ae1 ae2 ae3 : arithmetic_expression,
  forall n1 n2 n3 : nat,
  (evaluate ae1 = Expressible_nat n1 /\
   evaluate ae2 = Expressible_nat n2 /\
   evaluate ae3 = Expressible_nat n3) ->
    evaluate (Times (Plus ae1 ae2) ae3) =
    evaluate (Plus (Times ae1 ae3) (Times ae2 ae3)).
Proof.
  intros ae1 ae2 ae3 n1 n2 n3 [Hae1 [Hae2 Hae3]].
  rewrite -> fold_unfold_evaluate_Times.
  rewrite -> 2 fold_unfold_evaluate_Plus.
  rewrite -> 2 fold_unfold_evaluate_Times.
  rewrite -> Hae1, Hae2, Hae3.
  rewrite -> Nat.mul_add_distr_r.
  reflexivity.
Qed.
```

While the proposition above does hold, the restrictions on the evaluated results are too strict. For example, if `ae1` evaluates to a message, the property should hold, but it is not captured in the statement of the proposition. Our evaluates ltr, on the LHS, the order of evaluation is as follows: `ae1 -> ae2 -> ae3` And on the RHS, the order of evaluation is as follows: `ae1 -> ae3 -> ae2`. Thus, if either `ae2` or `ae3` evaluates to an error message, they must evaluate to the same error message, as they are evaluated in the same order. Let us formalize the statement:

```
Proposition Times_is_conditionally_distributive_over_Plus_on_the_right :
  forall ae1 ae2 ae3 : arithmetic_expression,
    (exists n : nat, evaluate ae2 = Expressible_nat n)
    \/
      (exists n : nat, evaluate ae3 = Expressible_nat n)
    \/
      (exists s : string,
```

```
       (evaluate ae2 = Expressible_msg s
       /\
       evaluate ae3 = Expressible_msg s) <->
      evaluate (Times (Plus ae1 ae2) ae3) =
        evaluate (Plus (Times ae1 ae3) (Times ae2 ae3)).
```

Yet the statement above (which we have spent a long time being unable to prove) does not capture the relationship fully. In fact, we forgot to include the case where evaluating `ae1` shortcircuits the evaluation due to the ltr nature of the program:

```
Proposition Times_is_conditionally_distributive_over_Plus_on_the_right :
  forall ae1 ae2 ae3 : arithmetic_expression,
    (exists n : nat, evaluate ae2 = Expressible_nat n)
    \/
      (exists n : nat, evaluate ae3 = Expressible_nat n)
    \/
      (exists s : string,
       (* either the error messages of
       ae2 and ae3 are the same *)
       (evaluate ae2 = Expressible_msg s
       /\
       evaluate ae3 = Expressible_msg s)
       \/
       (* or the error message is from ae1 *)
       evaluate ae1 = Expressible_msg s) <->
      evaluate (Times (Plus ae1 ae2) ae3) =
        evaluate (Plus (Times ae1 ae3) (Times ae2 ae3)).
```

Or the equivalent, but more logical statement:

```
Proposition Times_distributes_over_Plus_on_the_right_conditionally' :
    forall (ae1 ae2 ae3 : arithmetic_expression),
      (exists s : string,
           evaluate ae1 = Expressible_msg s)
      \/
      (exists n : nat,
          evaluate ae2 = Expressible_nat n)
      \/
      (exists n : nat,
          evaluate ae3 = Expressible_nat n)
      \/
      (exists s : string,
```

```
          evaluate ae2 = Expressible_msg s
          /\
          evaluate ae3 = Expressible_msg s)
      <->
      evaluate (Times (Plus ae1 ae2) ae3) =
      evaluate (Plus (Times ae1 ae3) (Times ae2 ae3)).

```

The proof itself is less interesting than the process in which we looked for the correct assumptions. If the assumptions were too weak (For example, in the second version of the proposition), we had an impossible goal in the goals tab, signaling us that a strong assumption had to be made. If assumptions were too strong, (for example, in the first version of the proposition, which was an implication, not an equality) the backward direction could not be proved.

2.4 Does Times Distribute on the Left of Plus?

After examining the distributivity of Times over Plus on the right, we now turn our attention to the left distributivity. We find that Times does indeed distribute on the left of Plus in our arithmetic expression language, without any conditions due to the order of evaluation. Let us examine the statement:

```
Proposition Times_distributive_over_Plus_on_the_left :
  forall ae1 ae2 ae3 : arithmetic_expression,
    evaluate (Times ae1 (Plus ae2 ae3)) =
    evaluate (Plus (Times ae1 ae2) (Times ae1 ae3)).
Qed.
```

Reasoning with order of evaluations, in both LHS and RHS, ae's are evaluated in this order: `ae1 -> ae2 -> ae3`, where the re-evaluation already evaluated expressions is idempotent and the re-evaluation of ae1 in the RHS is dropped when reasoning about the order of evaluation.

This proof demonstrates that Times is universally distributive over Plus on the left in our language. Let's break down the proof strategy:

1. We start by unfolding the definitions of `evaluate` for Times and Plus.
2. We then perform case analysis on the evaluation of `ae1`, `ae2`, and `ae3`.
3. When all expressions evaluate to numbers, we apply the left distributivity of multiplication over addition for natural numbers (`Nat.mul_add_distr_l`).
4. In all other cases, where at least one expression evaluates to an error message, both sides of the equation reduce to the same error message, and we can prove equality with `reflexivity`.

This result reveals some interesting properties about the interaction between Times and Plus in our language:

1. Times is universally distributive over Plus on the left, regardless of whether the expressions evaluate to numbers or error messages.

2. The order of evaluation plays a crucial role in this property. In the expression `(Times ae1 (Plus ae2 ae3))`, `ae1` is always evaluated first. If it results in an error, this error is propagated regardless of the values of `ae2` and `ae3`, which matches the behavior of both `(Times ae1 ae2)` and `(Times ae1 ae3)` in the distributed form.

3. This property holds even in the presence of errors, unlike the right distributivity we examined earlier.

The universal left distributivity of Times over Plus stands in contrast to the conditional right distributivity we observed earlier. This asymmetry highlights the importance of expression structure and evaluation order in our language. It demonstrates that seemingly similar properties can have quite different behaviors depending on the specific arrangement of operations.

In conclusion, the universal left distributivity of Times over Plus adds another layer to our understanding of the algebraic properties in our language. It demonstrates that while some properties hold conditionally or fail in the presence of errors, others hold universally due to the specific semantics and evaluation order of our language constructs.

2.5 Short aside about Distributivity of Times over Plus, for rtl evaluation.

Let us consider how the distributivity works in an rtl evaluator by reasoning about the order of evaluation. Starting with distributivity on the right:

```
Proposition Times_distributive_over_Plus_on_the_right_rtl :
  forall ae1 ae2 ae3 : arithmetic_expression,
    evaluate_rtl (Times (Plus ae1 ae2) ae3) =
    evaluate_rtl (Plus (Times ae1 ae3) (Times ae2 ae3)).
```

We see that the arithmetic expressions are evaluated in the following order:
`ae3 -> ae2 -> ae1` in both the LHS and the RHS (once again, re-evaluation of ae3 is idempotent in order of evaluation of the RHS). We prove that times does distribute over plus on the right unconditionally for rtl evaluators. Let us now consider distributivity on the left:

```
Proposition Times_distributive_over_Plus_on_the_left_rtl :
  forall ae1 ae2 ae3 : arithmetic_expression,
    evaluate_rtl (Times ae1 (Plus ae2 ae3)) =
    evaluate_rtl (Plus (Times ae1 ae2) (Times ae1 ae3)).
```

We see that the arithmetic expressions are evaluated in the following order:
`ae3 -> ae2 -> ae1` in the LHS and the following order :`ae3 -> ae1 -> ae2` in the RHS (second evaluation of `ae1` dropped due to idempotence of re-evaluation already evaluated expressions). Thus, reasoning with order of evaluations as we did above, we can come up with a counter example that disproves the unconditional statement above:

```
Compute (let ae1 := (Minus (Literal 1) (Literal 2)) in
         let ae2 := (Minus (Literal 1) (Literal 3)) in
         let ae3 := (Literal 0) in
         evaluate_rtl (Times ae1 (Plus ae2 ae3)) =
           evaluate_rtl (Plus (Times ae1 ae2) (Times ae1 ae3))).

(*
     = Expressible_msg "numerical underflow: -2" =
       Expressible_msg "numerical underflow: -1"
     : Prop
*)
```

Thus, we make a conditional statement about Times distributing over Plus if 1) the first ae to be evaluated evaluates to an error message, 2) if either ae1 or ae2 evaluates to a nat, 3) or if both evaluate to an error message, the error message is the same. We prove the above in the .v file.

In conclusion, we can say this about order of evaluation and distributivity of Times over Plus: for ltr evaluators, Times distributes over Plus on the left without condition. Conversely, for rtl evaluators, Times distributes over Plus on the right without condition.

2.6 Literal 1 as a Neutral Element for Times

Mutatis mutandis. The proof and the conversation is (almost) identical to that of section 1.2.

2.7 Literal 0 and Its Absorbing Property for Times

Next, we investigate whether Literal 0 has an absorbing property for Times.

```
Proposition Literal_0_is_absorbing_for_Times_on_the_left :
  forall ae : arithmetic_expression,
    evaluate (Times (Literal 0) ae) = evaluate (Literal 0).
Proof.
  intro ae.
  rewrite -> fold_unfold_evaluate_Times.
  rewrite -> fold_unfold_evaluate_Literal.
  case (evaluate ae) as [n | s].
  - rewrite -> (Nat.mul_0_l n).
    reflexivity.
```

```
  - (* not absorbing on the left. *)
Abort.

Proposition Literal_0_is_not_absorbing_for_Times_on_the_left :
  exists ae : arithmetic_expression,
    evaluate (Times (Literal 0) ae) <> evaluate (Literal 0).
Proof.
  exists (Minus (Literal 1) (Literal 3)).
  compute.
  intro H_absurd.
  discriminate H_absurd.
Qed.
```

Notice that regardless of whether `Literal 0` is on the left or the right of an arithmetic expression, if the arithmetic expression evaluates to an error the equality fails. Therefore, the property fails for both the left and the right. This means that the property can be conditionally re-written as:

```
Proposition Literal_0_is_conditionally_absorbing_for_Times_on_the_left :
  forall (ae : arithmetic_expression),
         (exists n, evaluate ae = Expressible_nat n)
         <->
         evaluate (Times (Literal 0) ae) = evaluate (Literal 0).
```

Mutatis mutandis. The key insight here is the same as that for Literal 0 is neutral on the left and right of `Plus` except that we use `(Nat.mul_0_l n)`.

1. Literal 0 is not universally absorbing for Times, either on the left or right.
2. We can construct counterexamples where multiplying by 0 does not result in 0, due to error propagation.
3. However, Literal 0 is conditionally absorbing for Times when the other operand evaluates to a number.

In conclusion, these additional properties provide a more complete picture of how Times behaves in our language, particularly with respect to special values like 0 and 1. They underscore the importance of considering both successful computations and error cases when reasoning about arithmetic expressions.

2.8 Conclusion

In conclusion, our investigation into the Times operation has revealed the following set of properties:

- Evaluation Order: Times follows strict left-to-right evaluation, which significantly impacts its behavior, especially in error handling.

- Associativity: Times is universally associative, maintaining this property even in the presence of errors. `ae1 -> ae2 -> ae3`.
- Commutativity: Times is conditionally commutative. It holds when at least one of the later two operands evaluates to a number, but fails when both operands have different error messages.
- Distributivity: Times exhibits asymmetric distributivity properties:
  - It is universally distributive over Plus on the left.
  - It is only conditionally distributive over Plus on the right, holding when the later two operands evaluate to numbers, or when they both evaluate to the same error message.
- Additional Properties:
  - Literal 1 is a universal neutral element for Times, both on the left and right.
  - Literal 0 is conditionally absorbing for Times, but this property breaks down when the other operand produces an error.

For rtl evaluation:

- Evaluation Order: It is the mirrored direction of ltr evaluators, and the assumptions for conditional properties are also mirrored.
- Associativity: Times is universally associative, even for rtl evaluation `ae3 -> ae2 -> ae1`.
- Commutativity: Times is conditionally commutative for rtl evaluation. It holds when at least one of the later two operands evaluates to a number, but fails when both operands have different error messages.
- Distributivity: Times exhibits asymmetric distributivity properties, which are mirrored versions of the ltr evaluator:
  - It is universally distributive over Plus on the right.
  - It is only conditionally distributive over Plus on the left, holding when the later two operands evaluate to numbers, or when they both evaluate to the same error message.
- Additional Properties: While not explicitly formalized and proved, we can conclude the following using our recent knowledge of order of evaluating arithmetic expressions for our program:
  - Literal 1 is a universal neutral element for Times, because on both the LHS and the RHS, one and the same ae are evaluated.
  - Litearl 0 is not universally absorbing for Times because if the ae evaluates to an error message, it cannot be equal to the side that evaluates to the expressible nat 0.

This section enables us to reapply the reasoning about the order of evaluations we observed for Plus in a slightly different setting. The two exercises sing the same song, just slightly different pitches.

3 Exercise 3: About reversing a list and mirroring a tree

3.1 Introduction

We revisit the `list` and `binary_tree` types and prove some four different implementations of the property where flattening a mirrored binary tree is equivalent to reversing the contents of a flattened binary tree. In each of these four versions, we prove that the property above holds for accumulator and non-accumulator implementations of binary tree flatten and list reverse and focus on reasoning about the accumulator-based functions.

3.2 Proving a property about `binary_tree_flatten_acc`

This property states that flattening a tree `t` with the accumulator being the concatenation of two lists `a1` and `a2` yields the same result as flattening the tree with just `a1` as the accumulator and appending the result to `a2`. This theorem is intuitive but let's test this statement with the usefully provided `Compute` statement.

```
Property about_binary_tree_flatten_acc :
  forall (V : Type)
         (t : binary_tree V)
         (a1 a2 : list V),
    binary_tree_flatten_acc V t (list_append V a1 a2) =
    list_append V (binary_tree_flatten_acc V t a1) a2.
Proof.
  Compute (let V := nat in
           let t := Node nat
                        (Node nat (Leaf nat 1) (Leaf nat 2))
                        (Node nat (Leaf nat 3) (Leaf nat 4)) in
           let a1 := 10 :: 20 :: nil in
           let a2 := 30 :: 40 :: nil in
           binary_tree_flatten_acc V t (list_append V a1 a2) =
           list_append V (binary_tree_flatten_acc V t a1) a2).
```

We begin the proof as usual by introducing `V` and `t` and starting an induction on the binary tree. We are reminded that `induction` is based on the type's constructors. So, for a binary tree, the first part contains the value in a leaf, and the second part includes the left and right subtrees and the induction hypotheses about them. Looking ahead, we anticipate the need for introducing the accumulators and as such, we use the semicolon and introduce them right after the induction.

The goals window is as follows:

```
2 goals (ID 125)

  V : Type
```

```
  v : V
  a1, a2 : list V
  ============================
  binary_tree_flatten_acc V (Leaf V v) (list_append V a1 a2) =
  list_append V (binary_tree_flatten_acc V (Leaf V v) a1) a2

goal 2 (ID 127) is:
 binary_tree_flatten_acc V (Node V t1 t2) (list_append V a1 a2) =
 list_append V (binary_tree_flatten_acc V (Node V t1 t2) a1) a2
```

For the first case, we simply use the fold-unfold lemma for the leaf case of `binary_tree_flatten_acc` twice followed by the fold-unfold lemma for the cons case of `list_append` and finish with `reflexivity`. Again, we use the fold-unfold lemma of `binary_tree_flatten_acc` for the node case twice. Now, we use the induction hypotheses to bring out the `list_append` call within the flattens and finish with `reflexivity`.

3.3 Analysis of List and Binary Tree Functions

`list_append` concatenates two lists by recursively adding elements from the first list to the second.

- `nil` is both left and right neutral: `list_append nil vs = vs = list_append vs nil`
- Associative: `list_append (list_append v1s v2s) v3s = list_append v1s (list_append v2s v3s)`
- Appending a singleton list to another list is equivalent to adding the element of the singleton list in front of the second list: `list_append [v] vs = v :: vs`

`list_reverse` reverses a list using a recursive, non-tail-recursive implementation.

- Reversing a nil : `list_reverse nil = nil`
- Reversing a singleton: `list_reverse [v] = [v]`
- Reversal distributes over append: `list_reverse (list_append v1s v2s) = list_append (list_reverse v2s) (list_reverse v1s)`

`list_reverse_alt` is a tail-recursive list reversal using an accumulator for efficiency.

- Uses auxiliary function `list_reverse_acc`
- Accumulator property: `list_reverse_acc [v] a = v :: a`
- Distributes over append: `list_reverse_acc (list_append v1s v2s) a = list_reverse_acc v2s (list_reverse_acc v1s a)`

`binary_tree_mirror` recursively swaps left and right subtrees, analogous to list reversal.

- Mirroring a leaf: `mirror (Leaf v) = Leaf v`

- Mirroring a node: `mirror (Node t1 t2) = Node (mirror t2)` coq(mirror t1)

`binary_tree_flatten` converts a binary tree to a list.

- Flattening a leaf: `flatten (Leaf v) = [v]`
- Flattening a node: `flatten (Node t1 t2) = list_append (flatten t1) (flatten t2)`

`binary_tree_flatten_alt` calls the auxiliary function `binary_tree_flatten_acc`, which uses an accumulator to convert a binary tree into a list. As proved in the previous section, `flatten_acc t (list_append a1 a2) = list_append (flatten_acc t a1) a2`.

3.4 about_mirroring_and_flattening_v1

Theorem v1 states that flattening a mirrored binary tree `t` yields the same result as reversing the flattened binary tree `t`. None of the functions used in version 1 use an accumulator, thus a Eureka Lemma reasoning about accumulator will not be needed in the proof. Let us begin the proof:

```
Theorem about_mirroring_and_flattening_v1 :
  forall (V : Type)
         (t : binary_tree V),
    binary_tree_flatten V (binary_tree_mirror V t) =
    list_reverse V (binary_tree_flatten V t).
Proof.
  intros V t.
  induction t as [ v | t1 IHt1 t2 IHt2].
  - rewrite -> fold_unfold_binary_tree_mirror_Leaf.
    rewrite -> fold_unfold_binary_tree_flatten_Leaf.
    rewrite -> about_applying_list_reverse_to_a_singleton_list.
    reflexivity.
  -
```

```
1 subgoal (ID 143)

  V : Type
  t1, t2 : binary_tree V
  IHt1 : binary_tree_flatten V (binary_tree_mirror V t1) =
         list_reverse V (binary_tree_flatten V t1)
  IHt2 : binary_tree_flatten V (binary_tree_mirror V t2) =
         list_reverse V (binary_tree_flatten V t2)
```

```
  ============================
  binary_tree_flatten V (binary_tree_mirror V (Node V t1 t2)) =
  list_reverse V (binary_tree_flatten V (Node V t1 t2))
```

The base case of `induction` is not a problem, as always. We know that we should use the two induction hypothesis here, but to do so, we need to `fold-unfold` what `mirror`ing and `flatten`ing a tree does:

```
    rewrite -> fold_unfold_binary_tree_mirror_Node.
    rewrite -> fold_unfold_binary_tree_flatten_Node.
    rewrite -> IHt1.
    rewrite -> IHt2.
```

```
1 subgoal (ID 151)

  V : Type
  t1, t2 : binary_tree V
  IHt1 : binary_tree_flatten V (binary_tree_mirror V t1) =
         list_reverse V (binary_tree_flatten V t1)
  IHt2 : binary_tree_flatten V (binary_tree_mirror V t2) =
         list_reverse V (binary_tree_flatten V t2)
  ============================
  list_append V (list_reverse V (binary_tree_flatten V t2))
    (list_reverse V (binary_tree_flatten V t1)) =
  list_reverse V (binary_tree_flatten V (Node V t1 t2))
```

We need to reason about list append and list reverse to push through further, and thankfully our past selves have proven a property about just that:

```
    rewrite <- (list_append_and_list_reverse_commute_with_each_other V
    ↪  (binary_tree_flatten V t1)
    (binary_tree_flatten V t2)).
    rewrite <- fold_unfold_binary_tree_flatten_Node.
    reflexivity.
Qed.
```

3.5 about_mirroring_and_flattening_v2

In the first version of the report, we tried to prove the main theorem `about_mirroring_and_flattening_v2` via induction. However, as guided mentioned

in the preliminary comments, `about_mirroring_and_flattening_v2` calls an auxiliary function that uses an accumulator, thus the theorem should be a corollary to a lemma reasoning about the auxiliary function.

Theorem v2 states that flattening a mirrored binary tree `t` yields the same result as reversing the (flattened binary tree `t`) using an accumulator. Note that `list_reverse_alt` uses an accumulator and the proof probably requires a Eureka Lemma reasoning about resetting the accumulator for `list_reverse_acc`.

Here is the auxiliary lemma that utilizes induction:

```
1  Lemma about_mirroring_and_flattening_v2_aux:
2    forall (V : Type)
3           (t : binary_tree V),
4      binary_tree_flatten V (binary_tree_mirror V t) =
5        list_reverse_acc V (binary_tree_flatten V t) nil.
6  Proof.
7    induction t as [ v | t1 IHt1 t2 IHt2].
8    - ...
9    - rewrite -> fold_unfold_binary_tree_mirror_Node.
10     rewrite -> fold_unfold_binary_tree_flatten_Node.
11     rewrite -> IHt1.
12     rewrite -> IHt2.
```

The base case and the usual rewrites are done for the inductive step. The goal now reads:

```
1  1 subgoal (ID 163)
2
3    V : Type
4    t1, t2 : binary_tree V
5    IHt1 : binary_tree_flatten V (binary_tree_mirror V t1) =
6           list_reverse_acc V (binary_tree_flatten V t1) nil
7    IHt2 : binary_tree_flatten V (binary_tree_mirror V t2) =
8           list_reverse_acc V (binary_tree_flatten V t2) nil
9    ============================
10   list_append V (list_reverse_acc V (binary_tree_flatten V t2) nil)
11     (list_reverse_acc V (binary_tree_flatten V t1) nil) =
12   list_reverse_acc V (binary_tree_flatten V (Node V t1 t2)) nil
```

No tactic can push us forward and it's looking like we need a general Eureka Lemma. We achieve this by generalizing line 12 as `a2s`; `(binary_tree_flatten V t2)` as `a1s` and `nil` as `a3s`. As trained, we test whether the hypothesis holds using a telling example:

```
1  Lemma eureka_about_mirroring_and_flattening_v2_aux :
2    forall (V : Type)
```

```
        (a1s a2s a3s : list V),
    list_append V (list_reverse_acc V a1s a3s) a2s =
      list_reverse_acc V a1s (list_append V a3s a2s).
Proof.
  Compute (let V := nat in
           let a1s := (1 :: 2 :: 3 :: nil) in
           let a2s := (4 :: 5 :: 6 :: nil) in
           let a3s := (10 :: 11 :: nil) in
           list_append V (list_reverse_acc V a1s a3s) a2s =
             list_reverse_acc V a1s (list_append V a3s a2s)).
```

and the proof itself is nothing special, just the same old `induction`, base case, apply `IH` in the induction case. Once the Eureka Lemma is proven, we can utilize it after populating the general case with our specific case, and only a few clean-up is required:

```
    rewrite -> (eureka_about_mirroring_and_flattening_v2_aux V
    ↪  (binary_tree_flatten V t2)
                (list_reverse_acc V (binary_tree_flatten V t1) nil) nil).
    rewrite -> nil_is_left_neutral_for_list_append.
    rewrite -> fold_unfold_binary_tree_flatten_Node.
    rewrite -> (list_append_and_list_reverse_acc_commute_with_each_other V
    ↪  (binary_tree_flatten V t1)
                (binary_tree_flatten V t2) nil).
    reflexivity.
```

Then, we can use the auxiliary lemma to prove the main theorem, without using induction:

```
Theorem about_mirroring_and_flattening_v2 :
  forall (V : Type)
         (t : binary_tree V),
    binary_tree_flatten V (binary_tree_mirror V t) =
      list_reverse_alt V (binary_tree_flatten V t).
Proof.
  intros V t.
  unfold list_reverse_alt.
  rewrite -> about_mirroring_and_flattening_v2_aux.
  reflexivity.
Qed.
```

3.6 about_mirroring_and_flattening_v3

Theorem v3 states that flattening (a mirrored binary tree `t`) using an accumulator yields the same result as reversing the (flattened binary tree `t`) using an accumulator. Note that two of the functions used utilize an accumulator, and our Eureka Lemmas should reason about both of them. Notice that because our main theorem calls two auxiliary functions that use accumulators, the main theorem itself should not be proved by induction, but as a corollary to an auxiliary proof reasoning about the auxiliary functions. Let us begin the auxiliary lemma:

```
Lemma about_mirroring_and_flattening_v3_aux :
  forall (V : Type) (t : binary_tree V) (acc : list V),
    binary_tree_flatten_acc V (binary_tree_mirror V t) acc =
      list_reverse_acc V (binary_tree_flatten_acc V t nil) acc.
Proof.
  intros V t.
  induction t as [v | t1' IHt1' t2' IHt2']; intro acc.
  - ...
  - rewrite -> fold_unfold_binary_tree_mirror_Node.
    rewrite ->2 fold_unfold_binary_tree_flatten_acc_Node.
    rewrite -> (IHt1' acc).
    rewrite -> (IHt2' (list_reverse_acc V (binary_tree_flatten_acc V t1' nil)
    ↪  acc)).
Qed.
```

```
1 subgoal

V : Type
t1', t2' : binary_tree V
IHt1' : forall acc : list V,
        binary_tree_flatten_acc V (binary_tree_mirror V t1') acc =
        list_reverse_acc V (binary_tree_flatten_acc V t1' nil) acc
IHt2' : forall acc : list V,
        binary_tree_flatten_acc V (binary_tree_mirror V t2') acc =
        list_reverse_acc V (binary_tree_flatten_acc V t2' nil) acc
acc : list V

========================== (1 / 1)

list_reverse_acc V (binary_tree_flatten_acc V t2' nil)
  (list_reverse_acc V (binary_tree_flatten_acc V t1' nil) acc) =
list_reverse_acc V
```

```
  (binary_tree_flatten_acc V t1' (binary_tree_flatten_acc V t2' nil)) acc
```

induction, base case, applying IH all taken care of. We know that we have a lemma about how list_append and list_reverse_acc interact with (in particular, commute with) each other. But we do not have a relationship between how list_reverse_acc interacts with binary_tree_flatten. So, if we can re-express binary_tree_flatten as a list_append, we can reason about binary_tree_flatten using commutativity of list_append and list_reverse_acc. We can do this using the following property:

```
Property about_binary_tree_flatten_acc :
  forall (V : Type)
         (t : binary_tree V)
         (a1 a2 : list V),
    binary_tree_flatten_acc V t (list_append V a1 a2) =
      list_append V (binary_tree_flatten_acc V t a1) a2.
```

To use this property, we first need to re-express our subgoal first by applying nil_is_left_neutral_for_list_append from right to left to modify the LHS. After that, we can use the property about_binary_tree_flatten_acc from left to right on the LHS and reason about the LHS using the lemma about list_append and list_reverse_acc commutating. After that, we can rearrange the expressions for (binary_tree_flatten_acc V t1' nil) and (binary_tree_flatten_acc V t2' nil)) in the LHS before rewriting and using nil_is_left_neutral_for_list_append to match the LHS with the RHS.

```
    rewrite <- (nil_is_left_neutral_for_list_append V nil) at 2.
    rewrite -> (about_binary_tree_flatten_acc V t1' nil nil).
    rewrite <- list_append_and_list_reverse_acc_commute_with_each_other.
    Check (about_binary_tree_flatten_acc).
    rewrite <-2 about_binary_tree_flatten_acc.
    rewrite -> nil_is_left_neutral_for_list_append.
    reflexivity.
Qed.
```

Then, we can prove the main theorem without using induction:

```
Theorem about_mirroring_and_flattening_v3 :
  forall (V : Type)
         (t : binary_tree V),
    binary_tree_flatten_alt V (binary_tree_mirror V t) =
      list_reverse_alt V (binary_tree_flatten_alt V t).
Proof.
  Compute (let V := nat in
           let t := Node V (Leaf V 1) (Node V (Leaf V 2) (Leaf V 3)) in
```

```
           binary_tree_flatten_alt V (binary_tree_mirror V t) =
             list_reverse_alt V (binary_tree_flatten_alt V t)).
  intros V t.
  unfold binary_tree_flatten_alt, list_reverse_alt.
  Check (about_mirroring_and_flattening_v3_aux).
  Check (about_mirroring_and_flattening_v3_aux V t).
  exact (about_mirroring_and_flattening_v3_aux V t nil).
Qed.
```

3.7 about_mirroring_and_flattening_v4

Theorem v4 states that flattening (a mirrored binary tree `t`) using an accumulator yields the same result as reversing the flattened binary tree `t`. Note that only the flattening uses an accumulator, thus our Eureka Lemma should reason about resetting the accumulator for flattening. Once again, let us create an auxiliary lemma reasoning about the auxiliary function:

```
Lemma about_mirroring_and_flattening_v4_aux :
  forall (V : Type) (t : binary_tree V) (acc : list V),
    binary_tree_flatten_acc V (binary_tree_mirror V t) acc =
      list_append V (list_reverse V (binary_tree_flatten_acc V t nil)) acc.
Proof.
  Compute ...
  intros V t.
  induction t as [v | t1' IHt1' t2' IHt2'].
  + ...
  + intro acc.
    rewrite -> fold_unfold_binary_tree_mirror_Node.
    rewrite ->2 fold_unfold_binary_tree_flatten_acc_Node.
    rewrite -> (IHt1' acc).
    rewrite -> (IHt2' (list_append V (list_reverse V (binary_tree_flatten_acc
    ↪  V t1' nil)) acc)).
Qed.
```

```
1 subgoal

V : Type
t1', t2' : binary_tree V
IHt1' : forall acc : list V,
        binary_tree_flatten_acc V (binary_tree_mirror V t1') acc =
```

```
        list_append V (list_reverse V (binary_tree_flatten_acc V t1' nil))
          acc
IHt2' : forall acc : list V,
        binary_tree_flatten_acc V (binary_tree_mirror V t2') acc =
        list_append V (list_reverse V (binary_tree_flatten_acc V t2' nil))
          acc
acc : list V

========================== (1 / 1)

list_append V (list_reverse V (binary_tree_flatten_acc V t2' nil))
  (list_append V (list_reverse V (binary_tree_flatten_acc V t1' nil)) acc) =
list_append V
  (list_reverse V
     (binary_tree_flatten_acc V t1' (binary_tree_flatten_acc V t2' nil))) acc
```

Once again, `induction`, base case and `IH` are taken care of. We know that `list_append` and `list_reverse` commute, but we don't know the relationship between `list_reverse` and how `binary_tree_flatten_acc` interacts with either of them. Or do we?
`Check about_binary_tree_flatten_acc`.

```
about_binary_tree_flatten_acc
     : forall (V : Type) (t : binary_tree V) (a1 a2 : list V),
       binary_tree_flatten_acc V t (list_append V a1 a2) =
       list_append V (binary_tree_flatten_acc V t a1) a2
```

In similar fashion to `about_mirroring_and_flattening_v3_aux`, we first re-express the subgoal using `fold_unfold_list_append_nil` so we can use the property `about_binary_tree_flatten_acc` to simplify the LHS.

```
1 subgoal

V : Type
t1', t2' : binary_tree V
IHt1' : forall acc : list V,
        binary_tree_flatten_acc V (binary_tree_mirror V t1') acc =
        list_append V (list_reverse V (binary_tree_flatten_acc V t1' nil))
          acc
IHt2' : forall acc : list V,
        binary_tree_flatten_acc V (binary_tree_mirror V t2') acc =
        list_append V (list_reverse V (binary_tree_flatten_acc V t2' nil))
          acc
acc : list V
```

```
14
15 ========================== (1 / 1)
16
17 list_append V
18   (list_reverse V (list_append V (binary_tree_flatten_acc V t2' nil) nil))
19   (list_append V (list_reverse V (binary_tree_flatten_acc V t1' nil)) acc) =
20 list_append V
21   (list_reverse V
22      (binary_tree_flatten_acc V t1' (binary_tree_flatten_acc V t2' nil))) acc
```

Checking `list_append_is_associative`, we can use this property to rearrange the subgoal and proceed with the proof using `list_append_and_list_reverse_commute_with_each_other` and clean up the LHS using `nil_is_right_neutral_for_list_append`. Finally, we can see that we can rewrite one last time using `about_binary_tree_flatten_acc` from right to left to match the LHS with the RHS.

```
1     Check (about_binary_tree_flatten_acc).
2     rewrite <- (fold_unfold_list_append_nil V nil) at 1.
3     Check (about_binary_tree_flatten_acc).
4     rewrite -> (about_binary_tree_flatten_acc V t2' nil nil).
5     Check (list_append_is_associative).
6     rewrite -> (list_append_is_associative V
7     (list_reverse V (list_append V (binary_tree_flatten_acc V t2' nil) nil))
8     (list_reverse V (binary_tree_flatten_acc V t1' nil)) acc).
9     Check (list_append_and_list_reverse_commute_with_each_other).
10    rewrite <- (list_append_and_list_reverse_commute_with_each_other V
11    (binary_tree_flatten_acc V t1' nil)
12    (list_append V (binary_tree_flatten_acc V t2' nil) nil)).
13    rewrite -> (nil_is_right_neutral_for_list_append V
      ↪  (binary_tree_flatten_acc V t2' nil)).
14    Check (about_binary_tree_flatten_acc).
15    rewrite <- (about_binary_tree_flatten_acc V t1' nil
      ↪  (binary_tree_flatten_acc V t2' nil)).
16    reflexivity.
17 Qed.
```

Then, we can use the auxiliary lemma, proved using induction, to prove the main theorem without induction.

3.8 Conclusion

Starting from `about_mirroring_and_flattening_v1`, which reasons with `binary_tree_flatten` and `list_reverse`, which do not call an auxiliary function in

their function calls and ending at `about_mirroring_and_flattening_v4`, we were able to revisit how to prove main functions that call an auxiliary tail-recursive functions. Furthermore, we reviewed how to create generalized Eureka Lemmas that reason about accumulators in this section.

4 Final Conclusion

As we conclude this project, let's reflect on our journey through the realm of arithmetic expressions, lists, and binary trees using Coq:

1. We explored the properties of `Plus` and `Times`, uncovering their conditional and universal behaviors.
2. We explored the behavior of `Plus` and `Times` for ltr and rtl evaluators.
3. We delved into list operations, proving properties about appending and reversing.
4. We examined binary trees, establishing connections between flattening, mirroring, and list operations.
5. We developed and proved several "eureka" lemmas, bridging complex concepts and simplifying our proofs.

Now that was what we did. Let us reflect on what we learned and relearned.

1. Properties that do not hold can be stated as conditional properties.
2. While our source language resembles arithmetic expressions, they are not actually arithmetic expressions and many universal arithmetic expressions do not hold due to the error propagation of our program. However, we are still able to reason about our source program by reasoning with order of evaluations for ltr and rtl evaluators.
3. The conditionals for conditional proofs sometimes require some time before the correct conditional is selected. If the goal of a proof is an impossible statement or requires an assumption to be proven, that's a great hint as to what the missing conditional is.
4. When reasoning with order of operations, it is better for us to write down the order of the operations are evaluated, then see which operations are evaluated at in the same order.
5. (relearned): main theorems reasoning with calls to recursive functions should be proved as corollaries to inductive proofs reasoning with recursive function.

   "In the beginner's mind there are many possibilities, but in the expert's mind there are few." - *Shunryu Suzuki*