YSC4217

Mechanised Reasoning

Week 1 Report

Expanding on the language processors for arithmetic expressions and revisiting lists and binary trees

ME LOOKING AT TCPA AFTER 8 MONTHS
Did The Past Really Happen?
10M views
239K
4.3K
Share
Download
Save
imgflip.com

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 20, 2024

Contents

1 Initial Introduction

In this assignment, we delve into the intricacies of arithmetic expressions, lists, and binary trees using the Coq proof assistant. Our journey begins with revisiting the Plus operation and adding the Times operation in the custom arithmetic expression language that we did as our FPP term project. We investigate their fundamental properties such as associativity, commutativity, and distributivity. We then transition to examining the relationships between list operations (appending, reversing) and binary tree operations (flattening, mirroring).

> "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

“Fawkes is a phoenix, Harry. Phoenixes burst into flame when it is time for them to die and are reborn from the ashes. Watch him...” - Dumbledore.

2.1 Introduction

Let us return to Plus and how it behaves when it is `evaluated`. Taking a quick look at our `specification_of_evaluate`, if evaluating the first `arithmetic_expression` returns an `Expressible_msg`, the same message is thrown, without evaluating the second `arithmetic_expression`. Only if the first one expression evaluates to an `Expressible_nat` is the second expression evaluated, and once again, if the second expression is `evalutes` 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?

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 `Proposition`s are true on the go.

```
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.
```

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`.

2.3 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.
  - reflexivity.
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. It allows us to regroup Plus operations without changing the overall meaning or result of the expression, providing flexibility in how we construct and evaluate arithmetic expressions in our language.

2.4 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. In particular, `Plus` is not commutative if the error message from the first `evaluate` is different from the latter `evaluate`. We can capture this property in 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. In particular, if one of the `aes evaluates` to an `Expressible_nat`, it ensures that evaluating the sum is either an `Expressible_nat` or the `Expressible_msg` thrown by the other `evaluate`:

```
Proposition Plus_is_conditionally_commutative:
  forall ae1 ae2 : arithmetic_expression,
  forall n1 n2 : nat,
  (evaluate ae1 = Expressible_nat n1 \/ evaluate ae2 = Expressible_nat n2) ->
  evaluate (Plus ae1 ae2) = evaluate (Plus ae2 ae1).
Proof.
  intros ae1 ae2 n1 n2 [H_ae1 | H_ae2]; rewrite ->2
   ↪  fold_unfold_evaluate_Plus.
  - destruct (evaluate ae2) as [m1 | s2]; rewrite -> H_ae1.
    + rewrite -> Nat.add_comm.
      reflexivity.
    + reflexivity.
  - destruct (evaluate ae1) as [m1 | s1]; rewrite -> H_ae2.
    + rewrite -> Nat.add_comm.
      reflexivity.
    + reflexivity.
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.

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 want to investigate the fundamental properties of Times, particularly its associativity.

3.2 Is Times Associative?

To determine whether Times is associative, we formulated and proved the following proposition:

```
Proposition Times_is_associative :
  forall ae1 ae2 ae3 : arithmetic_expression,
    evaluate (Times ae1 (Times ae2 ae3)) = evaluate (Times (Times ae1 ae2)
    ↪  ae3).
Proof.
  intros ae1 ae2 ae3.
  rewrite ->4 fold_unfold_evaluate_Times.
  case (evaluate ae1) as [n1 | s1].
  - case (evaluate ae2) as [n2 | s2].
    + case (evaluate ae3) as [n3 | s3].
      * rewrite -> Nat.mul_assoc.
        reflexivity.
      * reflexivity.
    + reflexivity.
  - reflexivity.
Qed.
```

This proof demonstrates that Times is indeed associative for our arithmetic expressions. The structure of this proof is remarkably similar to the one for Plus associativity, which highlights the consistency in our language design.

This proof reveals important properties of our Times operation:

1. Associativity holds when all sub-expressions evaluate to numbers, utilizing the associativity of natural number multiplication.

2. The short-circuiting behavior of Times 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.

These properties are the same as that of Plus.

3.3 Is Times Commutative?

After examining the associativity of Times, we naturally turn to another fundamental property: commutativity. Our initial attempt to prove that Times is commutative for all arithmetic expressions led to an interesting discovery:

```
Proposition Times_is_commutative :
  forall ae1 ae2 : arithmetic_expression,
    evaluate (Times ae1 ae2) = evaluate (Times ae2 ae1).
Proof.
  intros ae1 ae2.
  rewrite -> 2 fold_unfold_evaluate_Times.
  case (evaluate ae1) as [n1 | s1].
  + case (evaluate ae2) as [n2 | s2].
    ++ rewrite -> Nat.mul_comm.
       reflexivity.
    ++ reflexivity.
  + case (evaluate ae2) as [n2 | s2].
    ++ reflexivity.
    ++ (* Times is not commutative *)
Abort.
```

Just before we aborted the proof, we encountered this subgoal:

```
1 subgoal
ae1, ae2 : arithmetic_expression
s1, s2 : string
========================= (1 / 1)
Expressible_msg s1 = Expressible_msg s2
```

This subgoal reveals that Times is not universally commutative in our language. The issue arises when both expressions evaluate to error messages. In this case, the order matters because we return the first encountered error message. We can observe this with the following example:

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

= Expressible_msg "numerical underflow: -2" =
  Expressible_msg "numerical underflow: -1"
  : Prop
```

To formally establish that Times is not commutative, we proved the following proposition:

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

This proof provides a concrete counterexample where Times is not commutative. Both expressions result in error messages, but these messages differ based on the order of evaluation.

However, Times does exhibit a form of conditional commutativity. We can prove that Times is commutative when at least one of the expressions evaluates to a number:

```
Proposition Times_is_conditionally_commutative  :
  forall ae1 ae2 : arithmetic_expression,
  forall n1 n2 : nat,
  (evaluate ae1 = Expressible_nat n1 \/ evaluate ae2 = Expressible_nat n2) ->
  evaluate (Times ae1 ae2) = evaluate (Times ae2 ae1).
Proof.
intros ae1 ae2 n1 n2 [H_ae1 | H_ae2].
+ rewrite -> 2 fold_unfold_evaluate_Times.
  destruct (evaluate ae2) as [m2 | s2].
  ++ rewrite -> H_ae1.
     rewrite -> Nat.mul_comm.
     reflexivity.
  ++ rewrite -> H_ae1.
     reflexivity.
+ rewrite -> 2 fold_unfold_evaluate_Times.
  destruct (evaluate ae1) as [m1 | s1].
  ++ rewrite -> H_ae2.
     rewrite -> Nat.mul_comm.
     reflexivity.
  ++ rewrite -> H_ae2.
     reflexivity.
Qed.
```

This proof demonstrates that Times is commutative under the condition that at least one of the expressions evaluates to a number. The proof strategy involves considering two main cases: When `ae1` evaluates to a number, or when `ae2` evaluates to a number.

In both cases, we show that the commutativity holds, either by applying the commutativity of natural number multiplication or by the short-circuiting behavior when an error occurs.

These results reveal some nuanced properties of our Times operation:

1. Times is not universally commutative due to its behavior with error messages.
2. The order of operands matters when both may produce errors, as it affects which error message is propagated.
3. With this in mind, Times is commutative when at least one operand evaluates to a number as then there will be no ambiguity about which expression the error message comes from.

This conditional commutativity of Times adds an interesting layer of complexity to our language. It highlights the interplay between the mathematical properties we expect from arithmetic operations and the practical considerations of error handling in our language design.

3.4 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.

To 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 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:

```
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.
```

This proof demonstrates that Times is distributive over Plus on the right under the condition that all expressions evaluate to numbers. The proof strategy involves: 1. Assuming all three expressions evaluate to numbers. 2. Unfolding the definitions of `evaluate` for Times and Plus. 3. Applying the distributivity of multiplication over addition for natural numbers.

These results reveal some nuanced properties of the interaction between Times and Plus in our language:

1. Times is not universally distributive over Plus on the right due to its behavior with error messages.

2. Times is distributive over Plus on the right when all operands evaluate to numbers.

3. The order and structure of expressions matter when errors may occur, as it affects which error message is propagated.

The similarities between these results and those for commutativity of Times underscore a consistent pattern in our language: many expected algebraic properties hold conditionally when we're dealing with numbers, but break down in the presence of errors. This pattern provides valuable insights for users of our language, guiding them in constructing robust and predictable arithmetic expressions.

3.5 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:

```
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)).
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_l.
           reflexivity.
       +++ reflexivity.
    ++ reflexivity.
  + reflexivity.
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.

3.6 Literal 1 as a Neutral Element for Times

In this section, we explore some additional properties of the Times operation, particularly its interaction with Literal 0 and Literal 1. These properties provide further insight into the behavior of our arithmetic expressions.

We first examine whether Literal 1 acts as a neutral element for Times, both on the left and right sides.

```
Proposition Literal_1_is_neutral_for_Times_on_the_left :
  forall ae : arithmetic_expression,
    evaluate (Times (Literal 1) ae) = evaluate ae.
Proof.
  intro ae.
  rewrite -> fold_unfold_evaluate_Times.
  rewrite -> fold_unfold_evaluate_Literal.
  case (evaluate ae) as [n | s].
  - rewrite -> (Nat.mul_1_l n).
    reflexivity.
  - reflexivity.
Qed.

Proposition Literal_1_is_neutral_for_Times_on_the_right :
  forall ae : arithmetic_expression,
    evaluate (Times ae (Literal 1)) = evaluate ae.
Proof.
  intro ae.
  rewrite -> fold_unfold_evaluate_Times.
  rewrite -> fold_unfold_evaluate_Literal.
  case (evaluate ae) as [n | s].
  - rewrite -> (Nat.mul_1_r n).
    reflexivity.
  - reflexivity.
Qed.
```

These proofs demonstrate that Literal 1 is indeed a neutral element for Times, both when multiplied on the left and on the right. This property holds universally, even in the presence of error messages, due to our language's evaluation strategy.

3.7 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.

3.7.1 Literal 0 on the Left

```
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.

Proposition Literal_0_is_conditionally_absorbing_for_Times_on_the_left :
  forall ae : arithmetic_expression,
  forall n : nat,
  (evaluate ae = Expressible_nat n) ->
    evaluate (Times (Literal 0) ae) = evaluate (Literal 0).
Proof.
  intros ae n H_ae.
  rewrite -> fold_unfold_evaluate_Times.
  rewrite -> fold_unfold_evaluate_Literal.
  rewrite -> H_ae.
  rewrite -> Nat.mul_0_l.
  reflexivity.
Qed.
```

3.7.2 Literal 0 on the Right

```
Proposition Literal_0_is_absorbing_for_Times_on_the_right :
  forall ae : arithmetic_expression,
    evaluate (Times ae (Literal 0)) = 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_r n).
    reflexivity.
  - (* not absorbing on the right. *)
Abort.

Proposition Literal_0_is_not_absorbing_for_Times_on_the_right :
  exists ae : arithmetic_expression,
    evaluate (Times ae (Literal 0)) <> evaluate (Literal 0).
Proof.
  exists (Minus (Literal 1) (Literal 3)).
  compute.
  intro H_absurd.
  discriminate H_absurd.
Qed.

Proposition Literal_0_is_conditionally_absorbing_for_Times_on_the_right :
  forall ae : arithmetic_expression,
  forall n : nat,
  (evaluate ae = Expressible_nat n) ->
    evaluate (Times ae (Literal 0)) = evaluate (Literal 0).
Proof.
  intros ae n H_ae.
  rewrite -> fold_unfold_evaluate_Times.
  rewrite -> fold_unfold_evaluate_Literal.
  rewrite -> H_ae.
  rewrite -> Nat.mul_0_r.
  reflexivity.
Qed.
```

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.

3.8 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 interesting 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` is the same 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`.

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: `append nil vs = vs = append vs nil`
- Associative: `append (append v1s v2s) v3s = append v1s (append v2s v3s)`
- Appending a singleton is cons: `append [v] vs = v :: vs`

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

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

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

- Uses auxiliary function `list_reverse_acc`
- Accumulator property: `reverse_acc [v] a = v :: a`
- Distributes over append: `reverse_acc (append v1s v2s) a = reverse_acc v2s (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 via in-order traversal.

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

`binary_tree_flatten_alt` is a tail-recursive tree flattening using an accumulator. It uses auxiliary function `binary_tree_flatten_acc`. As proved in the previous section, `flatten_acc t (append a1 a2) = append (flatten_acc t a1) a2`.

4.4 about_mirroring_and_flattening_v1

Theorem v1 states that flattening a mirrored binary tree `t` is the same 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.
```

4.5 about_mirroring_and_flattening_v2

Theorem v2 states that flattening a mirrored binary tree `t` is the same 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`.

```
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.
  induction t as [ v | t1 IHt1 t2 IHt2].
  - ...
  - rewrite -> fold_unfold_binary_tree_mirror_Node.
    rewrite ->2 fold_unfold_binary_tree_flatten_Node.
    rewrite -> IHt1.
    rewrite -> IHt2.
    rewrite -> list_append_and_list_reverse_acc_commute_with_each_other.
```

```
1
2  1 subgoal (ID 200)
3
4    V : Type
5    t1, t2 : binary_tree V
6    IHt1 : binary_tree_flatten V (binary_tree_mirror V t1) =
7           list_reverse_acc V (binary_tree_flatten V t1) nil
8    IHt2 : binary_tree_flatten V (binary_tree_mirror V t2) =
9           list_reverse_acc V (binary_tree_flatten V t2) nil
10   ============================
11   list_append V (list_reverse_acc V (binary_tree_flatten V t2) nil)
12     (list_reverse_acc V (binary_tree_flatten V t1) nil) =
13   list_reverse_acc V (binary_tree_flatten V t2)
14     (list_reverse_acc V (binary_tree_flatten V t1) nil)
```

The base case and applying the induction hypothesis after `fold_unfold` is taken care of. We see that no theorems or tactics can push us any further and begin formulating a Eureka Lemma. We achieve this by generalizing line 12 as `a2s`; `(binary_tree_flatten V t2)` as `a1s` and `nil` as `prefix`. As trained, we test whether the hypothesis holds using a telling example:

```
Lemma about_mirroring_and_flattening_v2_aux :
  forall (V : Type)
         (a1s a2s prefix : list V),
    list_append V (list_reverse_acc V a1s prefix) a2s =
      list_reverse_acc V a1s (list_append V prefix a2s).
Proof.
```

```
Compute (let V := nat in
         let a1s := (1 :: 2 :: 3 :: nil) in
         let a2s := (4 :: 5 :: 6 :: nil) in
         let prefix := (10 :: 11 :: nil) in
         list_append V (list_reverse_acc V a1s prefix) a2s =
           list_reverse_acc V a1s (list_append V prefix 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 -> (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.
    reflexivity.
```

4.6 about_mirroring_and_flattening_v3

Theorem v3 states that flattening (a mirrored binary tree `t`) using an accumulator is the same 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. **We notice that the structure of v2 (which also calls an auxiliary function) does not follow this pattern, but the time was too pressing to make changes to the .v file and rewrite a report about it.** 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)).
```

```
1 subgoal (ID 228)

  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
  ============================
  list_reverse_acc V (binary_tree_flatten_acc V t2' nil)s
    (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`:

```
Lemma eureka_binary_tree_flatten_acc_append :
  forall (V : Type) (t : binary_tree V) (acc : list V),
    binary_tree_flatten_acc V t acc =
    list_append V (binary_tree_flatten_acc V t nil) acc.
Proof.
  intros V t acc.
  rewrite <- (about_binary_tree_flatten_acc V t nil acc).
  rewrite -> nil_is_left_neutral_for_list_append.
  reflexivity.
Qed.
```

Once proven, we can apply the eureka lemma to the left hand side and right hand side of the relation (verbose to show which side we're applying the eureka lemma to) and we can reasong about the relationship using the lemma about `list_append` and `list_reverse_acc` commutating:

```
    rewrite -> (eureka_binary_tree_flatten_acc_append V t2' nil).
    rewrite -> (eureka_binary_tree_flatten_acc_append V t1' (list_append V
    ↪  (binary_tree_flatten_acc V t2' nil) nil)).
    Check list_append_and_list_reverse_acc_commute_with_each_other.
    rewrite -> (list_append_and_list_reverse_acc_commute_with_each_other V
    ↪  (binary_tree_flatten_acc V t1' nil)
    (list_append V (binary_tree_flatten_acc V t2' nil) nil) acc).
    reflexivity.
```

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.
  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.
```

4.7 about_mirroring_and_flattening_v4

Theorem v4 states that flattening (a mirrored binary tree `t`) using an accumulator is the same 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 auxilary lemma reasoning about the auxilary 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)).
```

```
1 subgoal (ID 196)

  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
  ============================
  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` 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 eureka_binary_tree_flatten_acc_append.
```

```
eureka_binary_tree_flatten_acc_append
     : forall (V : Type) (t : binary_tree V) (acc : list V),
       binary_tree_flatten_acc V t acc =
       list_append V (binary_tree_flatten_acc V t nil) acc
```

We actually have already proven such a lemma. If we apply this to the subgoal, we will be able to use the commutativity between `list_append` and `list_reverse` to prove the lemma:

```
    rewrite -> (eureka_binary_tree_flatten_acc_append V t2' nil).
    rewrite -> (eureka_binary_tree_flatten_acc_append V t1'
    (list_append V (binary_tree_flatten_acc V t2' nil) nil)).
    rewrite -> (list_append_and_list_reverse_commute_with_each_other V
    ↪  (binary_tree_flatten_acc V t1' nil)
    (list_append V (binary_tree_flatten_acc V t2' nil) nil)).
    rewrite <- (list_append_is_associative V (list_reverse V (list_append V
    ↪  (binary_tree_flatten_acc V t2' nil) nil))
    (list_reverse V (binary_tree_flatten_acc V t1' nil)) acc).
     reflexivity.
Qed.
```

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

4.8 Conclusion

This exercise explored the relationship between list operations (appending, reversing) and binary tree operations (flattening, mirroring). We progressively built our understanding through four theorems, each relying on key "eureka" lemmas:

Version 1 established the basic relationship between flattening, mirroring, and reversing without accumulators, utilizing the crucial property `list_append_and_list_reverse_commute_with_each_other`. This lemma deepened our understanding of how list operations interact, showing that reversing distributes over appending with a reversal of order.

Version 2 introduced accumulator-based list reversal, requiring our first eureka lemma `about_mirroring_and_flattening_v2_aux`. This lemma revealed how accumulator-based reversal interacts with list appending, a key insight for handling accumulator resets.

Version 3 expanded to accumulator-based tree flattening, necessitating a more complex auxiliary lemma and the crucial eureka lemma `eureka_binary_tree_flatten_acc_append`. This lemma exposed the relationship between accumulator-based flattening and list appending, bridging tree, and list operations.

Version 4 combined accumulator-based flattening with non-accumulator list reversal, reusing `eureka_binary_tree_flatten_acc_append` and leveraging `list_append_and_list_reverse_commute_with_each_other` and `list_append_is_associative`.

These eureka lemmas were pivotal in our proofs, each offering deeper insights. They revealed how accumulator-based functions relate to their non-accumulator counterparts. They showed how list operations (append, reverse) interact with tree operations (flatten, mirror). They demonstrated that seemingly complex operations can often be expressed in terms of simpler, known operations. The use of `list_append_is_associative` in v4 highlighted how fundamental list properties remain relevant even in complex scenarios involving trees.

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.

This project served as a refresher course in Type-Checked Program Analysis, reminding us of the power and intricacies of formal verification. We rediscovered the importance of induction, case analysis, and auxiliary lemmas in constructing robust proofs. The process of revisiting these concepts highlighted both the challenges and the satisfaction of formal reasoning.

Throughout this report, we saw the power of building upon previous proofs and lemmas. Each theorem increased in complexity, but our growing library of proven properties made tackling these more intricate relationships manageable. The eureka lemmas, in particular, served as bridges between different concepts, allowing us to transform complex goals into more manageable forms.

As we return to TCPA, we find ourselves with both the enthusiasm of beginners and the insights of our past experiences. This combination has allowed us to approach familiar concepts with fresh perspectives, deepening our understanding and appreciation of formal methods in computer science.

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