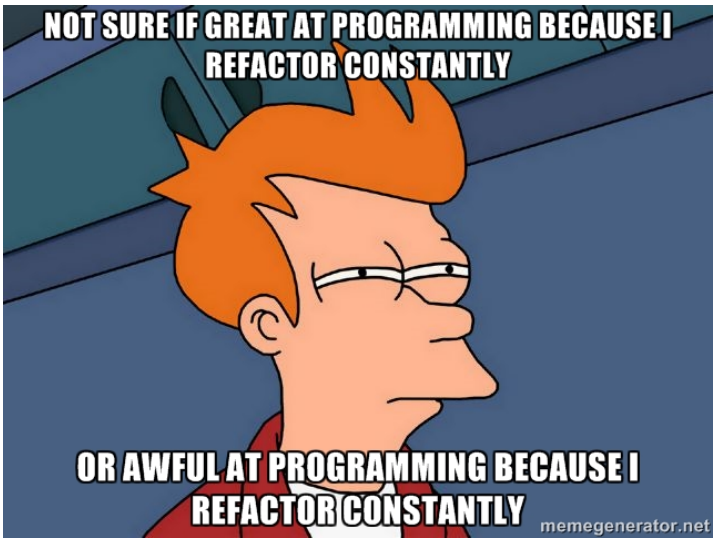
YSC4217

Mechanised Reasoning

Week 2 Report

On refactoring the language for arithmetic expressions with preserved evaluation and idempotence



Alan Matthew Anggara (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 27, 2024

Contents

1 Initial Introduction

In this week's assignment, we delve further into the realm of program processors for arithmetic expressions by exploring two refactoring functions. Although these functions modify the structure of arithmetic expressions, the changes do not alter the output of evaluation. We examined the behavior of these refactor functions and proved that they not only preserve the correctness of evaluation but are also idempotent.

> "If you can get today's work done today, but you do it in such a way that you can't possibly get tomorrow's work done tomorrow, then you lose." - *Martin Fowler*

2 Exercise 1: Refactor

2.1 Introduction

In this section, we discuss `refactor`, which takes in an arithmetic expression and returns an arithmetic expression. The function calls an auxiliary function using an accumulator. We are interested in whether the resulting arithmetic expression evaluates to the same result the original arithmetic expression evaluates to. During the proof, we can see the right-associating nature of the refactoring.

2.2 What refactor does

Let us divide the operators one by one and observe what `refactor` does.

- Literal n: A binary tree where Plus is the root and the left child is the original (Literal n) and the right child is a new (Literal 0) is created.
- Plus ae1 ae2: A right-skewed binary tree where Plus is the root, left child is ae1 and the right child is a binary tree holding ae2 on the LHS and (Literal 0) on the RHS is created.
- Minus ae1 ae2: First, a balanced binary tree where Minus is the root, and ae1 on the LHS, ae2 on the RHS is created. Then, another binary tree is created where the LHS is the above binary tree and the RHS is (Literal 0). The later binary tree is returned.

Feeding nested Pluses into the refactor and drawing the tree out yields a surprising result. Regardless of the shape of the original binary tree of Pluses, the resulting binary tree is a flattened version where the LHS of the tree are the values, with a (Literal 0) appended to the tail of the list as the nil case. In other words, refactoring is applying right-associativity to a sequence of Plus operations.

As for Minus, the flattening does not occur, because subtraction is not associative. Using the language of binary trees and lists, Minus is similar to appending the trees flattened by Plus. During the appending, a (Literal 0) is added as the right sibling joined by a Plus as a parent.

One final observation is that this means that refactoring always ensures that a Plus is at the node of the refactored tree.

2.3 Proving that refactor preserves evaluation

Refactor calls a recursively defined auxiliary function. Thus, it is fitting for us to have an inductively proved auxiliary proposition, namely

`refactoring_preserves_evaluation_aux`. The eureka lemma generalizes the instantiated accumulator, (Literal 0), when refactoring is unfolded.

```
Lemma refactoring_preserves_evaluation_aux :
  forall (ae a : arithmetic_expression),
    evaluate (refactor_aux ae a) = evaluate (Plus ae a).
Proof.
```

After introducing ae, let us induct on it, then introduce a, in case we might need the Light of Inductil down the line. Literal n case is not a problem for us:

```
  intro ae.
  induction ae as [ n | ae1 IHae1 ae2 IHae2 | ae1 IHae1 ae2 IHae2
  ↪  ];
    intro a.
  - rewrite -> fold_unfold_refactor_aux_Literal.
    reflexivity.
  -
```

```
  ae1, ae2 : arithmetic_expression
  IHae1 : forall a : arithmetic_expression,
          evaluate (refactor_aux ae1 a) = evaluate (Plus ae1 a)
  IHae2 : forall a : arithmetic_expression,
          evaluate (refactor_aux ae2 a) = evaluate (Plus ae2 a)
  a : arithmetic_expression
  ============================
  evaluate (refactor_aux (Plus ae1 ae2) a) = evaluate (Plus (Plus
  ↪  ae1 ae2) a)
```

It's a good thing we delayed the introduction of a as long as possible, because we can instantiate the accumulator in the inductive hypothesis. Rewriting with IHae1, IHae2 and a few fold-unfolds reveals the underlying equality:

```
    rewrite -> fold_unfold_refactor_aux_Plus.
    rewrite -> (IHae1 (refactor_aux ae2 a)).
```

```
    rewrite -> fold_unfold_evaluate_Plus.
    rewrite -> (IHae2 a).
```

```
...
  ============================
  match evaluate ae1 with
  | Expressible_nat n1 =>
      match evaluate (Plus ae2 a) with
      | Expressible_nat n2 => Expressible_nat (n1 + n2)
      | Expressible_msg s2 => Expressible_msg s2
      end
  | Expressible_msg s1 => Expressible_msg s1
  end = evaluate (Plus (Plus ae1 ae2) a)
```

Because we have no assumptions about evaluating ae1, we must reason about it by cases. Focusing on the nat case for ae1 and pushing forth with fold-unfolds:

```
    case (evaluate ae1) as [n1 | s1] eqn:E_ae1.
    + rewrite ->3 fold_unfold_evaluate_Plus.
      rewrite -> E_ae1.
```

```
...
  ============================
  match
    match evaluate ae2 with
    | Expressible_nat n0 => ...
    | Expressible_msg s1 => ...
```

Once again, we must reason about our proof via cases. Focusing on the nat case for ae2 now has a match statement for evaluate a. Twice again, we must reason about our poof via cases. Focusing on the nat case for a:

```
      case (evaluate ae2) as [n2 | s2] eqn:E_ae2.
      * case (evaluate a) as [n | s] eqn:E_a.
        --
```

```
  ============================
  Expressible_nat (n1 + (n2 + n)) = Expressible_nat (n1 + n2 + n)
```

This is the aha! moment in the proof. Remember how in section 2.2, we discussed how refactoring is applying right-associativity to chained plus operations? Now we see the right-association in action in the goal because we're in the case where ae1, ae2, and the accumulator a all evaluate to a natural number. This is the interesting case in the proof and all other sub-branches for Plus can be proved via reflexivity.

```
          rewrite -> Nat.add_assoc.
          reflexivity.
        -- reflexivity.
      * reflexivity.
    + rewrite ->2 fold_unfold_evaluate_Plus.
      rewrite -> E_ae1.
      reflexivity.
```

Now let us move the discussion onto Minuses. Because the goals do get messy here, let us only discuss how we get to the mess, for now. As always, we unfold as much as possible and apply the inductive hypotheses:

```
  - rewrite -> fold_unfold_refactor_aux_Minus.
    rewrite ->2 fold_unfold_evaluate_Plus.
    rewrite ->2 fold_unfold_evaluate_Minus.
    rewrite -> IHae1, IHae2.
    rewrite ->2 fold_unfold_evaluate_Plus.
```

The goals now should show us the exact match cases we are looking for and we see that they require statements about evaluate ae1 and evaluate ae2. Thus we reason them by cases:

```
    case (evaluate ae1) as [n1 | s1] eqn:E_ae1.
    + case (evaluate ae2) as [n2 | s2] eqn:E_ae2.
      * rewrite -> fold_unfold_evaluate_Literal.
```

Focusing on the case where both ae1 and ae2 evaluate to numbers, we see that we want to prove the equality between the following statements:

```
  match
    (if n1 + 0 <? n2 + 0
     then Expressible_msg ("numerical underflow: -" ++
     ↪  string_of_nat (n2 + 0 - (n1 + 0)))
     else Expressible_nat (n1 + 0 - (n2 + 0)))
  with ...
  =
  match
    (if n1 <? n2
     then Expressible_msg ("numerical underflow: -" ++
     ↪  string_of_nat (n2 - n1))
     else Expressible_nat (n1 - n2))
  with ...
```

In the goal, we see the observation we made about Minus. We see that a 0 is added to the RHS of n1 such that Plus always stays on the top of the refactored binary tree and another 0 is added to the RHS of n2 because the right-most leaf for the refactored binary tree must be 0. We know 0 is absorbing for addition on the right, so we can conclude the proof as such:

```
        rewrite ->2 Nat.add_0_r.
        reflexivity.
      * rewrite -> fold_unfold_evaluate_Literal.
        reflexivity.
    + reflexivity.
Qed.
```

The other branches are not so interesting, as they are the error cases and can be proved via reflexivity. Once this lemma has been proven, we can instantiate it with (Literal 0) in the accumulator, along with an already-proven neutrality of 0 for Plus on the right to prove the main theorem (which we do, in the file).

2.4 Equivalence of the two lemmas

During class, we saw two versions of the lemma we needed to show refactoring preserves evaluation. The version we came to was a generalization of the instance we saw when refactor was unfolded, the other version was a conjunction of three assumptions. Let us consider if they are equivalent. We tried our first attempt via what felt natural, via induction:

```
Proposition equivalence_of_the_two_lemmas_for_refactor_ind :
  forall ae : arithmetic_expression,
    (forall s : string,
        evaluate ae = Expressible_msg s ->
        forall a : arithmetic_expression,
          evaluate (refactor_aux ae a) = Expressible_msg s)
    /\
      (forall (n : nat)
              (s : string),
          evaluate ae = Expressible_nat n ->
          forall a : arithmetic_expression,
            evaluate a = Expressible_msg s ->
            evaluate (refactor_aux ae a) = Expressible_msg s)
    /\
      (forall n1 n2 : nat,
          evaluate ae = Expressible_nat n1 ->
          forall a : arithmetic_expression,
            evaluate a = Expressible_nat n2 ->
            evaluate (refactor_aux ae a) = Expressible_nat (n1 +
            ↪  n2))
    <->
      forall a : arithmetic_expression,
        evaluate (refactor_aux ae a) = evaluate (Plus ae a).
Proof.
  intro ae.
  split.
  - (* Forward direction *).
```

The forward direction was not a problem. We did not even have to use any of the assumptions from the LHS to prove this direction, the inductive hypotheses were enough. However, the converse could have been possible, not using the IHs but the assumptions to prove the forward direction. This

should have been a red flag about using induction for this proof, but our eyes were too glued to the screen to rise above. Then, things became quite chaotic in the reverse direction.

```
      - induction ae as [ n | ae1 IHae1 ae2 IHae2 | ae1 IHae1 ae2
      ↪  IHae2 ].
      (* Abstract illustration of the chaos that soon ensued *)
...
              -- discriminate H_ae.
              -- discriminate H_ae.
...
           reflexivity.
...
            ** intro H_absurd.
               discriminate H_absurd.
...
        reflexivity
... ... ...
```

So, we did prove it in the end via induction, but in none of the cases did we ever need to use the inductive hypothesis. Furthermore, this proof contains multiple `discriminates` because we had to consider all inductive cases of ae1 and ae2, all possible cases for ae1 and ae2 (if they do not have inductive hypotheses), for Plus cases and Minus cases for three propositions. Because we cleared all possible combinations of the above cases, not only was the proof extremely long, there were also instances where we looked at impossible cases (thus cleared with `discriminate`). We now see why this proof takes over 350 lines and have decided to omit most of it in the report.

So, let's do what we have been taught to do if our inductive proof does not (or does not need to) use the inductive hypotheses: consider them by cases.

```
Proposition equivalence_of_the_two_lemmas_for_refactor_case :
  forall ae : arithmetic_expression,
    (forall s : string,
        evaluate ae = Expressible_msg s ->
        forall a : arithmetic_expression,
          evaluate (refactor_aux ae a) = Expressible_msg s)
    /\
```

```
      (forall (n : nat)
              (s : string),
          evaluate ae = Expressible_nat n ->
          forall a : arithmetic_expression,
            evaluate a = Expressible_msg s ->
            evaluate (refactor_aux ae a) = Expressible_msg s)
    /\
      (forall n1 n2 : nat,
          evaluate ae = Expressible_nat n1 ->
          forall a : arithmetic_expression,
            evaluate a = Expressible_nat n2 ->
            evaluate (refactor_aux ae a) = Expressible_nat (n1 +
            ↪  n2))
    <->
      forall a : arithmetic_expression,
        evaluate (refactor_aux ae a) = evaluate (Plus ae a).
Proof.
  intro ae.
  split.
  - intros [E_s [E_n_s E_n_n]] a.
```

In a previous version, we did not use the premises of the implication to prove the RHS, and shortcutted the proof by relying on `refactoring_preserves_evaluation_aux`. In this version, we name and introduce the premises, along with the accumulator, `a`. Fold-unfolding evaluate Plus shows us the nested match cases we must deal with:

```
1 subgoal (ID 822)

  ae : arithmetic_expression
  E_s : forall s : string,
        evaluate ae = Expressible_msg s ->
        forall a : arithmetic_expression, evaluate (refactor_aux
        ↪  ae a) = Expressible_msg s
  E_n_s : forall (n : nat) (s : string),
          evaluate ae = Expressible_nat n ->
          forall a : arithmetic_expression,
          evaluate a = Expressible_msg s ->
```

```
          evaluate (refactor_aux ae a) = Expressible_msg s
  E_n_n : forall n1 n2 : nat,
          evaluate ae = Expressible_nat n1 ->
          forall a : arithmetic_expression,
          evaluate a = Expressible_nat n2 ->
          evaluate (refactor_aux ae a) = Expressible_nat (n1 +
          ↪  n2)
  a : arithmetic_expression
  ============================
  evaluate (refactor_aux ae a) =
  match evaluate ae with
  | Expressible_nat n1 =>
      match evaluate a with
      | Expressible_nat n2 => Expressible_nat (n1 + n2)
      | Expressible_msg s2 => Expressible_msg s2
      end
  | Expressible_msg s1 => Expressible_msg s1
  end
```

We also see that the value of the evaluations are premises for `E_s`, `E_n_s`, and `E_n_n`. Thus, let us observe all four possible cases:

```
case (evaluate ae) as [ n | s ] eqn:E_ae;
      case (evaluate a) as [ n' | s' ] eqn:E_a.
    +
```

1. Both `ae` and `a` evaluate to a nat: Use `E_n_n` to show that the result of evaluating the factored version is `(n + n')`
2. Evaluating `ae` results in a nat, but evaluating `a` results in an error: Use `E_n_s` to show that the result of evaluating the factored version is the error message from `a`
3. Evaluating `ae` results in an error, but evaluating `a` results in a nat: Use `E_s` to show that the result of evaluating the factored version is the error message from `ae`
4. Evaluating `ae` and `a` results in an error: Use `E_s` to show that the result of evaluating the factored version is the error message from `ae`

Note that for cases 3 and 4, we don't have to evaluate `a`, once `ae` returns an error message, but we force the casing by using a semicolon after `case (evaluate ae)`. The forward direction is thus resolved:

```
    + Check (E_n_n n n' (eq_refl (Expressible_nat n)) a E_a).
      exact (E_n_n n n' (eq_refl (Expressible_nat n)) a E_a).
    + Check (E_n_s n s' (eq_refl (Expressible_nat n)) a E_a).
      exact (E_n_s n s' (eq_refl (Expressible_nat n)) a E_a).
    + Check (E_s s (eq_refl (Expressible_msg s)) a).
      exact (E_s s (eq_refl (Expressible_msg s)) a).
    + Check (E_s s (eq_refl (Expressible_msg s)) a).
      exact (E_s s (eq_refl (Expressible_msg s)) a).
  -
```

In the reverse direction, we will consider the 3 possible cases of ae, and consider the 3 subclauses of the conjunction separately. This should give us 3 * 3 = 9 subtrees to consider, and it does:

```
    + intro E.
      split.
      { intros s E_ae_s a.
        rewrite -> fold_unfold_refactor_aux_Literal.
        rewrite -> fold_unfold_evaluate_Plus.
        rewrite -> E_ae_s.
        reflexivity.
      }
      split.
      { intros n' s E_ae_n' a E_a_s.
        rewrite -> fold_unfold_refactor_aux_Literal.
        rewrite -> fold_unfold_evaluate_Plus.
        rewrite -> E_ae_n'.
        rewrite -> E_a_s.
        reflexivity.
      }
      { intros n1 n2 E_ae_n a E_a_n.
        rewrite -> fold_unfold_refactor_aux_Literal.
        rewrite -> fold_unfold_evaluate_Plus.
        rewrite -> E_ae_n.
        rewrite -> E_a_n.
```

```
        reflexivity.
      }
    + (* Plus case - 3 sub-clauses *)
    + (* Minus case - 3 sub-clauses *)
Qed.
```

The proof for the plus case and the minus case involve the same intros-fold_unfold-rewrite-reflexivity loop.

2.5 Idempotence of refactor

We've shown that `refactor` changes the source arithmetic expression but preserves the output of evaluation. We can take it a step further and show that `refactor` is idempotent, that is performing an operation multiple times has the same effect as doing it once.

Although `refactor` is not strictly idempotent in the sense that it alters the arithmetic expression in a non-trivial way, they are idempotent concerning the evaluation of the expression. While the structure of the expression might change with each application of `refactor`, the evaluated output remains consistent. Chaining these operations does not affect the final result of the evaluation, demonstrating their idempotence in terms of the evaluation outcome.

Here is the proposition and proof that `refactor` is idempotent.

```
Proposition refactor_is_idempotent :
  forall (ae : arithmetic_expression),
      evaluate (refactor ae) = evaluate (refactor (refactor ae)).
Proof.
  intro ae.
  rewrite -> (refactoring_preserves_evaluation (refactor ae)).
  reflexivity.
Qed.
```

The proposition states that for all arithmetic expressions, evaluating the refactored arithmetic expression yields the same result as evaluating an arithmetic expression that has been refactored twice. The proof is trivial. We introduce the arithmetic expression, rewrite with the theorem that refactoring preserves evaluation, and finish with reflexivity.

2.6 Conclusion

Throughout the discussion, we were able to show that refactor preserves evaluation and is idempotent. In the process, we (re)came across the following concepts:

1. Refactoring for Literal and Plus is similar to flattening a binary tree with a (Literal 0) at the end of the list
2. Refactoring for Minus is similar to appending two lists (Literal 0) and appending a (Literal 0) at the end of the list
3. If reasoning by cases works, use it over induction for a more concise proof.
4. Characteristics of functions (such as right-associativity) should be observed in proofs that reason about the structure of such functions.
5. When reasoning about premises and conclusions, use the premises of the implication to prove its conclusions, rather than relying on previous lemmas that lets us override the premises.

3 Exercise 2: Super Refactor

3.1 Introduction

In this section, we deal with super refactor, which is a mutually recursively function defined on both `super_refactor` and `super_refactor_aux`. The proof of preserving evaluation most likely requires premises to discuss conditions on those two functions.

3.2 What super refactor does

Let us discuss super refactor in the same manner we discussed refactor:

- Literal n: Same as input
- Plus ae1 ae2: A right-skewed binary tree where Plus is the root, the right-most leaf is the nil case of flatten.
- Minus ae1 ae2: Same as input

Unlike refactor, The root of the resulting binary tree can be a Literal or Minus. Plus flattens the tree, but does so without introducing a new nil case (Literal 0). Minus appends trees ae1 and ae2 with a Minus as the root.

3.3 Proving that super refactor preserves evaluation

Unlike refactor, we cannot unfold super refactor to expose the underlying instantiated general eureka lemma. However, we can reason about it in the following logical manner:

1. Because super refactor is defined mutually recursively on super refactor and super refactor aux, the eureka lemma must reason about both functions.
2. Following the train of thought from above, one side of the conjunction reasons with super refactor.
   `evaluate (super_refactor ae) = evaluate ae` is already a generalized statement.
3. The other side of the conjunction reasons with the auxiliary function. Super refactor behaves the same way refactor does, albeit super refactor uses the right-most leaf as the nil case. Thus, the logic of the auxiliary side of the conjunction should be similar to that of `refactoring_preserves_evaluation_aux`.

Thus, (with the help of Professor Danvy), we formalize the following lemma:

```
Lemma super_refactoring_preserves_evaluation_aux :
  forall ae : arithmetic_expression,
    (evaluate (super_refactor ae) = evaluate ae)
    /\
      (forall a : arithmetic_expression,
          (evaluate (super_refactor_aux ae a) = evaluate (Plus ae
          ↪  a))).
Proof.
```

Let us prepare an induction on ae, with correctly named induction hypotheses. Note that there are two induction hypotheses, one regarding super refactor, the other regarding the auxiliary function. Let us also split each case because we know we will always reason about two sides of the conjunction:

```
  intro ae.
  induction ae as [ n
                  | ae1 [IHae1 IHae1_aux] ae2 [IHae2 IHae2_aux]
                  | ae1 [IHae1 IHae1_aux] ae2 [IHae2 IHae2_aux]
                  ↪  ];
    split.
    -
```

The num case is nothing too special, the correct fold-unfold gets us through.

```
    rewrite -> fold_unfold_super_refactor_Literal.
    reflexivity.
  - intro a.
    rewrite -> fold_unfold_super_refactor_aux_Literal.
    reflexivity.
  -
```

```
ae1 : arithmetic_expression
IHae1 : evaluate (super_refactor ae1) = evaluate ae1
ae2 : arithmetic_expression
IHae1_aux : forall a : arithmetic_expression,
            evaluate (super_refactor_aux ae1 a) =
            evaluate (Plus ae1 a)
IHae2 : evaluate (super_refactor ae2) = evaluate ae2
IHae2_aux : forall a : arithmetic_expression,
            evaluate (super_refactor_aux ae2 a) =
            evaluate (Plus ae2 a)
============================
evaluate (super_refactor (Plus ae1 ae2)) =
evaluate (Plus ae1 ae2)
```

Here, unfolding the super refactor in the Plus case converts the goal into the following:

```
...
============================
evaluate (Plus ae1 (super_refactor ae2)) =
evaluate (Plus ae1 ae2)
```

We see that that's the same as `IHae1_aux` instantiated with `(super_refactor_aux ae2)`. rewriting and unfolding the new Plus:

```
  rewrite -> (IHae1_aux (super_refactor ae2)).
  rewrite -> fold_unfold_evaluate_Plus.
```

```
============================
match evaluate ae1 with
| Expressible_nat n1 =>
    match evaluate (super_refactor ae2) with
    | Expressible_nat n2 =>
        Expressible_nat (n1 + n2)
    | Expressible_msg s2 => Expressible_msg s2
    end
```

```
  | Expressible_msg s1 => Expressible_msg s1
  end = evaluate (Plus ae1 ae2)
```

We can replace `evaluate (super_refactor ae2` with `evaluate ae2` via IHae2, and unfold the RHS to show equality (in the .v file, we folded the LHS for readability).

```
    rewrite -> IHae2.
    rewrite <- fold_unfold_evaluate_Plus.
    reflexivity.
  -
```

```
  ae1 : arithmetic_expression
  IHae1 : evaluate (super_refactor ae1) = evaluate ae1
  ae2 : arithmetic_expression
  IHae1_aux : forall a : arithmetic_expression,
              evaluate (super_refactor_aux ae1 a) =
              evaluate (Plus ae1 a)
  IHae2 : evaluate (super_refactor ae2) = evaluate ae2
  IHae2_aux : forall a : arithmetic_expression,
              evaluate (super_refactor_aux ae2 a) =
              evaluate (Plus ae2 a)
  ============================
  forall a : arithmetic_expression,
  evaluate (super_refactor_aux (Plus ae1 ae2) a) =
  evaluate (Plus (Plus ae1 ae2) a)
```

And we've seen something like this before. The goal and the assumptions are identical to that of refactor in the Plus case, albeit `refactor_aux` is replaced by `super_refactor_aux`. Because we have used uniform naming conventions, we can copy-paste the proof from above here, with slight alterations to the naming. Let us now say the magic word: *mutatis mutandis* and onto the next subcase:

```
  ae1 : arithmetic_expression
  IHae1 : evaluate (super_refactor ae1) = evaluate ae1
  ae2 : arithmetic_expression
```

```
  IHae1_aux : forall a : arithmetic_expression,
              evaluate (super_refactor_aux ae1 a) =
              evaluate (Plus ae1 a)
  IHae2 : evaluate (super_refactor ae2) = evaluate ae2
  IHae2_aux : forall a : arithmetic_expression,
              evaluate (super_refactor_aux ae2 a) =
              evaluate (Plus ae2 a)
  ============================
  evaluate (super_refactor (Minus ae1 ae2)) =
  evaluate (Minus ae1 ae2)
```

We once again unfold the Minus case of super refactor and evaluate until we reach the nested match cases:

```
    rewrite -> fold_unfold_super_refactor_Minus.
    rewrite ->2 fold_unfold_evaluate_Minus.
    rewrite -> IHae1.
```

```
  ============================
  match evaluate (super_refactor ae1) with
  | Expressible_nat n1 =>
      match evaluate (super_refactor ae2) with
      | Expressible_nat n2 => ...
      | Expressible_msg s2 => ...
      end
  | Expressible_msg s1 => ...
  end =
  match evaluate ae1 with
  | Expressible_nat n1 =>
      match evaluate ae2 with
      | Expressible_nat n2 => ...
      | Expressible_msg s2 => ...
      end
  | Expressible_msg s1 => ...
  end
```

Seems like the only difference between the two sides of the equivalence is evaluating an original arithmetic expression and evaluating the super-refactored one. We have assumed them to be true and our inductive hypotheses state the equality. Thus:

```
    rewrite -> IHae1.
    rewrite -> IHae2.
    reflexivity.
```

Cleans the slate for us. Once again, let us reason about the auxiliary case as we did with the vanilla case, by unfolding as much of the as we can to visualize which match cases we're comparing. Because this is the Minus case and match cases become ugly really quickly, we have decided to only unfold the LHS first.

```
    intro a.
    rewrite -> fold_unfold_super_refactor_aux_Minus.
    rewrite -> fold_unfold_evaluate_Plus.
    rewrite -> fold_unfold_evaluate_Minus.
```

```
    1 subgoal (ID 346)

  ae1 : arithmetic_expression
  IHae1 : evaluate (super_refactor ae1) = evaluate ae1
  ae2 : arithmetic_expression
  IHae1_aux : forall a : arithmetic_expression,
              evaluate (super_refactor_aux ae1 a) =
              evaluate (Plus ae1 a)
  IHae2 : evaluate (super_refactor ae2) = evaluate ae2
  IHae2_aux : forall a : arithmetic_expression,
              evaluate (super_refactor_aux ae2 a) =
              evaluate (Plus ae2 a)
  a : arithmetic_expression
  ============================
  match
    match evaluate (super_refactor ae1) with
    | Expressible_nat n1 =>
```

```
        match evaluate (super_refactor ae2) with
        | Expressible_nat n2 => ...
        | Expressible_msg s2 => ...
        end
    | Expressible_msg s1 => Expressible_msg s1
    end
  with
  ...
  ...
  end = evaluate (Plus (Minus ae1 ae2) a)
```

The problem here is that if the two evaluates concerning super refactor can be re-expressed as the vanilla evaluate, we should have no problem. In fact, the induction hypotheses allow us to do exactly that. Then, we can re-fold the LHS with vanilla evaluate to lead to a `reflexivity`.

```
    rewrite -> IHae1.
    rewrite -> IHae2.
    rewrite <- fold_unfold_evaluate_Minus.
    rewrite <- fold_unfold_evaluate_Plus.
    reflexivity.
```

3.4 Equivalence of the two lemmas

Similar to our exploration of refactor, we now examine the equivalence of two formulations of the lemma for super refactor. The proposition we prove is as follows:

```
Proposition equivalence_of_the_two_lemmas_for_super_refactor :
  forall ae : arithmetic_expression,
    (forall s : string,
        evaluate ae = Expressible_msg s ->
        forall a : arithmetic_expression,
          (evaluate (super_refactor ae) = Expressible_msg s)
          /\
            evaluate (super_refactor_aux ae a) = Expressible_msg
            ↪  s)
    /\
```

```
      (forall (n : nat)
              (s : string),
          evaluate ae = Expressible_nat n ->
          forall a : arithmetic_expression,
            evaluate a = Expressible_msg s ->
            (evaluate (super_refactor ae) = Expressible_nat n)
            /\
              evaluate (super_refactor_aux ae a) =
              ↪  Expressible_msg s)
    /\
      (forall n1 n2 : nat,
          evaluate ae = Expressible_nat n1 ->
          forall a : arithmetic_expression,
            evaluate a = Expressible_nat n2 ->
            (evaluate (super_refactor ae) = Expressible_nat n1)
            /\
              evaluate (super_refactor_aux ae a) =
              ↪  Expressible_nat (n1 + n2))
    <->
      (evaluate (super_refactor ae) = evaluate ae)
      /\
        forall a : arithmetic_expression,
          evaluate (super_refactor_aux ae a) = evaluate (Plus ae
          ↪  a).
```

Firstly, we want to note that the formulation of the proposition for the equivalence of the two lemmas for super refactor is slightly different than the proposition for the equivalence of the two lemmas for refactor. Namely, because super refactor is mutually exclusive its lemma is expressed using the following conjunction

```
forall ae : arithmetic_expression,
    (evaluate (super_refactor ae) = evaluate ae)
    /\
      (forall a : arithmetic_expression,
          (evaluate (super_refactor_aux ae a) = evaluate (Plus ae
          ↪  a))).
```

We need to consider the different cases when `ae` and `a` evaluate to different results. To make this concrete, if, for example, `evaluate ae` evaluates to `Expressible_nat n` and `evaluate a` evaluates to `Expressible_msg s`, it makes sense for the first conjunction, `(evaluate (super_refactor ae)`, to evaluate to `Expressible_nat n`, since it only considers `ae`. However, the second conjunction `evaluate (super_refactor_aux ae a)` must evaluate to an error message `Expressible_msg s` since it considers `a`.

Furthermore, we also need to consider the differences between super refactor and super refactor aux where super refactor aux is involved in combining two arithmetic expressions `ae` and `a`. This is why in the example

```
(forall n1 n2 : nat,
          evaluate ae = Expressible_nat n1 ->
          forall a : arithmetic_expression,
            evaluate a = Expressible_nat n2 ->
            (evaluate (super_refactor ae) = Expressible_nat n1)
            /\
              evaluate (super_refactor_aux ae a) =
               ↪  Expressible_nat (n1 + n2))
```

We have `(evaluate (super_refactor ae) = Expressible_nat n1)` and `evaluate (super_refactor_aux ae a) = Expressible_nat (n1 + n2)`.

For the proof itself, we begin similarly with the proof for the equivalence of the two lemmas for refactor. Let us first prove the forward direction. Our first subgoal reads:

```
1 subgoal
(1 unfocused at this level)

ae : arithmetic_expression
E_s : forall s : string,
      evaluate ae = Expressible_msg s ->
      forall a : arithmetic_expression,
      evaluate (super_refactor ae) = Expressible_msg s /\
      evaluate (super_refactor_aux ae a) = Expressible_msg s
E_n_s : forall (n : nat) (s : string),
        evaluate ae = Expressible_nat n ->
        forall a : arithmetic_expression,
        evaluate a = Expressible_msg s ->
        evaluate (super_refactor ae) = Expressible_nat n /\
```

```
        evaluate (super_refactor_aux ae a) = Expressible_msg s
E_n_n : forall n1 n2 : nat,
        evaluate ae = Expressible_nat n1 ->
        forall a : arithmetic_expression,
        evaluate a = Expressible_nat n2 ->
        evaluate (super_refactor ae) = Expressible_nat n1 /\
        evaluate (super_refactor_aux ae a) = Expressible_nat (n1
        ↪  + n2)

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

evaluate (super_refactor ae) = evaluate ae /\
(forall a : arithmetic_expression,
 evaluate (super_refactor_aux ae a) = evaluate (Plus ae a))
```

Here, we notice that we have `evaluate ae` in the goal and the premises, prompting us to proceed by casing `evaluate ae`.

```
1 subgoal
(1 unfocused at this level)

ae : arithmetic_expression
n1 : nat
E_ae : evaluate ae = Expressible_nat n1
E_s : forall s : string,
      Expressible_nat n1 = Expressible_msg s ->
      forall a : arithmetic_expression,
      evaluate (super_refactor ae) = Expressible_msg s /\
      evaluate (super_refactor_aux ae a) = Expressible_msg s
E_n_s : forall (n : nat) (s : string),
        Expressible_nat n1 = Expressible_nat n ->
        forall a : arithmetic_expression,
        evaluate a = Expressible_msg s ->
        evaluate (super_refactor ae) = Expressible_nat n /\
        evaluate (super_refactor_aux ae a) = Expressible_msg s
E_n_n : forall n2 n3 : nat,
        Expressible_nat n1 = Expressible_nat n2 ->
        forall a : arithmetic_expression,
```

```
        evaluate a = Expressible_nat n3 ->
        evaluate (super_refactor ae) = Expressible_nat n2 /\
        evaluate (super_refactor_aux ae a) = Expressible_nat (n2
        ↪  + n3)

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

evaluate (super_refactor ae) = Expressible_nat n1

E_n_n n1 0 eq_refl (Literal 0) (fold_unfold_evaluate_Literal 0)
     : evaluate (super_refactor ae) = Expressible_nat n1 /\
       evaluate (super_refactor_aux ae (Literal 0)) =
       Expressible_nat (n1 + 0)
```

From inspection, we notice that the subgoal is exactly the LHS of the premise `E_n_n`. We destruct on this and use the `exact` tactic to prove the subgoal.

The next subgoal requires us to consider the arithmetic expression `a`, prompting us to case on it too.

```
1 subgoal

ae : arithmetic_expression
n1 : nat
E_ae : evaluate ae = Expressible_nat n1
E_s : forall s : string,
      Expressible_nat n1 = Expressible_msg s ->
      forall a : arithmetic_expression,
      evaluate (super_refactor ae) = Expressible_msg s /\
      evaluate (super_refactor_aux ae a) = Expressible_msg s
E_n_s : forall (n : nat) (s : string),
        Expressible_nat n1 = Expressible_nat n ->
        forall a : arithmetic_expression,
        evaluate a = Expressible_msg s ->
        evaluate (super_refactor ae) = Expressible_nat n /\
        evaluate (super_refactor_aux ae a) = Expressible_msg s
E_n_n : forall n2 n3 : nat,
        Expressible_nat n1 = Expressible_nat n2 ->
```

```
        forall a : arithmetic_expression,
        evaluate a = Expressible_nat n3 ->
        evaluate (super_refactor ae) = Expressible_nat n2 /\
        evaluate (super_refactor_aux ae a) = Expressible_nat (n2
        ↪  + n3)

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

forall a : arithmetic_expression,
evaluate (super_refactor_aux ae a) = evaluate (Plus ae a)
```

From here, we noticed that the subgoals can be similarly proved as the previous subgoal by destructing and exacting on the LHS of the premises `E_n_n` and `E_n_s` respectively.

The last `split` case in the forward direction requires us to consider when `super_refactor ae` and `super_refactor_aux ae a` both evaluate to an error message. For this subgoal, the proof for both conjunctions instead uses the premise `E_s`.

```
1 subgoal

ae : arithmetic_expression
s1 : string
E_ae : evaluate ae = Expressible_msg s1
E_s : forall s : string,
      Expressible_msg s1 = Expressible_msg s ->
      forall a : arithmetic_expression,
      evaluate (super_refactor ae) = Expressible_msg s /\
      evaluate (super_refactor_aux ae a) = Expressible_msg s
E_n_s : forall (n : nat) (s : string),
        Expressible_msg s1 = Expressible_nat n ->
        forall a : arithmetic_expression,
        evaluate a = Expressible_msg s ->
        evaluate (super_refactor ae) = Expressible_nat n /\
        evaluate (super_refactor_aux ae a) = Expressible_msg s
E_n_n : forall n1 n2 : nat,
        Expressible_msg s1 = Expressible_nat n1 ->
        forall a : arithmetic_expression,
```

```
        evaluate a = Expressible_nat n2 ->
        evaluate (super_refactor ae) = Expressible_nat n1 /\
        evaluate (super_refactor_aux ae a) = Expressible_nat (n1
        ↪  + n2)

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

evaluate (super_refactor ae) = Expressible_msg s1 /\
(forall a : arithmetic_expression,
 evaluate (super_refactor_aux ae a) = evaluate (Plus ae a))
```

With that, we are done with the forward direction. On to the proof for the backward direction. After introducing the conjunctions for `super_refactor` and `super_refactor_aux`, we have

```
1 subgoal

ae : arithmetic_expression
E_sr_ae : evaluate (super_refactor ae) = evaluate ae
E_sr_aux_ae : forall a : arithmetic_expression,
              evaluate (super_refactor_aux ae a) = evaluate (Plus
              ↪  ae a)

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

(forall s : string,
 evaluate ae = Expressible_msg s ->
 forall a : arithmetic_expression,
 evaluate (super_refactor ae) = Expressible_msg s /\
 evaluate (super_refactor_aux ae a) = Expressible_msg s) /\
(forall (n : nat) (s : string),
 evaluate ae = Expressible_nat n ->
 forall a : arithmetic_expression,
 evaluate a = Expressible_msg s ->
 evaluate (super_refactor ae) = Expressible_nat n /\
 evaluate (super_refactor_aux ae a) = Expressible_msg s) /\
(forall n1 n2 : nat,
 evaluate ae = Expressible_nat n1 ->
```

```
 forall a : arithmetic_expression,
 evaluate a = Expressible_nat n2 ->
 evaluate (super_refactor ae) = Expressible_nat n1 /\
 evaluate (super_refactor_aux ae a) = Expressible_nat (n1 + n2))
```

We first proceed by proving the case when `evaluate ae` evaluates to an error message

```
1 subgoal
(1 unfocused at this level)

ae : arithmetic_expression
E_sr_ae : evaluate (super_refactor ae) = evaluate ae
E_sr_aux_ae : forall a : arithmetic_expression,
              evaluate (super_refactor_aux ae a) = evaluate (Plus
              ↪  ae a)

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

forall s : string,
evaluate ae = Expressible_msg s ->
forall a : arithmetic_expression,
evaluate (super_refactor ae) = Expressible_msg s /\
evaluate (super_refactor_aux ae a) = Expressible_msg s
```

The proof for this is straightforward, only requiring us to use `rewrites` and `reflexivity`.

```
...
    { intros s E_ae_s a.
      split.
      - rewrite E_ae_s in E_sr_ae.
        exact E_sr_ae.
      - rewrite -> (E_sr_aux_ae a).
        rewrite -> fold_unfold_evaluate_Plus.
        rewrite -> E_ae_s.
        reflexivity.
    }
```

```
...
```

Next we consider the case when `evaluate ae = Expressible_nat n` and `evaluate a = Expressible_msg s`.

```
1 subgoal
(1 unfocused at this level)

ae : arithmetic_expression
E_sr_ae : evaluate (super_refactor ae) = evaluate ae
E_sr_aux_ae : forall a : arithmetic_expression,
              evaluate (super_refactor_aux ae a) = evaluate (Plus
              ↪  ae a)

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

forall (n : nat) (s : string),
evaluate ae = Expressible_nat n ->
forall a : arithmetic_expression,
evaluate a = Expressible_msg s ->
evaluate (super_refactor ae) = Expressible_nat n /\
evaluate (super_refactor_aux ae a) = Expressible_msg s
```

Similarly, the proof for this is straightforward.

```
...
    { intros n s E_ae_n a E_a_s.
      split.
      - rewrite E_ae_n in E_sr_ae.
        exact E_sr_ae.
      - rewrite -> (E_sr_aux_ae a).
        rewrite -> fold_unfold_evaluate_Plus.
        rewrite -> E_ae_n, E_a_s.
        reflexivity.
    }
...
```

Finally, let us solve the last case when `evaluate ae = Expressible_nat n1`

and `evaluate a = Expressible_nat n2`

```
1 subgoal

ae : arithmetic_expression
E_sr_ae : evaluate (super_refactor ae) = evaluate ae
E_sr_aux_ae : forall a : arithmetic_expression,
              evaluate (super_refactor_aux ae a) = evaluate (Plus
              ↪  ae a)

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

forall n1 n2 : nat,
evaluate ae = Expressible_nat n1 ->
forall a : arithmetic_expression,
evaluate a = Expressible_nat n2 ->
evaluate (super_refactor ae) = Expressible_nat n1 /\
evaluate (super_refactor_aux ae a) = Expressible_nat (n1 + n2)
```

The proof for this is also straightforward.

```
...
    { intros n1 n2 E_ae_n a E_a_n.
      split.
      - rewrite E_ae_n in E_sr_ae.
        exact E_sr_ae.
      - rewrite -> (E_sr_aux_ae a).
        rewrite -> fold_unfold_evaluate_Plus.
        rewrite -> E_ae_n, E_a_n.
        reflexivity.
    }
...
```

3.5 Idempotence of super refactor

Just as `refactor` changes the source arithmetic expression but preserves evaluation, so does `super_refactor`. Thus, we can make the same claims about `super_refactor`'s idempotence as we have done for `refactor`.

The proposition and proof that `super_refactor` is idempotent is nearly identical to the proof for `refactor`.

```
Proposition super_refactor_is_idempotent :
  forall (ae : arithmetic_expression),
    evaluate (super_refactor ae) = evaluate (super_refactor
    ↪  (super_refactor ae)).
Proof.
  intro ae.
  rewrite -> (super_refactoring_preserves_evaluation
  ↪  (super_refactor ae)).
  reflexivity.
Qed.
```

Because we have used uniform naming conventions, we can copy-paste the proof from `refactor`, with slight alterations to the naming – *mutatis mutandis*.

3.6 Conclusion

Similar to refactor, we showed that super refactor also preserves evaluation and is idempotent. We also (re)came across the following concepts:

1. Super refactor does nothing when the arithmetic expression is a Literal.

2. If we encounter a Plus, if it is a Plus of two literals, nothing changes. However, if it is a chain of Plus-es, we have a right-skewed binary tree where Plus is the root, and the right-most leaf of the original tree is the nil case of the flattened binary tree associating to the right.

3. If we encounter a Minus of two expressions, we preserve the original expression. Note that the behaviour of Minus is similar to `list_append` where the right-most leaf of the original tree is the accumulator

4. When reasoning about premises and conclusions, use the premises of the implication to prove its conclusions, rather than relying on previous lemmas that let us override the premises

4 Final Conclusion

As we conclude our exploration of refactoring arithmetic expressions using Coq, let's reflect on our journey and the insights we've gained:

1. We examined two refactoring functions, `refactor` and `super_refactor`, understanding their behavior and impact on arithmetic expressions.

2. We proved that both refactoring functions preserve evaluation, ensuring that the refactored expressions yield the same results as the original ones.

3. We explored the equivalence of different formulations of lemmas for both refactoring functions, deepening our understanding of proof strategies.

4. We demonstrated the idempotence of both `refactor` and `super_refactor` with respect to evaluation outcomes.

Now, let us reflect on the key lessons and concepts we've learned and reinforced:

1. Refactoring can significantly alter the structure of expressions while preserving their semantic meaning.

2. Mutual recursion in functions like `super_refactor` requires careful consideration in proofs, often necessitating conjunctive lemmas.

3. The choice between induction and case analysis can greatly impact the clarity and conciseness of proofs.

4. Characteristics of functions, such as right-associativity in `refactor`, often manifest in the structure of proofs about those functions.

5. When proving equivalence, it's crucial to consider all possible evaluation outcomes and their interactions, but it is equivalently important to be mindful of whether the sub-branch of the proof occurs in practice of not (ex: in an ltr programme, if the ae1 evaluates to an error, we do not have to consider the what ae2 evaluates to).

6. Idempotence in the context of program transformations can be nuanced, applying to evaluation outcomes rather than structural equality.

   "The solution of a problem lies in the understanding of the problem; the answer is not outside the problem, it is in the problem." - *Jiddu Krishnamurti*