

YSC4217

Mechanised Reasoning

Week 1 Report

Expanding on the program processors
for arithmetic expressions
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)

August 22, 2024

Contents

1	Initial Introduction	3
2	Exercise 1a: About Plus	4
2.1	Introduction	4
2.2	Is (<code>Literal 0</code>) neutral on the left and/or right of <code>Plus</code> ?	4
2.3	Is <code>Plus</code> Associative?	5
2.4	Is <code>Plus</code> Commutative?	6
2.5	Conclusion	10
3	Exercise 1b: About Times	11
3.1	Introduction	11
3.2	Is <code>Times</code> Associative/Commutative?	11
3.3	Does <code>Times</code> Distribute on the Right of <code>Plus</code> ?	11
3.4	Does <code>Times</code> Distribute on the Left of <code>Plus</code> ?	15
3.5	<code>Literal 1</code> as a Neutral Element for <code>Times</code>	16
3.6	<code>Literal 0</code> and Its Absorbing Property for <code>Times</code>	16
3.7	Conclusion	18
4	Exercise 3: About reversing a list and mirroring a tree	19
4.1	Introduction	19
4.2	Proving a property about <code>binary_tree_flatten_acc</code>	19
4.3	Analysis of List and Binary Tree Functions	20
4.4	<code>about_mirroring_and_flattening_v1</code>	21
4.5	<code>about_mirroring_and_flattening_v2</code>	22
4.6	<code>about_mirroring_and_flattening_v3</code>	24
4.7	<code>about_mirroring_and_flattening_v4</code>	27
4.8	Conclusion	29
5	Final Conclusion	30

1 Initial Introduction

In this assignment, we first return to our FPP finals. Throughout the journey, we revisit the observational equivalence in compiling and then running a source program and interpreting the source program; get our hands wet (once again) with T CPA; and include the Times expression as a valid expression for our program. Then, we formalize and prove fundamental properties such as associativity, commutativity, and distributivity. For properties that do not hold, we attempt to salvage them as conditional properties.

We then transition to examining the relationships between list operations (appending, reversing) and binary tree operations (flattening, mirroring). In these proofs, we revisit auxiliary functions implemented through tail recursion, auxiliary lemmas proven through induction, main functions that call auxiliary functions, and main theorems proven as a corollary to those auxiliary functions.

“The capacity to learn is a gift; The ability to learn is a skill; The willingness to learn is a choice.” - Brian Herbert

2 Exercise 1a: About Plus

“There used to be a quote about ashes, flames, rebirth and Dumbledore that tied in with the comment left in the .v file because out of the ashes of FPP the phoenix of MR is reborn, but a joke explained is no longer a joke.” - Anonymous Potterhead.

2.1 Introduction

Let us return to Plus and how it behaves when it is evaluated. We remember that by default, our programs are implemented left to right (unless one looks at the alternative rtl versions). Thus, returning for a quick look at 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`. Which means that only if the first one expression evaluates to an `Expressible_nat` is the second expression evaluated, and once again, if the second expression is evaluates to an `Expressible_msg`, that message is thrown, but if the second expression evaluates to an `Expressible_nat`, addition is performed on both numbers. We’re ready to prove some properties now.

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

If this were YSC1212: Intro to CS or YSC2209: Proof, we would have to do some tests, strengthen those tests, and finally make a hypothesis that we would have to prove. But we’re now equipped with tools to test if `Propositions` are true on the go.

```
1 Proposition Literal_0_is_neutral_for_Plus_on_the_left :
2   forall ae : arithmetic_expression,
3     evaluate (Plus (Literal 0) ae) = evaluate ae.
4 Proof.
5   intro ae.
6   rewrite -> fold_unfold_evaluate_Plus.
7   rewrite -> fold_unfold_evaluate_Literal.
8   case (evaluate ae) as [n | s].
9   - rewrite -> (Nat.add_0_l n).
10    reflexivity.
11  - reflexivity.
12 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 expressions: `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 then consider the case for `Expressible_msg s`. Seeing an equality, we clear the subcase with `reflexivity`.

Apart from searching our brain 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.

2.3 Is Plus Associative?

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

```

1 Proposition Plus_is_associative :
2   forall ae1 ae2 ae3 : arithmetic_expression,
3     evaluate (Plus ae1 (Plus ae2 ae3)) = evaluate (Plus (Plus ae1 ae2) ae3).
4 Proof.
5   intros ae1 ae2 ae3.
6   rewrite ->4 fold_unfold_evaluate_Plus.
7   case (evaluate ae1) as [n1 | s1].
8   - case (evaluate ae2) as [n2 | s2].
9     + case (evaluate ae3) as [n3 | s3].
10      * rewrite -> Nat.add_assoc.
11        reflexivity.
12      * reflexivity.
13    + 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 associativity property is crucial for reasoning about more complex arithmetic expressions. This allows us to reason with left-associative Plus operations the same way we reason with right-associative Plus operations, which could be useful in reasoning with our alternative rtl evaluators.

2.4 Is Plus Commutative?

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

```

1 Proposition Plus_is_commutative :
2   forall ae1 ae2 : arithmetic_expression,
3     evaluate (Plus ae1 ae2) = evaluate (Plus ae2 ae1).
4 Proof.
5   intros ae1 ae2.
6   rewrite ->2 fold_unfold_evaluate_Plus.
7   case (evaluate ae1) as [n1 | s1].
8   - case (evaluate ae2) as [n2 | s2].
9     + rewrite -> Nat.add_comm.
10      reflexivity.
11      + reflexivity.
12   - case (evaluate ae2) as [n2 | s2].
13      + reflexivity.
```

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

```

1 1 subgoal (ID 543)
2
3 ae1, ae2 : arithmetic_expression
4 s1, s2 : string
```

```

5 =====
6 Expressible_msg s1 = Expressible_msg s2

```

and we have no properties about either strings `s1` or `s2`, so plus is not commutative in general. We prove this in the following proposition.

```

1 Proposition Plus_is_not_commutative :
2   exists ae1 ae2: arithmetic_expression,
3     evaluate (Plus ae1 ae2) <> evaluate (Plus ae2 ae1).
4 Proof.
5   exists (Minus (Literal 1) (Literal 3)).
6   exists (Minus (Literal 2) (Literal 3)).
7   compute.
8   intro H_absurd.
9   discriminate H_absurd.
10 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**:

```

1 Proposition Plus_is_conditionally_commutative :
2   forall ae1 ae2 : arithmetic_expression,
3     ((exists n : nat,
4       evaluate ae1 = Expressible_nat n)
5     \ /
6     (exists n : nat,
7       evaluate ae2 = Expressible_nat n)
8     \ /
9     (exists s : string,
10      evaluate ae1 = Expressible_msg s /\ evaluate ae2 = Expressible_msg
11        ↪ s))
12     <->
13     evaluate (Plus ae1 ae2) = evaluate (Plus ae2 ae1).
14 Proof.
15   intros ae1 ae2.
16   split.
17   - intros [[n1 H_n1] | [[n2 H_n2] | [s [H_s1 H_s2]]]].
18     + rewrite ->2 fold_unfold_evaluate_Plus.
19     + rewrite -> H_n1.
20     + case (evaluate ae2) as [n2 | s2].
21     + * rewrite -> (Nat.add_comm n2 n1).
22     + reflexivity.

```

```

22     * reflexivity.
23 + rewrite ->2 fold_unfold_evaluate_Plus.
24     rewrite -> H_n2.
25     case (evaluate ae1) as [n1 | s1].
26     * rewrite -> (Nat.add_comm n2 n1).
27       reflexivity.
28     * reflexivity.
29 + rewrite ->2 fold_unfold_evaluate_Plus.
30     rewrite -> H_s1.
31     rewrite -> H_s2.
32     reflexivity.

```

```

1 1 subgoal (ID 530)
2
3 ae1, ae2 : arithmetic_expression
4 =====
5 evaluate (Plus ae1 ae2) = evaluate (Plus ae2 ae1) ->
6 (exists n : nat, evaluate ae1 = Expressible_nat n) \/
7 (exists n : nat, evaluate ae2 = Expressible_nat n) \/
8 (exists s : string,
9   evaluate ae1 = Expressible_msg s /\
10  evaluate ae2 = Expressible_msg s)

```

The forward direction is not difficult to prove. The backward direction requires reasoning with all values `evaluate ae1` and `evaluate ae2` evaluates to:

```

1     rewrite ->2 fold_unfold_evaluate_Plus;
2     case (evaluate ae1) as [n1 | s1] eqn:H_ae1;
3     case (evaluate ae2) as [n2 | s2] eqn:H_ae2.

```

In total, 4 subcases are generated. We pick which branch of 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 `n1`
3. Show that there exists a natural number equal to `n2`
4. Show that the message returned by the first evaluation is identical to that of the second evaluation.

Let us consider case 4 more closely:

```

1 1 subgoal (ID 667)
2
3   ae1, ae2 : arithmetic_expression
4   s1 : string
5   H_ae1 : evaluate ae1 = Expressible_msg s1
6   s2 : string
7   H_ae2 : evaluate ae2 = Expressible_msg s2
8   H_eq_s1_s2 : Expressible_msg s1 =
9                 Expressible_msg s2
10  =====
11  (exists n : nat,
12    Expressible_msg s1 = Expressible_nat n) /\
13  (exists n : nat,
14    Expressible_msg s2 = Expressible_nat n) /\
15  (exists s : string,
16    Expressible_msg s1 = Expressible_msg s /\
17    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 1 subgoal (ID 671)
2
3   ae1, ae2 : arithmetic_expression
4   s1 : string
5   H_ae1 : evaluate ae1 = Expressible_msg s1
6   s2 : string
7   H_ae2 : evaluate ae2 = Expressible_msg s2
8   H_eq_s1_s2 : Expressible_msg s1 =
9                 Expressible_msg s2
10  =====
11  exists s : string,
12    Expressible_msg s1 = Expressible_msg s /\
13    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:

```

1   exists s1.
2   split.
3   * reflexivity.

```

```
4      * exact (eq_sym H_eq_s1_s2).  
5 Qed.
```

2.5 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 salvage properties that did not hold as conditional properties. That's something we did not do back in FPP. Lastly, with some guidance, we were able to prove conditional properties not only as injections, but as bijections. 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.

3 Exercise 1b: About Times

3.1 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.

3.2 Is Times Associative/Commutative?

mutatis mutandis. The proof is (almost) identical to that of Plus.

3.3 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:

```
1 Proposition Times_distributive_over_Plus_on_the_right :
2   forall ae1 ae2 ae3 : arithmetic_expression,
3     evaluate (Times (Plus ae1 ae2) ae3) =
4     evaluate (Plus (Times ae1 ae3) (Times ae2 ae3)).
5 Proof.
6   intros ae1 ae2 ae3.
7   rewrite -> fold_unfold_evaluate_Times.
8   rewrite ->2 fold_unfold_evaluate_Plus.
9   rewrite ->2 fold_unfold_evaluate_Times.
10  case (evaluate ae1) as [n1 | s1] eqn:Hae1.
11  + case (evaluate ae2) as [n2 | s2] eqn:Hae2.
12    ++ case (evaluate ae3) as [n3 | s3] eqn:Hae3.
13      +++ rewrite -> Nat.mul_add_distr_r.
14        reflexivity.
15      +++ reflexivity.
16    ++ case (evaluate ae3) as [n3 | s3] eqn:Hae3.
17      +++ reflexivity.
18  +++ Abort. (* Expressible_msg s2 = Expressible_msg s3 *)
```

Just before aborting the proof, we encounter this subgoal:

```
1 subgoal
2
3   ae1, ae2, ae3 : arithmetic_expression
```

```

4   n1 : nat
5   Hae1 : evaluate ae1 = Expressible_nat n1
6   s2 : string
7   Hae2 : evaluate ae2 = Expressible_msg s2
8   s3 : string
9   Hae3 : evaluate ae3 = Expressible_msg s3
10  ===== (1 / 1)
11  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.

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

```

1  Proposition Times_is_not_distributive_over_Plus_on_the_right :
2    exists ae1 ae2 ae3 : arithmetic_expression,
3      evaluate (Times (Plus ae1 ae2) ae3) <>
4      evaluate (Plus (Times ae1 ae3) (Times ae2 ae3)).
5  Proof.
6    exists (Literal 0).
7    exists (Minus (Literal 0) (Literal 5)).
8    exists (Minus (Literal 2) (Literal 3)).
9    compute.
10   intro H_absurd.
11   discriminate H_absurd.
12  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 propagated. 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 distributivity.

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

```

1 Proposition Times_is_conditionally_distributive_over_Plus_on_the_right :
2   forall ae1 ae2 ae3 : arithmetic_expression,
3   forall n1 n2 n3 : nat,
4   (evaluate ae1 = Expressible_nat n1 /\
5    evaluate ae2 = Expressible_nat n2 /\
6    evaluate ae3 = Expressible_nat n3) ->
7     evaluate (Times (Plus ae1 ae2) ae3) =
8     evaluate (Plus (Times ae1 ae3) (Times ae2 ae3)).
9 Proof.
10  intros ae1 ae2 ae3 n1 n2 n3 [Hae1 [Hae2 Hae3]].
11  rewrite -> fold_unfold_evaluate_Times.
12  rewrite -> 2 fold_unfold_evaluate_Plus.
13  rewrite -> 2 fold_unfold_evaluate_Times.
14  rewrite -> Hae1, Hae2, Hae3.
15  rewrite -> Nat.mul_add_distr_r.
16  reflexivity.
17 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. In particular, we adopt this way of thinking from Prof. Danvy's email to us: 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` evaluate 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:

```

1 Proposition Times_is_conditionally_distributive_over_Plus_on_the_right :
2   forall ae1 ae2 ae3 : arithmetic_expression,
3   (exists n : nat, evaluate ae2 = Expressible_nat n)
4   \/
5   (exists n : nat, evaluate ae3 = Expressible_nat n)
6   \/
7   (exists s : string,
8    (evaluate ae2 = Expressible_msg s
9     /\
10    evaluate ae3 = Expressible_msg s) <->
11    evaluate (Times (Plus ae1 ae2) ae3) =
12    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:

```

1 Proposition Times_is_conditionally_distributive_over_Plus_on_the_right :
2   forall ae1 ae2 ae3 : arithmetic_expression,
3     (exists n : nat, evaluate ae2 = Expressible_nat n)
4   /\
5     (exists n : nat, evaluate ae3 = Expressible_nat n)
6   /\
7     (exists s : string,
8       (* either the error messages of
9        ae2 and ae3 are the same *)
10      (evaluate ae2 = Expressible_msg s
11      /\
12      evaluate ae3 = Expressible_msg s)
13    /\
14      (* or the error message is from ae1 *)
15      evaluate ae1 = Expressible_msg s) <->
16      evaluate (Times (Plus ae1 ae2) ae3) =
17      evaluate (Plus (Times ae1 ae3) (Times ae2 ae3))).

```

Or the equivalent, but more logical statement:

```

1 Proposition Times_distributes_over_Plus_on_the_right_conditionally' :
2   forall (ae1 ae2 ae3 : arithmetic_expression),
3     (exists s : string,
4       evaluate ae1 = Expressible_msg s)
5   /\
6     (exists n : nat,
7       evaluate ae2 = Expressible_nat n)
8   /\
9     (exists n : nat,
10      evaluate ae3 = Expressible_nat n)
11   /\
12     (exists s : string,
13       evaluate ae2 = Expressible_msg s
14     /\
15     evaluate ae3 = Expressible_msg s)
16   <->
17   evaluate (Times (Plus ae1 ae2) ae3) =
18   evaluate (Plus (Times ae1 ae3) (Times ae2 ae3)).
19

```

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, signalling 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.

3.4 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. Interestingly, we find that Times does indeed distribute on the left of Plus in our arithmetic expression language, without any conditions. Let's examine the proof:

```

1 Proposition Times_distributive_over_Plus_on_the_left :
2   forall ae1 ae2 ae3 : arithmetic_expression,
3     evaluate (Times ae1 (Plus ae2 ae3)) =
4     evaluate (Plus (Times ae1 ae2) (Times ae1 ae3)).
5 Proof.
6   intros ae1 ae2 ae3.
7   rewrite -> fold_unfold_evaluate_Times.
8   rewrite ->2 fold_unfold_evaluate_Plus.
9   rewrite ->2 fold_unfold_evaluate_Times.
10  case (evaluate ae1) as [n1 | s1] eqn:Hae1.
11  + case (evaluate ae2) as [n2 | s2] eqn:Hae2.
12    + case (evaluate ae3) as [n3 | s3] eqn:Hae3.
13      +++ rewrite -> Nat.mul_add_distr_l.
14        reflexivity.
15      +++ reflexivity.
16    ++ reflexivity.
17  + reflexivity.
18 Qed.

```

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.

3.5 Literal 1 as a Neutral Element for Times

mutatis mutandis. The proof and the conversation is (almost) identical to that of section 2.2.

3.6 Literal 0 and Its Absorbing Property for Times

Next, we investigate whether Literal 0 has an absorbing property for Times. Interestingly, we find that this property doesn't hold universally in our language.

```

1 Proposition Literal_0_is_absorbing_for_Times_on_the_left :
2   forall ae : arithmetic_expression,
3     evaluate (Times (Literal 0) ae) = evaluate (Literal 0).
4 Proof.
5   intro ae.
6   rewrite -> fold_unfold_evaluate_Times.
7   rewrite -> fold_unfold_evaluate_Literal.
8   case (evaluate ae) as [n | s].
9   - rewrite -> (Nat.mul_0_1 n).
10    reflexivity.
11    - (* not absorbing on the left. *)
12 Abort.
13
14 Proposition Literal_0_is_not_absorbing_for_Times_on_the_left :
15   exists ae : arithmetic_expression,
```



```

16     evaluate (Times (Literal 0) ae) <> evaluate (Literal 0).
17 Proof.
18   exists (Minus (Literal 1) (Literal 3)).
19   compute.
20   intro H_absurd.
21   discriminate H_absurd.
22 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 salvage by adding a conditional that reads:

```

1 Proposition Literal_0_is_conditionally_absorbing_for_Times_on_the_left :
2   forall (ae : arithmetic_expression),
3     (exists n, evaluate ae = Expressible_nat n)
4     <->
5     evaluate (Times (Literal 0) ae) = evaluate (Literal 0).
6 Proof.
7   intros ae.
8   split.
9   + intros [n H_ae].
10    rewrite -> fold_unfold_evaluate_Times.
11    rewrite -> fold_unfold_evaluate_Literal.
12    rewrite -> H_ae.
13    rewrite -> Nat.mul_0_1.
14    reflexivity.
15   + intros H_absorb.
16    rewrite -> fold_unfold_evaluate_Times in H_absorb.
17    rewrite -> fold_unfold_evaluate_Literal in H_absorb.
18    case (evaluate ae) as [n | s].
19    ++ exists n.
20      reflexivity.
21    ++ discriminate H_absorb.
22 Qed.

```

Note that the same condition is requires for the conditional property for literal 0 being absorbing for Times on the right. These proofs reveal some interesting properties about `Literal 0` and `Times` in our language:

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 in our language.

3.7 Conclusion

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

- Associativity: Times is universally associative, maintaining this property even in the presence of errors.
- Commutativity: Times is conditionally commutative. It holds when at least one operand evaluates to a number, but fails when both operands produce errors due to the order-dependent nature of error propagation.
- 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 all operands evaluate to numbers.
- 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.

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

4.1 Introduction

We revisit the `list` and `binary_tree` types and prove some properties relating to the connection between flattening trees and appending lists, and between mirroring trees and reversing lists.

4.2 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.

```
1 Property about_binary_tree_flatten_acc :
2   forall (V : Type)
3     (t : binary_tree V)
4     (a1 a2 : list V),
5     binary_tree_flatten_acc V t (list_append V a1 a2) =
6     list_append V (binary_tree_flatten_acc V t a1) a2.
7 Proof.
8   Compute (let V := nat in
9     let t := Node nat
10      (Node nat (Leaf nat 1) (Leaf nat 2))
11      (Node nat (Leaf nat 3) (Leaf nat 4)) in
12     let a1 := 10 :: 20 :: nil in
13     let a2 := 30 :: 40 :: nil in
14     binary_tree_flatten_acc V t (list_append V a1 a2) =
15     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:

```
1 2 goals (ID 125)
2
3   V : Type
4   v : V
5   a1, a2 : list V
```

```

6      =====
7      binary_tree_flatten_acc V (Leaf V v) (list_append V a1 a2) =
8      list_append V (binary_tree_flatten_acc V (Leaf V v) a1) a2
9
10     goal 2 (ID 127) is:
11     binary_tree_flatten_acc V (Node V t1 t2) (list_append V a1 a2) =
12     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`.

4.3 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`.

4.4 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:

```

1 Theorem about_mirroring_and_flattening_v1 :
2   forall (V : Type)
3     (t : binary_tree V),
4     binary_tree_flatten V (binary_tree_mirror V t) =
5     list_reverse V (binary_tree_flatten V t).
6 Proof.
7   intros V t.
8   induction t as [ v | t1 IHt1 t2 IHt2 ].
9   - rewrite -> fold_unfold_binary_tree_mirror_Leaf.
10    rewrite -> fold_unfold_binary_tree_flatten_Leaf.
11    rewrite -> about_applying_list_reverse_to_a_singleton_list.
12    reflexivity.
13 -
```

```

1 1 subgoal (ID 143)
2
3 V : Type
4 t1, t2 : binary_tree V
5 IHt1 : binary_tree_flatten V (binary_tree_mirror V t1) =
6       list_reverse V (binary_tree_flatten V t1)
7 IHt2 : binary_tree_flatten V (binary_tree_mirror V t2) =
8       list_reverse V (binary_tree_flatten V t2)
9 =====
10 binary_tree_flatten V (binary_tree_mirror V (Node V t1 t2)) =
11 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 mirroring and flattening a tree does:

```

1  rewrite -> fold_unfold_binary_tree_mirror_Node.
2  rewrite -> fold_unfold_binary_tree_flatten_Node.
3  rewrite -> IHt1.
4  rewrite -> IHt2.

```

```

1  subgoal (ID 151)
2
3  V : Type
4  t1, t2 : binary_tree V
5  IHt1 : binary_tree_flatten V (binary_tree_mirror V t1) =
6        list_reverse V (binary_tree_flatten V t1)
7  IHt2 : binary_tree_flatten V (binary_tree_mirror V t2) =
8        list_reverse V (binary_tree_flatten V t2)
9  =====
10 list_append V (list_reverse V (binary_tree_flatten V t2))
11   (list_reverse V (binary_tree_flatten V t1)) =
12 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:

```

1  rewrite <- (list_append_and_list_reverse_commute_with_each_other V
2    ↪ (binary_tree_flatten V t1)
3    (binary_tree_flatten V t2)).
4  rewrite <- fold_unfold_binary_tree_flatten_Node.
5  reflexivity.
6  Qed.

```

4.5 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 proof should be a corollary to an inductive proof 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)
3     (a1s a2s a3s : list V),
4     list_append V (list_reverse_acc V a1s a3s) a2s =
5     list_reverse_acc V a1s (list_append V a3s a2s).
6 Proof.

```

```

7   Compute (let V := nat in
8       let a1s := (1 :: 2 :: 3 :: nil) in
9       let a2s := (4 :: 5 :: 6 :: nil) in
10      let a3s := (10 :: 11 :: nil) in
11      list_append V (list_reverse_acc V a1s a3s) a2s =
12      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:

```

1   rewrite -> (eureka_about_mirroring_and_flattening_v2_aux V
2       ↪ (binary_tree_flatten V t2)
3       (list_reverse_acc V (binary_tree_flatten V t1) nil) nil).
4   rewrite -> nil_is_left_neutral_for_list_append.
5   rewrite -> fold_unfold_binary_tree_flatten_Node.
6   rewrite -> (list_append_and_list_reverse_acc_commute_with_each_other V
7       ↪ (binary_tree_flatten V t1)
8       (binary_tree_flatten V t2) nil).
9   reflexivity.

```

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

```

1   Theorem about_mirroring_and_flattening_v2 :
2   forall (V : Type)
3       (t : binary_tree V),
4       binary_tree_flatten V (binary_tree_mirror V t) =
5       list_reverse_alt V (binary_tree_flatten V t).
6   Proof.
7       intros V t.
8       unfold list_reverse_alt.
9       rewrite -> about_mirroring_and_flattening_v2_aux.
10      reflexivity.
11   Qed.

```

4.6 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:

```

1 Lemma about_mirroring_and_flattening_v3_aux :
2   forall (V : Type) (t : binary_tree V) (acc : list V),
3     binary_tree_flatten_acc V (binary_tree_mirror V t) acc =
4     list_reverse_acc V (binary_tree_flatten_acc V t nil) acc.
5 Proof.
6   intros V t.
7   induction t as [v | t1' IHt1' t2' IHt2']; intro acc.
8   - ...
9   - rewrite -> fold_unfold_binary_tree_mirror_Node.
10    rewrite ->2 fold_unfold_binary_tree_flatten_acc_Node.
11    rewrite -> (IHt1' acc).
12    rewrite -> (IHt2' (list_reverse_acc V (binary_tree_flatten_acc V t1' nil)
13      ↪ acc)).
14 Qed.

```

```

1 subgoal
2
3 V : Type
4 t1', t2' : binary_tree V
5 IHt1' : forall acc : list V,
6   binary_tree_flatten_acc V (binary_tree_mirror V t1') acc =
7   list_reverse_acc V (binary_tree_flatten_acc V t1' nil) acc
8 IHt2' : forall acc : list V,
9   binary_tree_flatten_acc V (binary_tree_mirror V t2') acc =
10  list_reverse_acc V (binary_tree_flatten_acc V t2' nil) acc
11 acc : list V
12
13 ===== (1 / 1)
14
15 list_reverse_acc V (binary_tree_flatten_acc V t2' nil)
16   (list_reverse_acc V (binary_tree_flatten_acc V t1' nil) acc) =
17 list_reverse_acc V
18   (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:

```

1 Property about_binary_tree_flatten_acc :
2   forall (V : Type)
3     (t : binary_tree V)
4     (a1 a2 : list V),
5     binary_tree_flatten_acc V t (list_append V a1 a2) =
6     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` commuting. 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.

```

1 rewrite <- (nil_is_left_neutral_for_list_append V nil) at 2.
2 rewrite -> (about_binary_tree_flatten_acc V t1' nil nil).
3 rewrite <- list_append_and_list_reverse_acc_commute_with_each_other.
4 Check (about_binary_tree_flatten_acc).
5 rewrite <-2 about_binary_tree_flatten_acc.
6 rewrite -> nil_is_left_neutral_for_list_append.
7 reflexivity.
8 Qed.

```

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

```

1 Theorem about_mirroring_and_flattening_v3 :
2   forall (V : Type)
3     (t : binary_tree V),
4     binary_tree_flatten_alt V (binary_tree_mirror V t) =
5     list_reverse_alt V (binary_tree_flatten_alt V t).
6 Proof.
7   Compute (let V := nat in
8     let t := Node V (Leaf V 1) (Node V (Leaf V 2) (Leaf V 3)) in
9     binary_tree_flatten_alt V (binary_tree_mirror V t) =
10    list_reverse_alt V (binary_tree_flatten_alt V t)).
11 intros V t.
12 unfold binary_tree_flatten_alt, list_reverse_alt.
13 Check (about_mirroring_and_flattening_v3_aux).
14 Check (about_mirroring_and_flattening_v3_aux V t).

```

```

15   exact (about_mirroring_and_flattening_v3_aux V t nil).
16   Qed.

```

4.7 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:

```

1  Lemma about_mirroring_and_flattening_v4_aux :
2    forall (V : Type) (t : binary_tree V) (acc : list V),
3      binary_tree_flatten_acc V (binary_tree_mirror V t) acc =
4      list_append V (list_reverse V (binary_tree_flatten_acc V t nil)) acc.
5  Proof.
6    Compute ...
7    intros V t.
8    induction t as [v | t1' IHt1' t2' IHt2'].
9    + ...
10   + intro acc.
11     rewrite -> fold_unfold_binary_tree_mirror_Node.
12     rewrite ->2 fold_unfold_binary_tree_flatten_acc_Node.
13     rewrite -> (IHt1' acc).
14     rewrite -> (IHt2' (list_append V (list_reverse V (binary_tree_flatten_acc
15       ↪ V t1' nil)) acc)).
16   Qed.

```

```

1  1 subgoal
2
3  V : Type
4  t1', t2' : binary_tree V
5  IHt1' : forall acc : list V,
6    binary_tree_flatten_acc V (binary_tree_mirror V t1') acc =
7    list_append V (list_reverse V (binary_tree_flatten_acc V t1' nil))
8    acc
9  IHt2' : forall acc : list V,
10    binary_tree_flatten_acc V (binary_tree_mirror V t2') acc =
11    list_append V (list_reverse V (binary_tree_flatten_acc V t2' nil))
12    acc

```

```

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

```

Once again, **induction**, base case and IH 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`.

```

1 about_binary_tree_flatten_acc
2   : forall (V : Type) (t : binary_tree V) (a1 a2 : list V),
3     binary_tree_flatten_acc V t (list_append V a1 a2) =
4     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 1 subgoal
2
3 V : Type
4 t1', t2' : binary_tree V
5 IHt1' : forall acc : list V,
6   binary_tree_flatten_acc V (binary_tree_mirror V t1') acc =
7   list_append V (list_reverse V (binary_tree_flatten_acc V t1' nil))
8   acc
9 IHt2' : forall acc : list V,
10  binary_tree_flatten_acc V (binary_tree_mirror V t2') acc =
11  list_append V (list_reverse V (binary_tree_flatten_acc V t2' nil))
12  acc
13 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
14   ⇨ (binary_tree_flatten_acc V t2' nil)).
15 Check (about_binary_tree_flatten_acc).
16 rewrite <- (about_binary_tree_flatten_acc V t1' nil
17   ⇨ (binary_tree_flatten_acc V t2' nil)).
18 reflexivity.
19 Qed.

```

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

4.8 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.

5 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 delved into list operations, proving properties about appending and reversing.
3. We examined binary trees, establishing connections between flattening, mirroring, and list operations.
4. 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. Proofs that do not hold can be salvaged as conditional proofs
2. 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.
3. 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.
4. (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