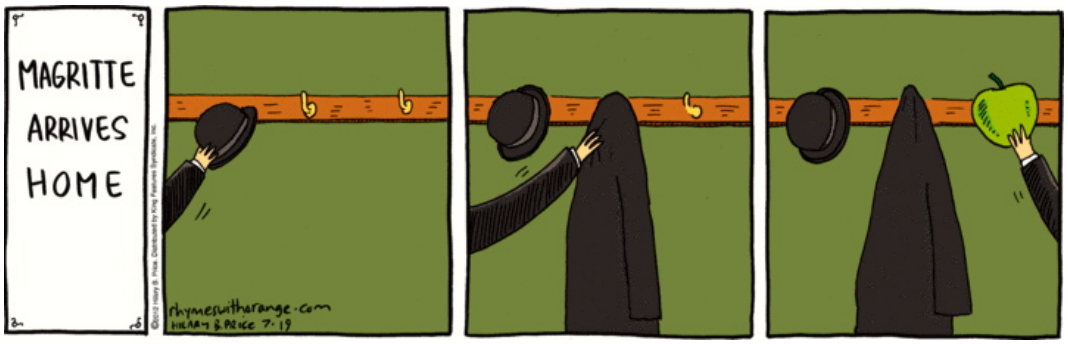
YSC3236: Functional Programming and Proving

Term Project: Three Language Processors for Arithmetic Expressions

Zhang Liu, A0190879J



Everything we see hides another thing, we always want to see what is hidden by what we see. There is an interest in that which is hidden and which the visible does not show us. This interest can take the form of a quite intense feeling, a sort of conflict, one might say, between the visible that is hidden and the visible that is present.

– René Margritte

Contents

1 Introduction

In this term project, we fulfil an earlier promise: to formalize an interpreter and compiler for arithmetic expressions. Back in Intro to CS, we implemented the interpreter and compiler and concluded with the commutativtiy test. Now, we will finally use what we have learned in Functional Programming and Proving to rigorously *prove* what we have *tested* back then.

1.1 The structure of the repot

For tasks 1 through 8, we implement and *prove* the functions that were previously implemented and *tested* in Intro to CS, namely, `evaluate`, `interpret`, `decode_execute`, `fetch_decode_execute`, `run`, `compile_aux`, `compile`, `compile_alt` (`compile` with accumulator). For any of the functions, its specification informs us about the the structure of cases. The proofs and definition associated with this particular function also has the same structure. Thus, in each task, the the report is also structured in the same way to highlight the systematic nature of the proofs. For each task, I will start with listing the cases contingent to that specific function (which can be observed from the specifications). Then I will use this list to structure the subtasks, namely the proof (about non-ambiguous specification) in subtask a, the definition in subtask b, and the proof (about satisfying the specification) in subtask c. I will also provide a flowchart diagram for longer proofs that involve many nested cases.

We will conclude with a proof of the commutative diagram in task 9, an investigation about byte-code verification and finally relook at everything in the Magritte world.

2 Task 1: interpreter for the source program

2.1 Introduction

In Task 1, we focus on definitions and theorems about the interpreter for the source program, namely the functions `evaluate` and `interpret`. In this task, we first proved that the given specifications of `evaluate` and `interpret` are not ambiguous. We then defined `evaluate` and `interpret` and showed that they satisfy the given specifications respectively.

The `evaluate` function is a recursive function (hence it is defined using Fixedpoint). Thus, the proofs in this section are proofs by induction (on the arithmetical expression ae) and we need to consider separately the base case and the induction case. In addition, within the induction case, we have two cases: PLUS and MINUS, each of which in turn has several subcases to consider.

- Base case: (Literal n)
- Induction case:
  - (i) (Plus ae1 ae2)
    - (a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value
    - (b) evaluating ae1 gives a nat value and evaluating ae2 gives an error message
    - (c) evaluating ae1 gives an error message
  - (ii) (Minus ae1 ae2)
    - (a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is true (there is numerical underflow);
    - (b) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is false (there is NO numerical underflow);
    - (c) evaluating ae1 gives a nat value and evaluating ae2 gives an error message;
    - (d) evaluating ae1 gives an error message;

2.2 1a: Theorem 1: there is at most one evaluate function.

First, we prove that there is at most one evaluate function. This confirms that the given specification of evaluate is not ambiguous.

This is a proof by induction on ae. An overview for the structure of the proof is shown in Figure 1 below.

First we specify the base case and induction case with

```
induction ae as [n | ae1 IHae1 ae2 IHae2 | ae1 IHae1 ae2 IHae2].
```

which gives us the goal window below:

```
3 subgoals (ID 38)

  evaluate_1, evaluate_2 : arithmetic_expression -> expressible_value
  S_evaluate_1 : specification_of_evaluate evaluate_1
  S_evaluate_2 : specification_of_evaluate evaluate_2
  n : nat
```

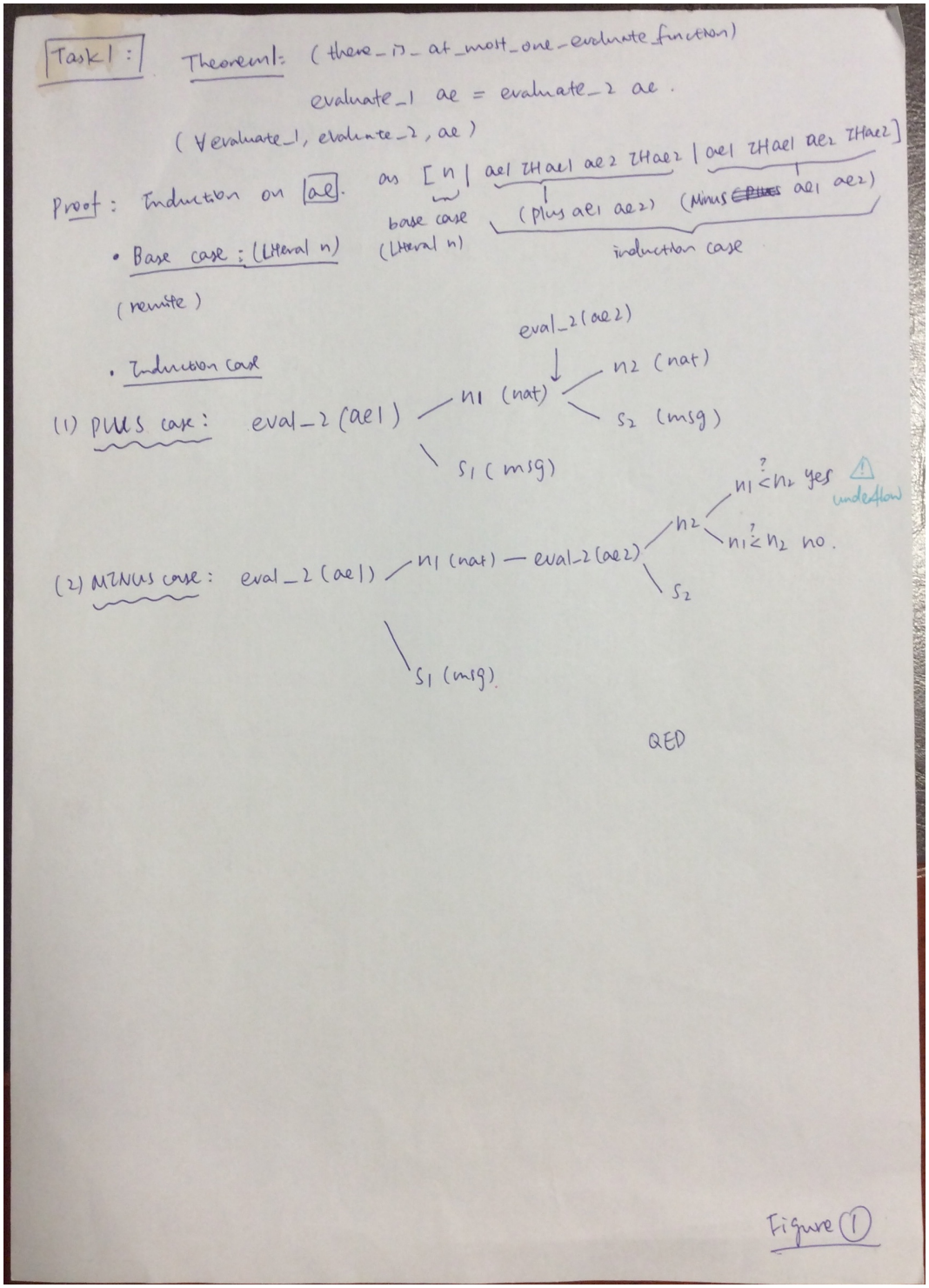


Figure 1: Figure 1

```
  ============================
  evaluate_1 (Literal n) = evaluate_2 (Literal n)

subgoal 2 (ID 43) is:
 evaluate_1 (Plus ae1 ae2) = evaluate_2 (Plus ae1 ae2)
subgoal 3 (ID 48) is:
 evaluate_1 (Minus ae1 ae2) = evaluate_2 (Minus ae1 ae2)
```

- Base case: (Literal n)

  The goal window for this case reads:

  ```
  1 subgoal (ID 38)

    evaluate_1, evaluate_2 : arithmetic_expression -> expressible_value
    S_evaluate_1 : specification_of_evaluate evaluate_1
    S_evaluate_2 : specification_of_evaluate evaluate_2
    n : nat
    ============================
    evaluate_1 (Literal n) = evaluate_2 (Literal n)
  ```

  For the base case, we use the corresponding hypotheses for this case, `[H_Literal_1` and `[H_Literal_2` as follows:

  ```
  [H_Literal_1 _].
  destruct S_evaluate_2 as [H_Literal_2 _].
  rewrite -> (H_Literal_1 n).
  rewrite -> (H_Literal_2 n).
  reflexivity.
  ```

- Induction Case: (Plus ae1 ae2)

  Now we focus on the case for `Plus`. There are three subcases under `Plus`, namely:

  (i) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value

  (ii) evaluating ae1 gives a nat value and evaluating ae2 gives an error message

  (iii) evaluating ae1 gives an error message

  We will focus on each and prove one by one.

  (i) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value:
  We use the corresponding hypotheses for this case, `H_Plus_1` and `H_Plus_2` as follows:

  ```
  destruct S_evaluate_1 as [_ [[_ [_ H_Plus_1]] _]].
  destruct S_evaluate_2 as [_ [[_ [_ H_Plus_2]] _]].
  rewrite -> (H_Plus_2 ae1 ae2 n1 n2 IHae1' IHae2').
  exact (H_Plus_1 ae1 ae2 n1 n2 IHae1 IHae2).
  ```

  (ii) evaluating ae1 gives a nat value and evaluating ae2 gives an error message
  We use the corresponding hypotheses for this case, `H_Plus_1_error2` and `H_Plus_2_error2` as follows:

```
destruct S_evaluate_1 as [_ [[_ [H_Plus_1_error2 _]] _]].
destruct S_evaluate_2 as [_ [[_ [H_Plus_2_error2 _]] _]].
Check (H_Plus_2_error2 ae1 ae2 n1 s2 IHae1' IHae2').
rewrite -> (H_Plus_2_error2 ae1 ae2 n1 s2 IHae1' IHae2').
Check (H_Plus_1_error2 ae1 ae2 n1 s2 IHae1 IHae2).
exact (H_Plus_1_error2 ae1 ae2 n1 s2 IHae1 IHae2).
```

(iii) evaluating ae1 gives an error message

We use the corresponding hypotheses for this case, `H_Plus_1_error1` and `H_Plus_2_error1` as follows:

```
  destruct S_evaluate_1 as [_ [[H_Plus_1_error1 _] _]].
destruct S_evaluate_2 as [_ [[H_Plus_2_error1 _] _]].
Check (H_Plus_2_error1 ae1 ae2 s1 IHae1').
rewrite -> (H_Plus_2_error1 ae1 ae2 s1 IHae1').
Check (H_Plus_1_error1 ae1 ae2 s1 IHae1).
exact (H_Plus_1_error1 ae1 ae2 s1 IHae1).
```

- Case 3: (Minus ae1 ae2)

  Minus has a similar case structure with Plus, except that for Minus we have an additional consideration for numerical underflow. Therefore, there are 4 cases for Minus. Specifically these cases are:

  (i) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is true (there is numerical underflow);

  (ii) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is false (there is NO numerical underflow);

  (iii) evaluating ae1 gives a nat value and evaluating ae2 gives an error message;

  (iv) evaluating ae1 gives an error message;

  This structure is also visualized in Figure 1. In what follows we will explain details of the proof below:

  (i) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is true (there is numerical underflow);

  We use the corresponding hypotheses for this case, `H_Minus_1_underflow` and `H_Minus_2_underflow` as follows:

  ```
  destruct S_evaluate_1 as [_ [_ [_ [_ [H_Minus_1_underflow _]]]]].
  destruct S_evaluate_2 as [_ [_ [_ [_ [H_Minus_2_underflow _]]]]].
  rewrite -> (H_Minus_2_underflow ae1 ae2 n1 n2 IHae1' IHae2' H_underflow).
  exact (H_Minus_1_underflow ae1 ae2 n1 n2 IHae1 IHae2 H_underflow).
  ```

  (ii) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is false (there is NO numerical underflow);

  We use the corresponding hypotheses for this case, `H_Minus_1` and `H_Minus_2` as follows:

```
destruct S_evaluate_1 as [_ [_ [_ [_ [_ H_Minus_1]]]]].
destruct S_evaluate_2 as [_ [_ [_ [_ [_ H_Minus_2]]]]].
rewrite -> (H_Minus_2 ae1 ae2 n1 n2 IHae1' IHae2' H_underflow).
exact (H_Minus_1 ae1 ae2 n1 n2 IHae1 IHae2 H_underflow).
```

(iii) evaluating ae1 gives a nat value and evaluating ae2 gives an error message;

We use the corresponding hypotheses for this case, `H_Minus_1_error2` and `H_Minus_2_error2` as follows:

```
destruct S_evaluate_1 as [_ [_ [_ [H_Minus_1_error2 _]]]].
 destruct S_evaluate_2 as [_ [_ [_ [H_Minus_2_error2 _]]]].
 rewrite -> (H_Minus_2_error2 ae1 ae2 n1 s2 IHae1' IHae2').
 exact (H_Minus_1_error2 ae1 ae2 n1 s2 IHae1 IHae2).
```

(iv) evaluating ae1 gives an error message;

We use the corresponding hypotheses for this case, `H_Minus_1_error1` and `H_Minus_2_error1` as follows:

```
 destruct S_evaluate_1 as [_ [_ [H_Minus_1_error1 _]]].
destruct S_evaluate_2 as [_ [_ [H_Minus_2_error1 _]]].
rewrite -> (H_Minus_2_error1 ae1 ae2 s1 IHae1').
exact (H_Minus_1_error1 ae1 ae2 s1 IHae1).
```

2.3 1b: Definition 2: implement evaluate function and its fold-unfold lemmas

Recall that in the introduction of this task we have summarized the structure based on the specification of `evaluate`:

- Base case: (Literal n)
- Induction case:
  - (i) (Plus ae1 ae2)
    - (a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value
    - (b) evaluating ae1 gives a nat value and evaluating ae2 gives an error message
    - (c) evaluating ae1 gives an error message
  - (ii) (Minus ae1 ae2)
    - (a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is true (there is numerical underflow);
    - (b) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is false (there is NO numerical underflow);
    - (c) evaluating ae1 gives a nat value and evaluating ae2 gives an error message;
    - (d) evaluating ae1 gives an error message;

The implementation of the `evaluate` function mirrors exactly that structure:

```
Fixpoint evaluate (ae : arithmetic_expression) : expressible_value :=
 match ae with
```

```
  | Literal n =>
    Expressible_nat n
  | Plus ae1 ae2 =>
    match evaluate ae1 with
    | Expressible_nat n1 =>
      match evaluate ae2 with
      | Expressible_nat n2 =>
        Expressible_nat (n1 + n2)
      | Expressible_msg s =>
        Expressible_msg s
      end
    | Expressible_msg s =>
      Expressible_msg s
    end
  | Minus ae1 ae2 =>
    match evaluate ae1 with
    | Expressible_nat n1 =>
      match evaluate ae2 with
      | Expressible_nat n2 =>
        if n1 <? n2
        then Expressible_msg
               (String.append
                  "numerical underflow: -"
                  (string_of_nat (n2 - n1)))
        else Expressible_nat (n1 - n2)
      | Expressible_msg s =>
        Expressible_msg s
      end
    | Expressible_msg s =>
      Expressible_msg s
    end
end.
```

2.4 1c: Theorem 3: evaluate satisfies the specification of evaluate.

This is a direct proof by cases and is contingent on the parallel between fold-unfold lemmas and the components of the specification. An overview for the structure of the proof is shown in Figure 2 below.

First we unfold the specification of evaluate. The goal window shows that we have the same structure for the cases as before, namely:

- Base case: (Literal n)
- Induction case:

  (i) (Plus ae1 ae2)

  (a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value

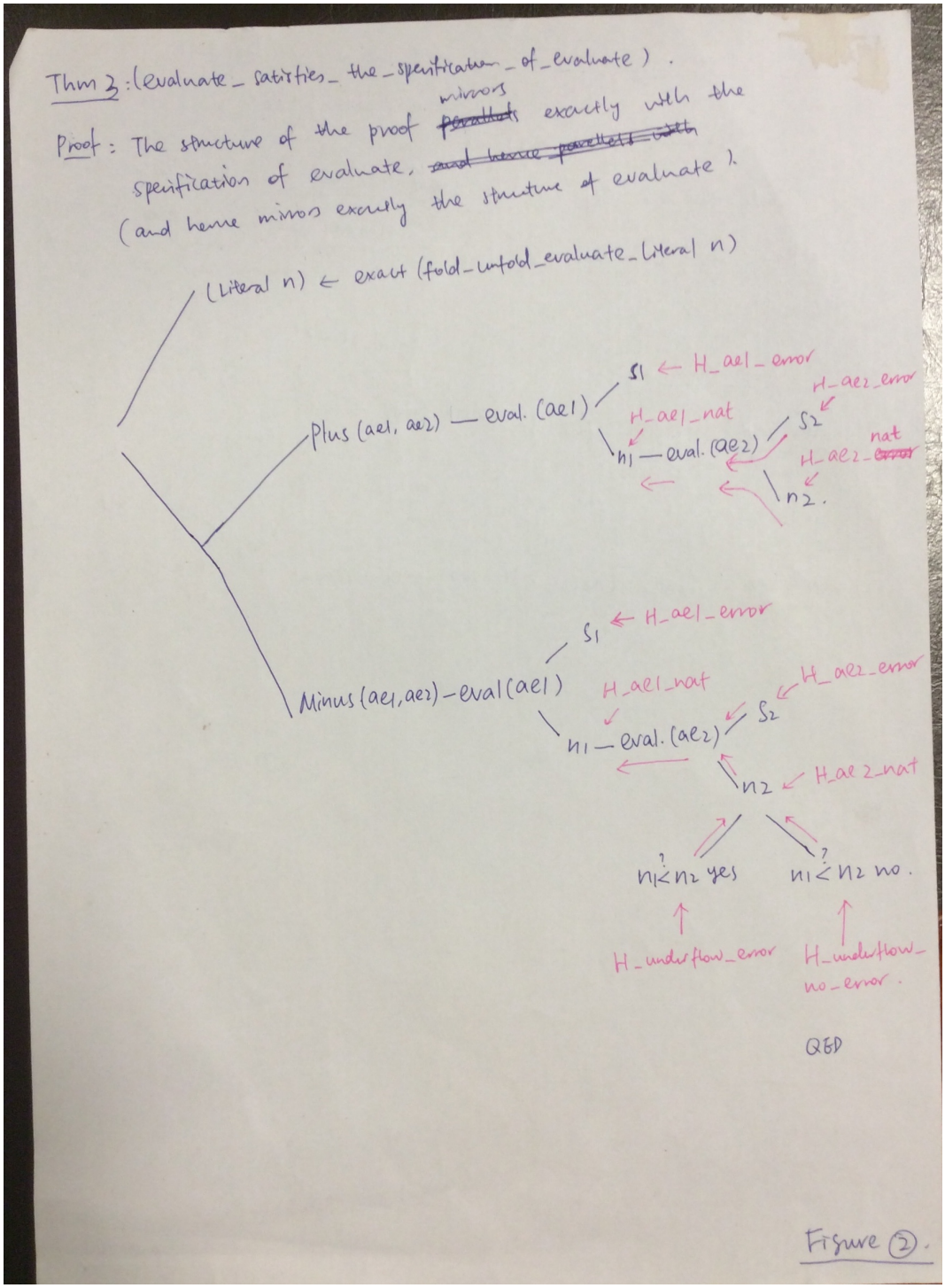


Figure 2: Figure 2

(b) evaluating ae1 gives a nat value and evaluating ae2 gives an error message
(c) evaluating ae1 gives an error message

(ii) (Minus ae1 ae2)

(a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is true (there is numerical underflow);
(b) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is false (there is NO numerical underflow);
(c) evaluating ae1 gives a nat value and evaluating ae2 gives an error message;
(d) evaluating ae1 gives an error message;

We focus on each subgoal and prove one by one.

(i) Case 1: (Literal n)

The goal window for this case reads:

```
1 subgoal (ID 36)

  ============================
  forall n : nat, evaluate (Literal n) = Expressible_nat n
```

We first introduce the variable n and use the fold-unfold lemma of evaluate (for the literal case) to complete this subgoal.

(ii) Case 2: (Plus ae1 ae2)

Then we focus on the case for `Plus`. The goal window now shows:

```
1 subgoal (ID 40)

  ============================
  (forall
     (ae1 ae2 : arithmetic_expression)
     (s1 : string),
   evaluate ae1 = Expressible_msg s1 ->
   evaluate (Plus ae1 ae2) =
   Expressible_msg s1) /\
  (forall
     (ae1 ae2 : arithmetic_expression)
     (n1 : nat) (s2 : string),
   evaluate ae1 = Expressible_nat n1 ->
   evaluate ae2 = Expressible_msg s2 ->
   evaluate (Plus ae1 ae2) =
   Expressible_msg s2) /\
  (forall
     (ae1 ae2 : arithmetic_expression)
     (n1 n2 : nat),
```

```
evaluate ae1 = Expressible_nat n1 ->
evaluate ae2 = Expressible_nat n2 ->
evaluate (Plus ae1 ae2) =
Expressible_nat (n1 + n2))
```

This suggests to us that there are three subcases under `Plus`, namely:

(a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value

(b) evaluating ae1 gives a nat value and evaluating ae2 gives an error message

(c) evaluating ae1 gives an error message

We will focus on each and prove one by one.

(a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value
We use the corresponding hypotheses for this case, `H_ae1_nat` and `H_ae1_nat` as follows:

```
intros ae1 ae2 n1 n2 H_ae1_nat H_ae2_nat.
rewrite -> fold_unfold_evaluate_Plus.
rewrite -> H_ae2_nat.
rewrite -> H_ae1_nat.
reflexivity.
```

(b) evaluating ae1 gives a nat value and evaluating ae2 gives an error message
We use the corresponding hypotheses for this case, `H_ae1_nat` and `H_ae2_error` as follows:

```
intros ae1 ae2 n1 s2 H_ae1_nat H_ae2_error.
 rewrite -> fold_unfold_evaluate_Plus.
 rewrite -> H_ae2_error.
 rewrite -> H_ae1_nat.
 reflexivity.
```

(c) evaluating ae1 gives an error message:
We use the corresponding hypotheses for this case, `H_ae1_error` as follows:

```
intros ae1 ae2 s H_ae1_error.
rewrite -> fold_unfold_evaluate_Plus.
rewrite -> H_ae1_error.
```

(iii) Case 3: (Minus ae1 ae2)

Minus has a similar case structure with Plus, except that for Minus we have an additional consideration for numerical underflow. Therefore, there are 4 cases for Minus. Specifically these cases are:

(a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is true (there is numerical underflow);

(b) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is false (there is NO numerical underflow);

(c) evaluating ae1 gives a nat value and evaluating ae2 gives an error message;

(d) evaluating ae1 gives an error message;

Each of these cases are proved using the corresponding hypotheses. For example, whenever evaluating ae1 gives an error message, we rewrite using `H_ae1_error`; whenever evaluating ae1 gives a nat value, we rewrite using `H_ae1_nat`. The proof structure is also visualized in Figure 2. We explain details of the proof below:

(i) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is true (there is numerical underflow);

We use the corresponding hypotheses for this case as follows:

```
intros ae1 ae2 n1 n2 H_ae1_nat H_ae2_nat H_underflow_error.
rewrite -> fold_unfold_evaluate_Minus.
rewrite -> H_ae2_nat.
rewrite -> H_ae1_nat.
rewrite -> H_underflow_error.
reflexivity.
```

(ii) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is false (there is NO numerical underflow);

We use the corresponding hypotheses for this case as follows:

```
intros ae1 ae2 n1 n2 H_ae1_nat H_ae2_nat H_underflow_noerror.
rewrite -> fold_unfold_evaluate_Minus.
rewrite -> H_ae2_nat.
rewrite -> H_ae1_nat.
rewrite -> H_underflow_noerror.
reflexivity.
```

(iii) evaluating ae1 gives a nat value and evaluating ae2 gives an error message;

We use the corresponding hypotheses for this case, as follows:

```
intros ae1 ae2 n1 s2 H_ae1_nat H_ae2_error.
rewrite -> fold_unfold_evaluate_Minus.
rewrite -> H_ae2_error.
rewrite -> H_ae1_nat.
reflexivity.
```

(iv) evaluating ae1 gives an error message;

We use the corresponding hypotheses for this case, `H_Minus_1_error1` and `H_Minus_2_error1` as follows:

```
intros ae1 ae2 s H_ae1_error.
rewrite -> fold_unfold_evaluate_Minus.
rewrite -> H_ae1_error.
reflexivity.
```

2.5 1d: Theorem 4: there is at most one interpret function.

This theorem states that given two `interpret` functions, they give the same output, i.e., there is no ambiguity in the specificaiton of the `interpret` function. This is a direct corollary of Theorem 3: evaluate satisfies the specification of evaluate. Specifically, the steps of the proof is shown below:

```
intros interpret_1 interpret_2.
unfold specification_of_interpret.
intros H_interpret_1 H_interpret_2.
intro ae.
rewrite -> (H_interpret_2 evaluate evaluate_satisfies_the_specification_of_evaluate
↪  ae).
exact (H_interpret_1 evaluate evaluate_satisfies_the_specification_of_evaluate ae).
```

2.6 1e: Definition 5: interpret function and its fold-unfold lemmas

Interpret takes in a source program and passes it through the predefined evaluate function, which in turn gives an expressible value. Thus, the definition is simply instantiating the evaluate function with arithmetic expression:

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

2.7 1f: Theorem 6: interpret satisfies the specification of interpret.

This is a direct corollary of (Theorem 1: there is at most one evaluate function) and (Theorem 3: evaluate satisfies the specification of evaluate). The specific steps for the proof are shown:

```
unfold specification_of_interpret, interpret.
intros evaluate' S_evaluate' ae.
exact (there_is_at_most_one_evaluate_function evaluate evaluate'
↪  evaluate_satisfies_the_specification_of_evaluate S_evaluate' ae).
```

3 Task 2: decode execute

3.1 Introduction

In this task, we first proved that the given specification of `decode_execute` is not ambiguous. We then defined the decode execute function and showed that it satisfies the given specification.

The `decode_execute_loop` function is a recursive function. Thus, the proofs in this section are proofs by induction (on bci) and we need to consider separately the base case and the induction case. In addition, within the induction case, we have two cases: ADD and SUB, each of which in turn has several subcases to consider.

The case structure specific to this task is:

- Base case: Push n
- Induction Cases:
  - (i) ADD
    - (a) there is no element in the stack (stack underflow)
    - (b) there is only one element in the stack (stack underflow)
    - (c) there are two elements in the stack (ADD is well-defined on them)
  - (ii) SUB
    - (a) there is no element in the stack (stack underflow)
    - (b) there is only one element in the stack (stack underflow)
    - (c) there are two elements in the stack (SUB is well-defined on them)
      - $n_1 < n_2$ is true, there is numerical underflow
      - $n_1 < n_2$ is false, there is NO numerical underflow

3.2 2a: Theorem 7: there is at most one decode execute function.

First, we prove that there is at most one evaluate function. This confirms that the given specification of evaluate is not ambiguous.

The structure of the proof is shown in the Figure 3 below.

More specifically, after the preliminaries (intro variables and hypotheses), we then do induction on the byte-code instruction (bci) with

```
induction bci as [ n | | ].
```

The current goal window shows the base case and induction cases:

```
  ============================
  dec_exec_1 (PUSH n) ds = dec_exec_2 (PUSH n) ds

subgoal 2 (ID 73) is:
 forall ds : data_stack, dec_exec_1 ADD ds = dec_exec_2 ADD ds
subgoal 3 (ID 74) is:
 forall ds : data_stack, dec_exec_1 SUB ds = dec_exec_2 SUB ds
```

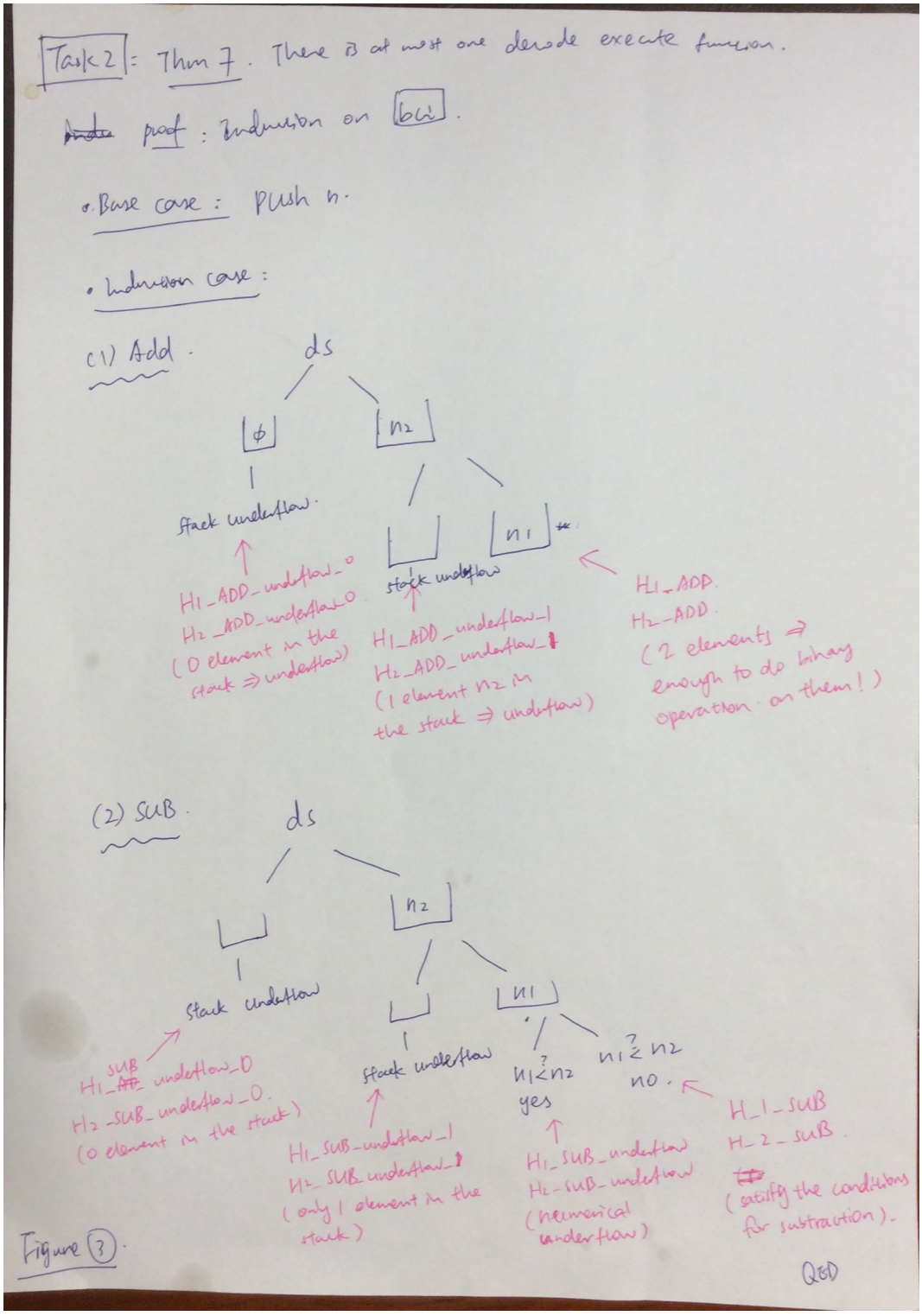


Figure 3: Figure 3

- Base case: Push n

  In the base case, we can use the hypothesis to rewrite:

```
rewrite -> H_1_Push.
rewrite -> H_2_Push.
```

- Induction Cases:

  (i) ADD

  (a) there is no element in the stack (stack underflow)

```
rewrite -> H_1_ADD_underflow0.
rewrite -> H_2_ADD_underflow0.
reflexivity.
```

  (b) there is only one element in the stack (stack underflow)

```
rewrite -> H_1_ADD_underflow1.
rewrite -> H_2_ADD_underflow1.
reflexivity.
```

  (c) there are two elements in the stack (ADD is well-defined on them)

```
rewrite -> H_1_ADD.
rewrite -> H_2_ADD.
reflexivity.
```

  (ii) SUB

  (a) there is no element in the stack (stack underflow)

```
rewrite -> H_1_SUB_underflow0.
rewrite -> H_2_SUB_underflow0.
reflexivity.
```

  (b) there is only one element in the stack (stack underflow)

```
rewrite -> H_1_SUB_underflow1.
rewrite -> H_2_SUB_underflow1.
reflexivity.
```

  (c) there are two elements in the stack (SUB is well-defined on them)

  – $n_1 < n_2$ is true, there is numerical underflow

```
rewrite -> (H_1_SUB_underflow n1 n2 ds H_underflow).
rewrite -> (H_2_SUB_underflow n1 n2 ds H_underflow).
reflexivity.
```

  – $n_1 < n_2$ is false, there is NO numerical underflow

```
rewrite -> (H_1_SUB n1 n2 ds H_underflow).
rewrite -> (H_2_SUB n1 n2 ds H_underflow).
reflexivity.
```

3.3 2b: Definition 8: decode execute function (non-recursive)

This directly mirrors the specification. To elaborate, there are three possibilities for bci: PUSH n, ADD, or SUB.

(i) PUSH n

In this case, we will always return OK and add n iteself to the data stack.

(ii) ADD

If there are only zero or one element in the stack, returns KO and an error message indicating stack underflow.

If there are two elements, returns OK and add the result of addition in the data stack.

(iii) SUB

If there are only zero or one element in the stack, returns KO and an error message indicating stack underflow (KO).

If there are two elements, check if $n_1 < n_2$. If yes, returns KO and an error message indicating numerical underflow. Otherwise, returns OK and add the result of subtraction in the data stack.

Given the above, we implement the `decode_execute` function:

```
Definition decode_execute (bcis : byte_code_instruction) (ds : data_stack) :
↪  result_of_decoding_and_execution :=
  match bcis with
  | PUSH n =>
    OK (n :: ds)
  | ADD =>
    match ds with
    | nil =>
      KO "ADD: stack underflow"
    | n2 :: nil =>
      KO "ADD: stack underflow"
    | n2 :: n1 :: ds =>
      OK ((n1 + n2) :: ds)
    end
  | SUB =>
    match ds with
    | nil =>
      KO "SUB: stack underflow"
    | n2 :: nil =>
      KO "SUB: stack underflow"
    | n2 :: n1 :: ds =>
      if n1 <? n2
      then KO (String.append "numerical underflow: -" (string_of_nat (n2 - n1)))
      else OK ((n1 - n2) :: ds)
    end
  end.
```

3.4 2c: Theorem 9: decode execute satisfies the specification of decode execute.

This is a direct proof by cases. The structure once again parallels that which is given in the introduction of this task: The case structure specific to this task is:

- Base case: Push n
- Induction Cases:
  (i) ADD
    (a) there is no element in the stack (stack underflow)
    (b) there is only one element in the stack (stack underflow)
    (c) there are two elements in the stack (ADD is well-defined on them)
  (ii) SUB
    (a) there is no element in the stack (stack underflow)
    (b) there is only one element in the stack (stack underflow)
    (c) there are two elements in the stack (SUB is well-defined on them)
      – $n_1 < n_2$ is true, there is numerical underflow
      – $n_1 < n_2$ is false, there is NO numerical underflow

Thus, we give the specific steps for the proof:

```
split.
- intros n ds.
  reflexivity.
- split.
  * split.
    + reflexivity.
    + split.
      intro n.
      reflexivity.

      intros n1 n2 ds.
      reflexivity.

  * split.
    + reflexivity.
    + split.
      intro n.
      reflexivity.

      split.
      intros n1 n2 ds H_underflow.
      rewrite -> H_underflow.
      reflexivity.

      intros n1 n2 ds H_no_underflow.
      rewrite -> H_no_underflow.
      reflexivity.
```

The above code can be made more compact (albeit less readable):

```
  unfold specification_of_decode_execute, decode_execute.
split; [intros n ds | split; [split; [ | split; [intro n | intros n1 n2 ds]] | split; [
↪  | split; [intro n | split; intros n1 n2 ds H_underflow; rewrite -> H_underflow]]];
↪  reflexivity.
```

4 Task 3: fetch decode execute loop

4.1 Introduction

In this task, we first proved that the given specification of `fetch_decode_execute_loop` is not ambiguous. We then defined the `fetch_decode_execute_loop` function and showed that it satisfies the given specification.

The `fetch_decode_execute_loop` function is a recursive function. Thus, the proofs in this section are proofs by induction (on bcis) and we need to consider separately the base case and the induction case.

4.2 3a: Theorem 10: there is at most one fetch decode execute loop function.

This is an induction proof. After the preliminaries, we do induction on bcis by using

```
induction bcis as [ | bci' bcis' IHbcis']; intro ds.
```

The goal window thus reads:

```
  ============================
  fdel_1 nil ds = fdel_2 nil ds

subgoal 2 (ID 113) is:
 fdel_1 (bci' :: bcis') ds =
 fdel_2 (bci' :: bcis') ds
```

This gives us a base case and induction case:

- Base case: rewrite using the hypothesis.
- Induction Case:

```
 - case (decode_execute bci' ds) as [ds' | s'] eqn:H_bcis_con.
+ rewrite -> (H_fdel_1_cons bci' bcis' ds ds' H_bcis_con).
  rewrite -> (H_fdel_2_cons bci' bcis' ds ds' H_bcis_con).
  exact (IHbcis' ds').
+ rewrite -> (H_fdel_1_error bci' bcis' ds s' H_bcis_con).
  rewrite -> (H_fdel_2_error bci' bcis' ds s' H_bcis_con).
  reflexivity.
```

4.3 3b: Definition 11: fetch decode execute loop function and its fold-unfold lemmas

`fetch_decode_execute_loop` is defined recursively with the base case nil =¿ OK ds—; and the induction case:

```
| bci :: bcis' =>
 match (decode_execute bci ds) with
 | OK ds' => fetch_decode_execute_loop bcis' ds'
 | KO s => KO s
 end
```

Thus we have:

```
Fixpoint fetch_decode_execute_loop (bcis : list byte_code_instruction) (ds : data_stack)
↪  : result_of_decoding_and_execution :=
  match bcis with
  | nil => OK ds
  | bci :: bcis' =>
    match (decode_execute bci ds) with
    | OK ds' => fetch_decode_execute_loop bcis' ds'
    | KO s => KO s
    end
  end.
```

We then state its associated fold-unfold lemmas (with this being almost second nature, as it should be).

```
Lemma fold_unfold_fetch_decode_execute_loop_nil :
  forall ds : data_stack,
    fetch_decode_execute_loop nil ds = OK ds.
Proof.
  fold_unfold_tactic fetch_decode_execute_loop.
Qed.

Lemma fold_unfold_fetch_decode_execute_loop_cons :
  forall (bci: byte_code_instruction)
         (bcis': list byte_code_instruction)
         (ds : data_stack),
    fetch_decode_execute_loop (bci :: bcis') ds =
    match decode_execute bci ds with
    | OK ds' => fetch_decode_execute_loop bcis' ds'
    | KO s => KO s
    end.
Proof.
  fold_unfold_tactic fetch_decode_execute_loop.
Qed.
```

4.4 3c: Theorem 12: fetch decode execute loop satisfies the specification.

After unfold and intros, we get a conjunction of three statements. With split tactic, we focus on each of the three one by one.

(i) The first subgoal:

```
  ============================
fetch_decode_execute_loop nil ds =
OK ds
```

We prove this subgoal simply by using the fold-unfold lemma:

```
intro ds.
rewrite -> (fold_unfold_fetch_decode_execute_loop_nil ds).
reflexivity.
```

(ii) The second subgoal:

```
============================
fetch_decode_execute_loop
  (bci :: bcis') ds =
fetch_decode_execute_loop bcis' ds'
```

This subgoal is a direct corollary of (Theorem 7: there is at most one decode execute function) and (Theorem 9: decode execute satisfies the specification of decode execute), after which we can rewrite with the hypothesis corresponding to thsi case, `H_OK`. Thus the steps are as follows:

```
intros bci bcis' ds ds'.
intro H_OK.
rewrite -> (fold_unfold_fetch_decode_execute_loop_cons bci bcis' ds).
rewrite -> (there_is_at_most_one_decode_execute_function decode_execute'
↪  decode_execute H_decode_execute
↪  decode_execute_satisfies_the_specification_of_decode_execute bci ds) in H_OK.
rewrite -> H_OK.
reflexivity.
```

(iii) The third subgoal:

```
============================
fetch_decode_execute_loop
  (bci :: bcis') ds = KO s
```

Similarly, this subgoal is a direct corollary of (Theorem 7: there is at most one decode execute function) and (Theorem 9: decode execute satisfies the specification of decode execute), after which we can rewrite with the hypothesis corresponding to thsi case, `H_KO`. Thus the steps are as follows:

```
intros bci bcis' ds s.
intro H_KO.
rewrite -> (fold_unfold_fetch_decode_execute_loop_cons bci bcis' ds).
rewrite -> (there_is_at_most_one_decode_execute_function decode_execute'
↪  decode_execute H_decode_execute
↪  decode_execute_satisfies_the_specification_of_decode_execute bci ds) in H_KO.
rewrite -> H_KO.
reflexivity.
```

5 Task 4: Theorem 13: about concatenation of two lists of byte code instructions

An overview for this proof is shown in Figure 4 below.

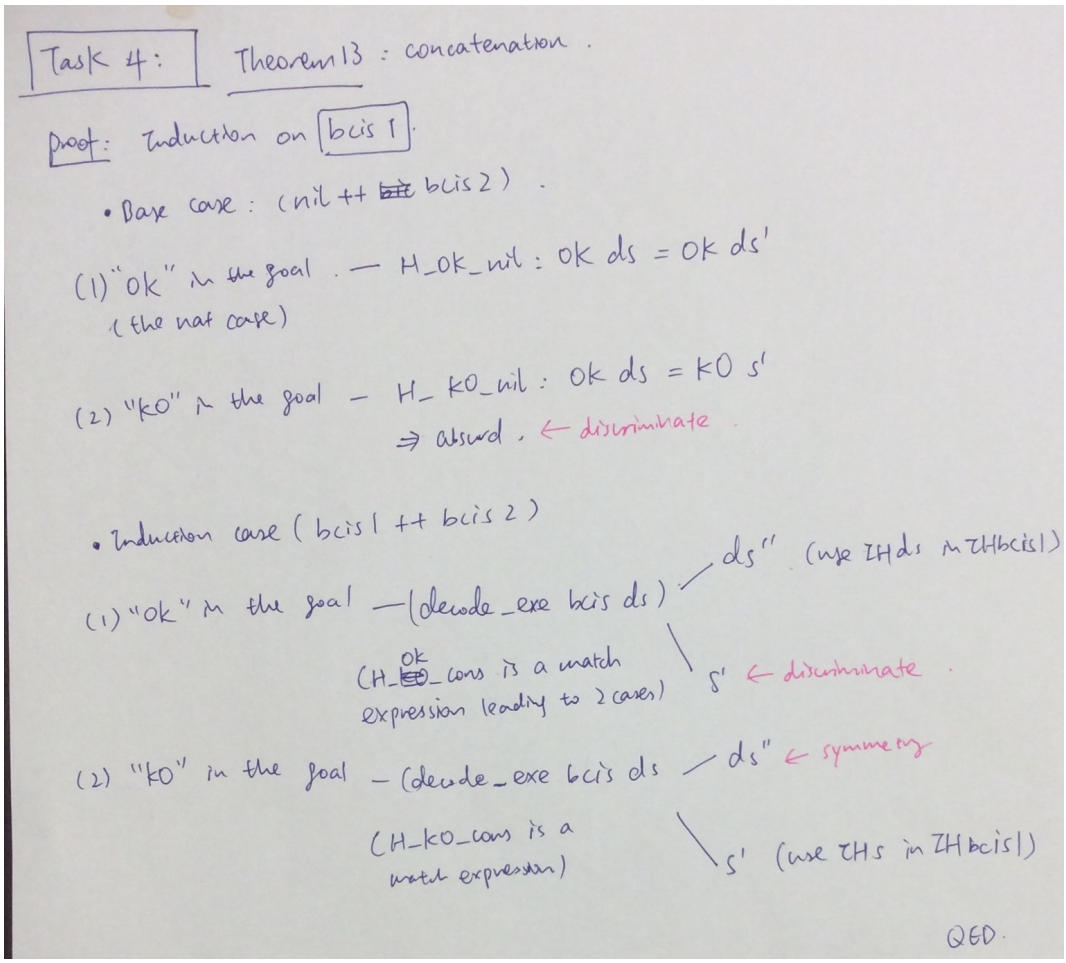


Figure 4: Figure 4

We first state the fold-unfold lemmas for append as a preliminary step to the proof. The next step is to formalize the theorem statement: Prove that for any lists of byte-code instructions bcis1 and bcis2, and for any data stack ds, executing the concatenation of bcis1 and bcis2 (i.e., bcis1 ++ bcis2) with ds gives the same result as (1) executing bcis1 with ds, and then (2) executing bcis2 with the resulting data stack, if there exists one.

Since executing any list has two cases: the nat case (`OK ds`) and the msg case (`KO s`), the above statement is equivalent to satisfying both of these conditions:

- The nat case: executing the concatenated list (bcis1 ++ bcis2) with `fetch_decode_execute_loop` gives the same nat value as executing bcis1 with ds first (and giving us `OK ds'`) and then executing bcis2 with ds'

- The msg case: if executing bcis1 with ds first gives us `KO s'`, we've already encountered error in the first list. And so executing the concatenated list (bcis1 ++ bcis2) with `fetch_decode_execute_loop` returns `KO s'` as well.

Thus we have the formal statement for this theorem:

```
Theorem concatenation :
  forall (bcis1 bcis2 : list byte_code_instruction)
         (ds: data_stack) ,
  (forall (ds': data_stack),
    (fetch_decode_execute_loop bcis1 ds) = OK ds'
    ->
    (fetch_decode_execute_loop bcis2 ds') = (fetch_decode_execute_loop (bcis1 ++ bcis2)
    ↪  ds))
  /\
  (forall (s' : string),
      (fetch_decode_execute_loop bcis1 ds) = KO s'
      ->
      KO s' = (fetch_decode_execute_loop (bcis1 ++ bcis2) ds)).
```

Having formalized the statement of the theorem, we then prove by induction on `bcis1` with

```
induction bcis1 as [ | bci bcis1' IHbcis1 ].
```

This will give us the base case and the induction case. Due to the conjunction, within each of the two cases, there are two subcases, one for OK (the nat case, when the result of execution is a natural number) and one for KO (the msg case, when the result of execution is an error message). We focus on each of the cases and prove as follows:

(i) Base case:

- OK (the nat case):
  This subgoal reads:

```
1 subgoal (ID 115)

  bcis2 : list
            byte_code_instruction
  ds : data_stack
  ============================
  forall ds' : data_stack,
  fetch_decode_execute_loop nil ds =
  OK ds' ->
  fetch_decode_execute_loop bcis2
    ds' =
  fetch_decode_execute_loop
    (nil ++ bcis2) ds
```

  We first intro `ds'` and `H_OK_nil`the hypothesis for this case (intuitively named). Now the goal becomes:

```
bcis2 : list
          byte_code_instruction
ds, ds' : data_stack
H_OK_nil : OK ds = OK ds'
============================
fetch_decode_execute_loop bcis2
  ds' =
fetch_decode_execute_loop bcis2
  ds
```

This prompts us to use the injection tactic `injection H_OK_nil` and get:

```
bcis2 : list
          byte_code_instruction
ds, ds' : data_stack
H_OK_nil : OK ds = OK ds'
============================
ds = ds' ->
fetch_decode_execute_loop bcis2
  ds' =
fetch_decode_execute_loop bcis2
  ds
```

In the nil case, the hypothesis `H_equal := ds = ds'` is true. Thus we are done after the following steps:

```
intro H_equal.
rewrite -> (H_equal).
reflexivity.
```

- KO (the msg case):

  This subgoal reads:

```
1 subgoal (ID 116)

bcis2 : list
          byte_code_instruction
ds : data_stack
============================
forall s' : string,
fetch_decode_execute_loop nil ds =
KO s' ->
KO s' =
fetch_decode_execute_loop
  (nil ++ bcis2) ds
```

  After rewriting with the fold-unfold lemma for fetch decode execute, we get:

```
1 subgoal (ID 129)

bcis2 : list
```

```
          byte_code_instruction
ds : data_stack
s' : string
H_KO_nil : OK ds = KO s'
============================
KO s' =
fetch_decode_execute_loop
  (nil ++ bcis2) ds
```

Observe that the hypothesis for this case `H_KO_nil : OK ds = KO s'` is always false because the nat value is never equal to the error message. Thus, we use the discriminate tactic.

(ii) Induction case:

- OK (the nat case):
  This subgoal reads:

```
============================
forall ds' : data_stack,
fetch_decode_execute_loop
  (bci :: bcis1') ds =
OK ds' ->
fetch_decode_execute_loop bcis2
  ds' =
fetch_decode_execute_loop
  ((bci :: bcis1') ++ bcis2) ds
```

  After rewriting, we get this:

```
ds, ds' : data_stack
  H_OK_cons : match
                decode_execute bci
                  ds
              with
              | OK ds' =>
                  fetch_decode_execute_loop
                    bcis1' ds'
              | KO s => KO s
              end = OK ds'
============================
fetch_decode_execute_loop bcis2
  ds' =
match decode_execute bci ds with
| OK ds'0 =>
    fetch_decode_execute_loop
      (bcis1' ++ bcis2) ds'0
| KO s => KO s
end
```

This means we have two cases and we use destruct and name each case ds" (the case where executing bci gives a nat) and s' (the case where executing bci gives an error message).

(a) executing bci gives a nat
Focusing on this goal, we have:

```
    bcis2 : list
            byte_code_instruction
ds, ds', ds'' : data_stack
H_OK_cons : fetch_decode_execute_loop
              bcis1' ds'' =
            OK ds'
============================
fetch_decode_execute_loop bcis2
  ds' =
fetch_decode_execute_loop
  (bcis1' ++ bcis2) ds''
```

This prompts us to apply the nat part of the induction hypothesis. Thus, we destruct IHbcis1 and name the nat part as IHds. Now we can rewrite the goal using IHds using `rewrite -> (IHds ds' H_OK_cons)`. And the goal now becomes:

```
    ============================
fetch_decode_execute_loop
  (bcis1' ++ bcis2) ds'' =
fetch_decode_execute_loop
  (bcis1' ++ bcis2) ds''
```

which allows us to use the reflexivity tactic to conclude this case.

(b) executing bci gives an error message Observe that `H_OK_cons : KO s' = OK ds'` is, once again, absurd. Thus we use the discriminate tactic.

- KO (the msg case):
This subgoal reads:

```
============================
forall ds' : data_stack,
fetch_decode_execute_loop
  (bci :: bcis1') ds =
OK ds' ->
fetch_decode_execute_loop bcis2
  ds' =
fetch_decode_execute_loop
  ((bci :: bcis1') ++ bcis2) ds
```

This is proved exactly the same way as the previous case (KO, cons), except for the last part where we take the msg part of the initial hypothesis IHbcis1 and name it as IHs.

Similarly, we have two cases and we use destruct and name each case ds" (the case where executing bci gives a nat) and s' (the case where executing bci gives an error message).

(a) executing bci gives an error message:

```
        destruct (IHbcis1 bcis2 ds'') as [_ IHs].
        rewrite -> (IHs s' H_KO_cons).
        reflexivity.
```

(b) executing bci gives a nat:

```
    symmetry.
    exact H_KO_cons.
```

6 Task 5: run

6.1 Introduction

In this task, we first proved that the given specification of `run` is not ambiguous. We then defined the `run` function and showed that it satisfies the given specification.

The `run` function is non-recursive. Thus the proofs in this section are proofs by cases. Common to all subtasks of task 5, there are 4 cases to consider:

(i) when there is no result on the data stack

(ii) when the result is a natural number

(iii) when there are too many results on the data stack

(iv) when the result is some error message

6.2 5a: Theorem 14: There is at most one run function.

This is a proof by case and does not involve induction. An overview of the proof is given in Figure 5.

We specify the cases with:

```
case (fetch_decode_execute_loop bcis nil) as [[ | n [ | n' ds'' ]] | s] eqn:H_bcis.
```

which gives us the subgoals that read as follows:

```
============================
run1 (Target_program bcis) = run2 (Target_program bcis)

subgoal 2 (ID 177) is:
 run1 (Target_program bcis) = run2 (Target_program bcis)
subgoal 3 (ID 180) is:
 run1 (Target_program bcis) = run2 (Target_program bcis)
subgoal 4 (ID 182) is:
 run1 (Target_program bcis) = run2 (Target_program bcis)
```

To be specific, the 4 cases are:

(i) when there is no result on the data stack, we rewrite using `H_run1_nil` and `H_run2_nil`:

```
rewrite -> (H_run1_nil bcis H_bcis).
rewrite -> (H_run2_nil bcis H_bcis).
```

after which the goal window reads:

```
============================
Expressible_msg "no result on the data stack" =
Expressible_msg "no result on the data stack"
```

and we can then use the reflexivity tactic to complete this subgoal.

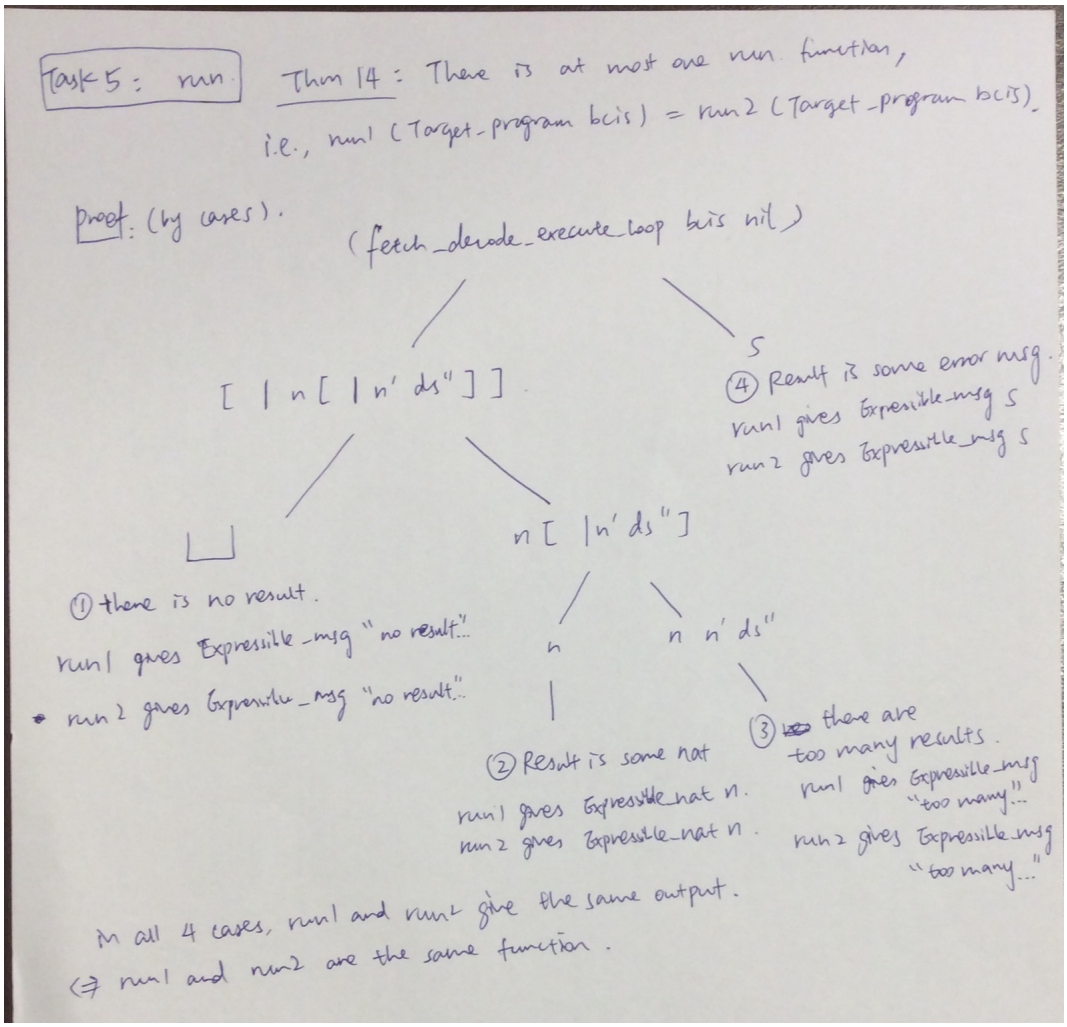


Figure 5: Figure 5

(ii) when the result is a natural number, we rewrite using `H_run1_1` and `H_run2_1`:

```
rewrite -> (H_run1_1 bcis n H_bcis).
rewrite -> (H_run2_1 bcis n H_bcis).
```

after which the goal window reads:

```
============================
Expressible_nat n = Expressible_nat n
```

and we can then use the reflexivity tactic to complete this subgoal.

(iii) when there are too many results on the data stack, we rewrite using `H_run1_2` and `H_run2_2`:

```
rewrite -> (H_run1_2 bcis n n' ds'' H_bcis).
rewrite -> (H_run2_2 bcis n n' ds'' H_bcis).
```

after which the goal window reads:

```
============================
Expressible_msg "too many results on the data stack" =
Expressible_msg "too many results on the data stack"
```

and we can then use the reflexivity tactic to complete this subgoal.

(iv) when the result is some error message, we rewrite using `H_run1_error` and `H_run2_error`:

```
rewrite -> (H_run1_error bcis s H_bcis).
rewrite -> (H_run2_error bcis s H_bcis).
```

after which the goal window reads:

```
============================
Expressible_msg s = Expressible_msg s
```

and we can then use the reflexivity tactic to complete this subgoal.

Having proved all subgoals, we complete the proof.

6.3 5b: Definition 15: run function

```
Definition run (tp : target_program) : expressible_value :=
  match tp with
  | Target_program bcis =>
    match (fetch_decode_execute_loop bcis nil) with
    | OK nil =>
      Expressible_msg "no result on the data stack"
    | OK (n :: nil) =>
      Expressible_nat n
```

```
  | OK (n :: n' :: ds'') =>
    Expressible_msg "too many results on the data stack"
  | KO s =>
    Expressible_msg s
  end
end.
```

Note that this definition mirrors the structure of the specification of run as well as the proof in 5a, where we split into 4 cases, namely:

(i) when there is no result on the data stack

(ii) when the result is a natural number

(iii) when there are too many results on the data stack

(iv) when the result is some error message

6.4 5c: Theorem 16: run satisfies the specification of run.

The proof structure once again follows from what we have stated in the introduction of this task. In each case, we apply the previously proved Theorem 14 about the non-ambiguity of specification of run:

(i) when there is no result on the data stack

The goal window reads:

```
1 subgoal (ID 127)

fedl : list byte_code_instruction ->
       data_stack -> result_of_decoding_and_execution
H_fedl : specification_of_fetch_decode_execute_loop fedl
============================
forall bcis : list byte_code_instruction,
fedl bcis nil = OK nil ->
match fetch_decode_execute_loop bcis nil with
| OK nil => Expressible_msg "no result on the data stack"
| OK (n :: nil) => Expressible_nat n
| OK (n :: _ :: _) =>
    Expressible_msg "too many results on the data stack"
| KO s => Expressible_msg s
end = Expressible_msg "no result on the data stack"
```

For this subgoal, we apply Theorem 14 in the hypothesis for the case where there is no result, `H_bcis_nil`:

```
intros bcis H_bcis_nil.
rewrite <- (there_is_at_most_one_fetch_decode_execute_loop_function
↪  fetch_decode_execute_loop fedl
↪  fetch_decode_execute_loop_satisfies_the_specification H_fedl bcis nil) in
↪  H_bcis_nil.
rewrite -> H_bcis_nil.
reflexivity.
```

(ii) when the result is a natural number

The goal window reads:

```
1 subgoal (ID 139)

  fedl : list byte_code_instruction ->
         data_stack -> result_of_decoding_and_execution
  H_fedl : specification_of_fetch_decode_execute_loop fedl
  ============================
  forall (bcis : list byte_code_instruction) (n : nat),
  fedl bcis nil = OK (n :: nil) ->
  match fetch_decode_execute_loop bcis nil with
  | OK nil => Expressible_msg "no result on the data stack"
  | OK (n0 :: nil) => Expressible_nat n0
  | OK (n0 :: _ :: _) =>
      Expressible_msg "too many results on the data stack"
  | KO s => Expressible_msg s
  end = Expressible_nat n
```

For this subgoal, we apply Theorem 14 in the hypothesis for the case where there is exactly one result of type nat, H_bcis_1:

```
intros bcis n H_bcis_1.
rewrite <- (there_is_at_most_one_fetch_decode_execute_loop_function
↪  fetch_decode_execute_loop fedl
↪  fetch_decode_execute_loop_satisfies_the_specification H_fedl bcis nil) in
↪  H_bcis_1.
rewrite -> H_bcis_1.
reflexivity.
```

(iii) when there are too many results on the data stack

The goal window reads:

```
1 subgoal (ID 152)

  fedl : list byte_code_instruction ->
         data_stack -> result_of_decoding_and_execution
  H_fedl : specification_of_fetch_decode_execute_loop fedl
  ============================
```

```
forall (bcis : list byte_code_instruction) (n n' : nat)
  (ds'' : data_stack),
fedl bcis nil = OK (n :: n' :: ds'') ->
match fetch_decode_execute_loop bcis nil with
| OK nil => Expressible_msg "no result on the data stack"
| OK (n0 :: nil) => Expressible_nat n0
| OK (n0 :: _ :: _) =>
    Expressible_msg "too many results on the data stack"
| KO s => Expressible_msg s
end = Expressible_msg "too many results on the data stack"
```

For this subgoal, we apply Theorem 14 in the hypothesis for the case where there are too many results on the data stack, H_bcis_2:

```
intros bcis n n' ds'' H_bcis_2.
rewrite <- (there_is_at_most_one_fetch_decode_execute_loop_function
↪  fetch_decode_execute_loop fedl
↪  fetch_decode_execute_loop_satisfies_the_specification H_fedl bcis nil)
↪  in H_bcis_2.
rewrite -> H_bcis_2.
reflexivity.
```

(iv) when the result is some error message The goal window reads:

```
1 subgoal (ID 153)

fedl : list byte_code_instruction ->
       data_stack -> result_of_decoding_and_execution
H_fedl : specification_of_fetch_decode_execute_loop fedl
============================
forall (bcis : list byte_code_instruction) (s : string),
fedl bcis nil = KO s ->
match fetch_decode_execute_loop bcis nil with
| OK nil => Expressible_msg "no result on the data stack"
| OK (n :: nil) => Expressible_nat n
| OK (n :: _ :: _) =>
    Expressible_msg "too many results on the data stack"
| KO s0 => Expressible_msg s0
end = Expressible_msg s
```

For this subgoal, we apply Theorem 14 in the hypothesis for the case where the result is some error message , H_bcis_error:

```
intros bcis s H_error.
rewrite <- (there_is_at_most_one_fetch_decode_execute_loop_function
↪  fetch_decode_execute_loop fedl
↪  fetch_decode_execute_loop_satisfies_the_specification H_fedl bcis nil)
↪  in H_error.
```

```
        rewrite -> H_error.
        reflexivity.
```

Having proved all subgoals, we complete the proof.

7 Task 6: compile aux

7.1 Introduction

In this task, we first proved that the given specification of`compile_aux` is not ambiguous. We then defined the `compile_aux` function and showed that it satisfies the given specification.

The `compile_aux` function is a recursive function. Thus, the proofs in this section are proofs by induction (on ae) and we need to consider separately the base case and the induction case.

(i) Base case: (Literal n)

(ii) Induction case:

(a) (Plus ae1 ae2)

(b) (Minus ae1 ae2)

7.2 6a: Theorem 17: There is at most one compile aux function.

This is a proof by induction on ae. Thus, we start by intros, destruct the hypothesis and then state the base case and induction case for the induction as follows:

```
intros compile_aux_1 compile_aux_2.
intros H1 H2 ae.
destruct H1 as [H_lit_1 [H_plus_1 H_minus_1]].
destruct H2 as [H_lit_2 [H_plus_2 H_minus_2]].
induction ae as [n | ae1 IHae1 ae2 IHae2 | ae1 IHae1 ae2 IHae2].
```

which gives us the subgoals that read as follows:

```
3 subgoals (ID 175)

  compile_aux_1,
  compile_aux_2 : arithmetic_expression -> list byte_code_instruction
  H_lit_1 : forall n : nat, compile_aux_1 (Literal n) = PUSH n :: nil
  H_plus_1 : forall ae1 ae2 : arithmetic_expression,
             compile_aux_1 (Plus ae1 ae2) =
             compile_aux_1 ae1 ++ compile_aux_1 ae2 ++ ADD :: nil
  H_minus_1 : forall ae1 ae2 : arithmetic_expression,
              compile_aux_1 (Minus ae1 ae2) =
              compile_aux_1 ae1 ++ compile_aux_1 ae2 ++ SUB :: nil
  H_lit_2 : forall n : nat, compile_aux_2 (Literal n) = PUSH n :: nil
  H_plus_2 : forall ae1 ae2 : arithmetic_expression,
             compile_aux_2 (Plus ae1 ae2) =
             compile_aux_2 ae1 ++ compile_aux_2 ae2 ++ ADD :: nil
  H_minus_2 : forall ae1 ae2 : arithmetic_expression,
              compile_aux_2 (Minus ae1 ae2) =
              compile_aux_2 ae1 ++ compile_aux_2 ae2 ++ SUB :: nil
  n : nat
```

```
  ============================
  compile_aux_1 (Literal n) = compile_aux_2 (Literal n)

subgoal 2 (ID 180) is:
 compile_aux_1 (Plus ae1 ae2) = compile_aux_2 (Plus ae1 ae2)
subgoal 3 (ID 185) is:
 compile_aux_1 (Minus ae1 ae2) = compile_aux_2 (Minus ae1 ae2)
```

We focus on each of the 3 subgoals and prove one by one as follows:

(i) Base case: (Literal n).

we rewrite using `H_lit_1` and `H_lit_2`:

```
rewrite (H_lit_1 n).
rewrite (H_lit_2 n).
```

after which the goal window reads:

```
  ============================
  PUSH n :: nil = PUSH n :: nil
```

and we can then use the reflexivity tactic to complete this subgoal.

(ii) Induction case:

(a) (Plus ae1 ae2), we can rewrite using `H_plus_1` and `H_plus_2` and the induction hypotheses about the two arithmetic expressions IHae1 and IHae2:

```
rewrite (H_plus_1 ae1 ae2).
rewrite (H_plus_2 ae1 ae2).
rewrite IHae1.
rewrite IHae2.
```

after which the goal window reads:

```
  ============================
 compile_aux_2 ae1 ++ compile_aux_2 ae2 ++ ADD :: nil =
 compile_aux_2 ae1 ++ compile_aux_2 ae2 ++ ADD :: nil
```

and we can then use the reflexivity tactic to complete this subgoal.

(b) (Minus ae1 ae2), similar to the plus case:

```
rewrite (H_minus_1 ae1 ae2).
rewrite (H_minus_2 ae1 ae2).
rewrite IHae1.
rewrite IHae2.
```

after which the goal window reads:

```
  ============================
  compile_aux_2 ae1 ++ compile_aux_2 ae2 ++ SUB :: nil =
  compile_aux_2 ae1 ++ compile_aux_2 ae2 ++ SUB :: nil
```

and we can then use the reflexivity tactic to complete this subgoal.

Having proved all subgoals, we complete the proof.

7.3 6b: Definition 18: compile aux function

To eventually define the compile function (in task 7), we first define the helper function `compile_aux` to contain the inductive part. And again, we state its associate fold-unfold lemmas.

```
Fixpoint compile_aux (ae : arithmetic_expression) : list byte_code_instruction :=
  match ae with
  | Literal n => PUSH n :: nil
  | Plus ae1 ae2 => (compile_aux ae1) ++ (compile_aux ae2) ++ (ADD :: nil)
  | Minus ae1 ae2 => (compile_aux ae1) ++ (compile_aux ae2) ++ (SUB :: nil)
  end.
```

Note that this definition mirrors the structure of the specification of run as well as the proof in 6a, where we split into 3 cases, namely:

(i) Base case: (Literal n)

(ii) Induction case:

  (a) (Plus ae1 ae2)

  (b) (Minus ae1 ae2)

7.4 6c: Theorem 19: compile aux satisfies the specification.

This is a direct proof. We first unfold the specification and the goal window reads:

```
1 subgoal (ID 165)

  ============================
  (forall n : nat, compile_aux (Literal n) = PUSH n :: nil) /\
  (forall ae1 ae2 : arithmetic_expression,
   compile_aux (Plus ae1 ae2) =
   compile_aux ae1 ++ compile_aux ae2 ++ ADD :: nil) /\
  (forall ae1 ae2 : arithmetic_expression,
   compile_aux (Minus ae1 ae2) =
   compile_aux ae1 ++ compile_aux ae2 ++ SUB :: nil)
```

The following proof is contingent on the parallel between fold-unfold lemmas and the components of the specification. Concretely:

```
split.
- exact fold_unfold_compile_aux_lit.
- split.
  + exact fold_unfold_compile_aux_plus.
  + exact fold_unfold_compile_aux_minus.
```

With this we complete the proof.

8 Task 7: compile

8.1 Introduction

In this task, we proved that the given specification of`compile` is not ambiguous. We then defined the `compile` function and showed that it satisfies the given specification.

The `compile` function is a non-recursive function.

(i) Base case: (Literal n)

(ii) Induction case:

(a) (Plus ae1 ae2)

(b) (Minus ae1 ae2)

8.2 7a: Theorem 20: There is at most one compile function.

This theorem is a direct corollary of Theorem 19: compile aux satisfies the specification. Thus, the proof is simply fitting the right argument into Theorem 19: compile aux satisfies the specification and rewrite. Concretely, the steps are shown below:

```
intros compile1 compile2.
unfold specification_of_compile.
intros H_compile1 H_compile2 ae.
rewrite -> ( H_compile1 compile_aux compile_aux_satisfies_the_specification ae).
rewrite -> (H_compile2 compile_aux compile_aux_satisfies_the_specification ae).
reflexivity.
```

8.3 7b: Definition 21: compile function

To define the compile function, we instantiate the predefined the helper function `compile_aux` with ae (an arithmetic expression). And again, we state its associate fold-unfold lemmas.

```
Definition compile (sp : source_program) : target_program :=
  match sp with
  | Source_program ae =>
    Target_program (compile_aux ae)
  end.
```

8.4 7c: Theorem 22: compile satisfies the specification of compile.

This proof is a direct corollary of Theorem 20 there is at most one compile function. The steps are shown below:

```
unfold specification_of_compile, compile.
intros compile_aux' S_compile_aux' ae.
rewrite -> (there_is_at_most_one_compile_aux compile_aux compile_aux'
↪  compile_aux_satisfies_the_specification S_compile_aux' ae).
reflexivity.
Qed.
```

9 Task 8: compiler with accumulator, compile alt

9.1 Introduction

In this task, we defined the `compile_alt` function which is the accumulator version of the compiler and showed that it satisfies the given specification of `compile`.

9.2 8a: Definition 23: compile alt

To define `compile_alt`, the accumulator version of the `compile` function, we first define its recursive part, `compile_aux_alt`. Recall that we have implemented the compiler with accumulator back in Intro to CS:

```
let compile_v2_ltor (Source_program e) =
  let rec translate e a =
    match e with
    | Literal n ->
       [Push n] @ a
    | Plus (e1, e2) ->
       (let a = [Add] @ a
        in translate e1 (translate e2 a))
    | Minus (e1, e2) ->
       (let a = [Sub] @ a
        in translate e1 (translate e2 a))
    | Quotient (e1, e2) ->
       (let a = [Quo] @ a
        in translate e1 (translate e2 a))
    | Remainder (e1, e2) ->
       (let a = [Rem] @ a
        in translate e1 (translate e2 a))
  in Target_program(translate e []);;
```

Using the same approach, the inductive specification for `compile_aux_alt` is as follows:

- Base case: Whenever we encounters `Literal n`, we yield a list by adding the BCI `Push n` into the existing list of BCIs and returns the updated accumulator. This is our base case.
- Induction step:
  - (i) Plus ae1 ae2
    Whenever we encounters `Plus ae1 ae2`, we yield a list by adding the BCI `ADD` and the respective compiled results of ae1 and ae2 into the existing list of BCIs and returns the updated accumulator.
  - (ii) Minus ae1 ae2
    Whenever we encounters `Minus ae1 ae2`, we yield a list by adding the BCI `SUB` and the respective compiled results of ae1 and ae2 into the existing list of BCIs and returns the updated accumulator.

Given the above inductive specification, our implementation is as follows.

```
Fixpoint compile_aux_alt (ae : arithmetic_expression) (a : list byte_code_instruction) :
↪  list byte_code_instruction :=
  match ae with
  | Literal n => PUSH n :: a
  | Plus ae1 ae2 => compile_aux_alt ae1 (compile_aux_alt ae2 (ADD :: a))
  | Minus ae1 ae2 => compile_aux_alt ae1 (compile_aux_alt ae2 (SUB :: a))
  end.
```

As a good habit, after defining the inductive function `compile_aux_alt`, we immediately state its associated fold-unfold lemmas:

```
Lemma fold_unfold_compile_aux_alt_lit :
  forall (n : nat)
         (a : list byte_code_instruction),
    compile_aux_alt (Literal n) a = PUSH n :: a.
Proof.
  fold_unfold_tactic compile_aux_alt.
Qed.

Lemma fold_unfold_compile_aux_alt_plus :
  forall (ae1 ae2 : arithmetic_expression)
         (a : list byte_code_instruction),
    compile_aux_alt (Plus ae1 ae2) a = compile_aux_alt ae1 (compile_aux_alt ae2 (ADD ::
    ↪  a)).
Proof.
  fold_unfold_tactic compile_aux_alt.
Qed.

Lemma fold_unfold_compile_aux_alt_minus :
  forall (ae1 ae2 : arithmetic_expression)
         (a : list byte_code_instruction),
    compile_aux_alt (Minus ae1 ae2) a = compile_aux_alt ae1 (compile_aux_alt ae2 (SUB ::
    ↪  a)).
Proof.
  fold_unfold_tactic compile_aux_alt.
Qed.
```

Now with `compile_aux_alt`, we can easily define `compile_alt` since the latter is simply instantiating the `compile_aux_alt` function with `ae`, and the nil list as the initial value for the accumulator.

```
Definition compile_alt (sp : source_program) : target_program :=
  match sp with
  | Source_program ae =>
    Target_program (compile_aux_alt ae nil)
  end.
```

9.3 8b: Theorem/Lemma 24: compile aux alt is equivalent to compile aux

This is an induction proof on ae:

```
induction ae as [ n | ae1 IHae1 ae2 IHae2 | ae1 IHae1 ae2 IHae2 ]; intro a.
```

which gives us 3 subgoals as follows:

```
3 subgoals (ID 203)

n : nat
a : list byte_code_instruction
============================
compile_aux_alt (Literal n) a =
compile_aux (Literal n) ++ a

subgoal 2 (ID 204) is:
 compile_aux_alt (Plus ae1 ae2) a =
 compile_aux (Plus ae1 ae2) ++ a
subgoal 3 (ID 205) is:
 compile_aux_alt (Minus ae1 ae2) a =
 compile_aux (Minus ae1 ae2) ++ a
```

This parallels the 3 cases for `compile_aux_alt` in Definition 23 for compiler with accumulator. We focus on each subgoal and prove one by one:

(i) base case: (Literal n)

This subgoal is:

```
n : nat
a : list byte_code_instruction
============================
compile_aux_alt (Literal n) a =
compile_aux (Literal n) ++ a
```

For this, we can use the fold-unfold lemma for the literal case first to get to:

```
n : nat
a : list byte_code_instruction
============================
PUSH n :: a =
(PUSH n :: nil) ++ a
```

which suggests the use of fold-unfold lemma for append (++):

```
rewrite -> (fold_unfold_append_cons (PUSH n) nil a).
rewrite -> fold_unfold_append_nil.
```

and we can then use the reflexivity tactic to complete this subgoal.

(ii) induction case:

(a) Plus ae1 ae2

The second subgoal is:

```
1 subgoal (ID 204)

  ae1, ae2 : arithmetic_expression
  IHae1 : forall
            a : list
                  byte_code_instruction,
          compile_aux_alt ae1 a =
          compile_aux ae1 ++ a
  IHae2 : forall
            a : list
                  byte_code_instruction,
          compile_aux_alt ae2 a =
          compile_aux ae2 ++ a
  a : list byte_code_instruction
  ============================
  compile_aux_alt (Plus ae1 ae2) a =
  compile_aux (Plus ae1 ae2) ++ a
```

Using fold-unfold lemma, we get:

```
  ============================
  compile_aux_alt ae1
    (compile_aux_alt ae2
       (ADD :: a)) =
  (compile_aux ae1 ++
   compile_aux ae2 ++ ADD :: nil) ++
  a
```

After using `Search ((_) ++ _ = _ ++ _)`, we reach the key realization to proceed from here: that append is associative, i.e.,

```
     ============================
  compile_aux_alt ae1
    (compile_aux_alt ae2
       (ADD :: a)) =
  (compile_aux ae1 ++
   compile_aux ae2 ++ ADD :: nil) ++
  a
```

Thus rewriting with the arithmetic property `app_assoc`,

(b) when there are too many results on the data stack, we get:

```
  ============================
  compile_aux_alt ae1
    (compile_aux_alt ae2
```

```
      (ADD :: a)) =
  compile_aux ae1 ++
  compile_aux ae2 ++
  (ADD :: nil) ++ a
```

which prompts us to use fold-unfold lemma for append. Indeed the goal now becomes:

```
1 subgoal (ID 220)

  ae1, ae2 : arithmetic_expression
  IHae1 : forall
            a : list
                  byte_code_instruction,
          compile_aux_alt ae1 a =
          compile_aux ae1 ++ a
  IHae2 : forall
            a : list
                  byte_code_instruction,
          compile_aux_alt ae2 a =
          compile_aux ae2 ++ a
  a : list byte_code_instruction
  ============================
  compile_aux_alt ae1
    (compile_aux_alt ae2
       (ADD :: a)) =
  compile_aux ae1 ++
  compile_aux ae2 ++ ADD :: a
```

Now, after rewriting using the induction hypothesis for ae2 (IHae2) and then the induction hypothesis for ae1 (IHae1), we can then use the reflexivity tactic to complete this subgoal.

(c) Minus ae1 ae2

This subgoal is:

```
1 subgoal (ID 205)

  ae1, ae2 : arithmetic_expression
  IHae1 : forall
            a : list
                  byte_code_instruction,
          compile_aux_alt ae1 a =
          compile_aux ae1 ++ a
  IHae2 : forall
            a : list
                  byte_code_instruction,
          compile_aux_alt ae2 a =
          compile_aux ae2 ++ a
  a : list byte_code_instruction
  ============================
```

```
  compile_aux_alt (Minus ae1 ae2)
    a =
  compile_aux (Minus ae1 ae2) ++ a
```

Since it has the exact same structure as the plus case, the steps are identical except we are using the fold-unfold lemma for minus case instead of plus case. The specific steps are shown:

```
rewrite -> fold_unfold_compile_aux_minus.
   rewrite -> fold_unfold_compile_aux_alt_minus.
   rewrite <- app_assoc.
   rewrite <- app_assoc.
   rewrite -> fold_unfold_append_cons.
   rewrite -> fold_unfold_append_nil.
   rewrite -> IHae2.
   rewrite -> IHae1.
   reflexivity.
```

Having proved all subgoals, we complete the proof.

9.4 8c: Theorem 25: compile alt satisfies the specification.

This is a direct corollary from Theorem/Lemma 24 compile aux alt is equivalent to compile aux and Theorem 17: There is at most one compile aux function. Thus, we can do the following steps:

```
  unfold specification_of_compile, compile_alt.
intros compile_aux_alt' S_compile_aux_alt' ae.
unfold compile_aux_alt.
rewrite -> compile_aux_alt_is_equivalent_to_compile_aux.
rewrite <- (there_is_at_most_one_compile_aux compile_aux compile_aux_alt'
↪  compile_aux_satisfies_the_specification S_compile_aux_alt' ae).
```

and the goal now becomes:

```
1 subgoal (ID 194)

compile_aux_alt' : arithmetic_expression ->
                   list
                    byte_code_instruction
S_compile_aux_alt' :
specification_of_compile_aux
  compile_aux_alt'
ae : arithmetic_expression
============================
Target_program
  (compile_aux ae ++ nil) =
Target_program (compile_aux ae)
```

Now, we turn to Search again and `Search (_ ++ nil = _).` prompts us to use the property `app_nil_r`, where

```
app_nil_r:
forall [A : Type] (l : list A),
l ++ nil = l
```

using , and then we get:

```
  ============================
Target_program (compile_aux ae) =
Target_program (compile_aux ae)
```

and with that we can use reflexivity tactic to complete this subgoal.

9.5 8b: Theorem/Lemma 26: compile is equivalent to compile alt.

Theorem 26 is a direct corollary of Theorem 25: compile alt satisfies the specification and Theorem 20: There is at most one compile function. And the proof is just passing the right arguments to those two theorems:

```
exact (there_is_at_most_one_compile_function compile compile_alt
↪  compile_satisfies_the_specification compile_alt_satisfies_the_specification ae).
```

9.6 Some Concluding Remarks

Having proved that compiler with accumulator is equivalent to the regular compiler, we can use the two functions interchangeably. This is a nice result because now we can always replace the regular compiler with the more efficient compiler with accumulator without any concerns. Based on what we have learned from Intro to CS, we know that the accumulator version is more efficient because it bypasses the list concatenation (++) by using (::) instead. List concatenation used in the regular compiler (in Task 8) incurs a linear cost since it prepends its first argument onto its second by copying it.

10 Task 9: the commutative diagram

10.1 Introduction

In this task, we prove that interpreting any given expression gives the same result as compiling this expression and then running the resulting compiled program.

Recall that the miniproject for the three language processors in Intro to CS culminated into the commutativity test, where we implemented a unit-test function to verify that interpreting source programs yields the same result as compiling them and running the resulting target programs.

We first defined the commutative test for one source program where we generated random arithmetic expressions to make up the source program and used it to build our unit-test function. Concretely:

```
  let commutativity_test_one candidate_interpret candidate_compile candidate_run p =
candidate_interpret p = candidate_run (candidate_compile p);;
```

Thus, the full commutativity test was simply a repetition of the commutative test for one source program:

```
let commutativity_test candidate_interpret candidate_compile candidate_run =
   (* commutativity_test : (source_program -> expressible_value) ->
                           (source_program -> target_program) ->
                           (target_program -> expressible_value) ->
                           source_program ->
                           bool *)
  let b0 = let thunk = (fun () ->
               let p = Source_program (generate_random_arithmetic_expression 10)
               in commutativity_test_one candidate_interpret candidate_compile
               ↪  candidate_run p)
           in repeat 10 thunk
  in b0;;

let () = assert(commutativity_test interpret_ltor compile_v1_ltor run);;
```

However, we have noted in the Intro to CS project many times that testing is incomplete. In this case, the test cases that we randomly generated is not exhaustive of all the possible source programs, (and for this reason we can never conclude a "for all" statement just with a unit test function). In this task, though, we will finally be able to ***prove*** that interpreting source programs yields the same result as compiling them and running the resulting target programs. This will enable us to state with complete certainty that this result holds *for all* source programs.

10.2 Theorem 27: The main theorem for commutative diagram

With that, we formalize the main theorem (Theorem 27) to be proved:

```
Theorem the_commutative_diagram :
  forall sp : source_program,
    interpret sp = run (compile sp).
```

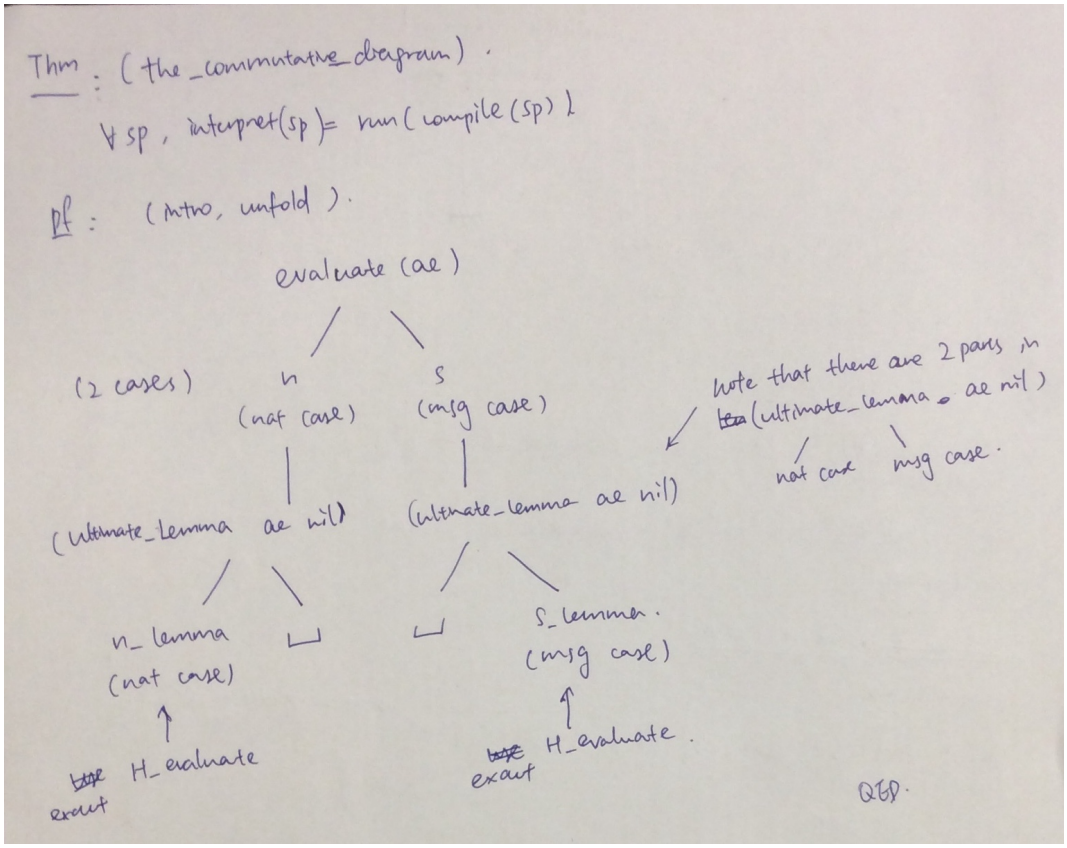


Figure 6: Figure 6

An overview for the proof of the main theorem is given in Figure 6. We start the proof by introducing the arithmetic expressions in the source program:

```
Proof.
  intros [ ae ].
```

The *goal* window reads:

```
1 subgoal (ID 205)

  ae : arithmetic_expression
  ============================
  interpret (Source_program ae) =
  run (compile (Source_program ae))
```

In the goal window, we have three non-recursive functions. Thus, we use unfold tactic:

```
  unfold interpret.
  unfold compile.
  unfold run.
```

The *goal* window now reads:

```
1 subgoal (ID 211)

  ae : arithmetic_expression
  ============================
  evaluate ae =
  match
    fetch_decode_execute_loop
      (compile_aux ae) nil
  with
  | OK nil =>
      Expressible_msg
        "no result on the data stack"
  | OK (n :: nil) =>
      Expressible_nat n
  | OK (n :: _ :: _) =>
      Expressible_msg
        "too many results on the data stack"
  | KO s => Expressible_msg s
  end
```

Now, we have a match expression and hence the two different cases to consider, which prompts us to use the destruct tactic. Here we also use `eqn` for naming equations for both cases respectively. (whenever using the destruct tactic, we can make the corresponding equation appear among the assumptions, by adding eqn on the right of that invocation.)

```
  destruct (evaluate ae) as [n | s] eqn : H_evaluate.
```

This will give us two cases in the goal window, (1) evaluating to a natural number and (2) evaluating to an error message:

```
2 subgoals (ID 224)

  Expressible_nat n =
  match
    fetch_decode_execute_loop
      (compile_aux ae) nil
  with
  | OK nil =>
      Expressible_msg
        "no result on the data stack"
  | OK (n0 :: nil) =>
      Expressible_nat n0
  | OK (n0 :: _ :: _) =>
      Expressible_msg
        "too many results on the data stack"
  | KO s => Expressible_msg s
  end

subgoal 2 (ID 226) is:
 Expressible_msg s =
 match
   fetch_decode_execute_loop
     (compile_aux ae) nil
 with
 | OK nil =>
     Expressible_msg
       "no result on the data stack"
 | OK (n :: nil) =>
     Expressible_nat n
 | OK (n :: _ :: _) =>
     Expressible_msg
       "too many results on the data stack"
 | KO s0 => Expressible_msg s0
 end
```

At this point, we have two methods of proving. The first method is to write two separate helper lemmas, one for the nat case (`evaluate_n_lemma`), the other for the error message case (`evaluate_error_lemma`). The second method is to write one single helper lemma (`evaluate_ultimate_lemma`) that can be used for both cases. In the following subsection I will focus on explaining `evaluate_ultimate_lemma` since it is essentially the combination of `evaluate_n_lemma` and `evaluate_error_lemma`.

10.3 Lemma 28: The evaluate ultimate lemma

First of all, we formalize the theorem:

```
Lemma evaluate_ultimate_lemma :
  forall (ae : arithmetic_expression)
         (ds : data_stack),
  (forall (n : nat),
      evaluate ae = Expressible_nat n -> fetch_decode_execute_loop (compile_aux ae) ds =
      ↪  OK (n :: ds)) /\
  (forall (s : string),
      evaluate ae = Expressible_msg s -> fetch_decode_execute_loop (compile_aux ae) ds =
      ↪  KO s).
```

An overview for the proof of the evaluate ultimate theorem is given in Figure 7. Observe the goal window below, we know that we need a proof by induction and that we have two cases (because of the conjunction) namely the nat case and the error message case:

```
1 subgoal (ID 199)

  ============================
  forall
    (ae : arithmetic_expression)
    (ds : data_stack),
  (forall n : nat,
   evaluate ae =
   Expressible_nat n ->
   fetch_decode_execute_loop
     (compile_aux ae) ds =
   OK (n :: ds)) /\
  (forall s : string,
   evaluate ae =
   Expressible_msg s ->
   fetch_decode_execute_loop
     (compile_aux ae) ds =
   KO s)
```

Thus, we introduce the induction hypotheses:

```
induction ae as [l | ae1 IHae1 ae2 IHae2 | ae1 IHae1 ae2 IHae2]; intro ds; split;
↪  intros output H_eval.
```

This gives us six subgoals as shown in the goal window:

```
6 subgoals (ID 232)

  l : nat
  ds : data_stack
  output : nat
  H_eval : evaluate (Literal l) = Expressible_nat output
  ============================
  fetch_decode_execute_loop (compile_aux (Literal l)) ds = OK (output :: ds)
```

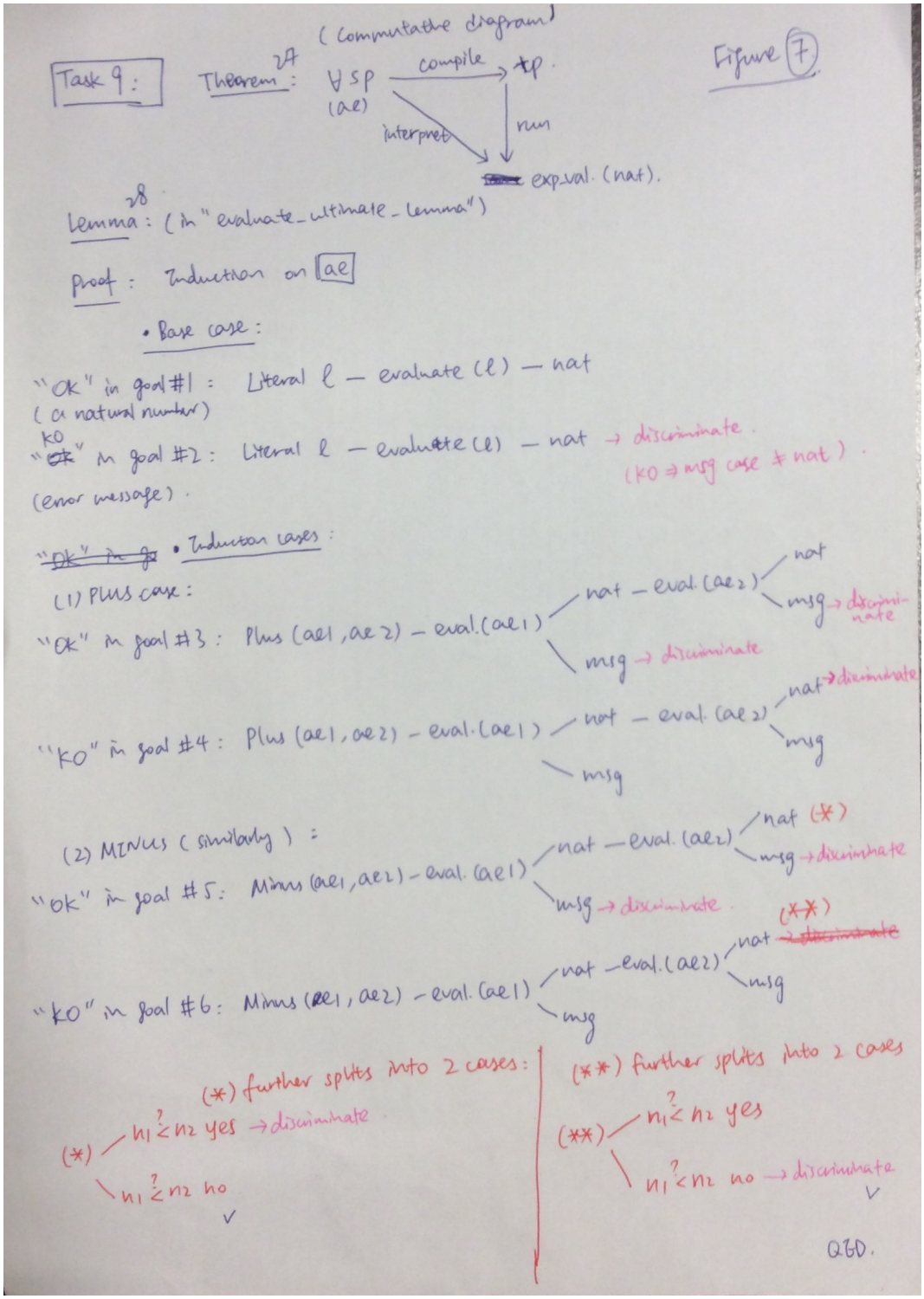


Figure 7: Figure 7

```
subgoal 2 (ID 234) is:
 fetch_decode_execute_loop (compile_aux (Literal l)) ds = KO output
subgoal 3 (ID 236) is:
 fetch_decode_execute_loop (compile_aux (Plus ae1 ae2)) ds = OK (output :: ds)
subgoal 4 (ID 238) is:
 fetch_decode_execute_loop (compile_aux (Plus ae1 ae2)) ds = KO output
subgoal 5 (ID 240) is:
 fetch_decode_execute_loop (compile_aux (Minus ae1 ae2)) ds = OK (output :: ds)
subgoal 6 (ID 242) is:
 fetch_decode_execute_loop (compile_aux (Minus ae1 ae2)) ds = KO output
```

(i) Subgoal 1: (Literal l), OK

In this case, we first use the fold-unfold lemmas of `compile`, `evaluate` to rewrite, that is:

```
rewrite -> fold_unfold_compile_aux_lit.
rewrite -> fold_unfold_evaluate_Literal in H_eval.
```

Then, the goal window reads:

```
l : nat
ds : data_stack
output : nat
H_eval : Expressible_nat l =
         Expressible_nat output
============================
fetch_decode_execute_loop
  (PUSH l :: nil) ds =
OK (output :: ds)
```

This prompts us to use the injection tactic. The rest of the proof is using the fold-unfold lemma of `fetch_decode_execute_loop` to rewrite:

```
injection H_eval as l_is_output.
rewrite -> l_is_output.
rewrite -> (fold_unfold_fetch_decode_execute_loop_cons (PUSH output) nil ds).
unfold decode_execute.
rewrite -> (fold_unfold_fetch_decode_execute_loop_nil (output :: ds)).
reflexivity.
```

(ii) Subgoal 2: (Literal l), KO

Since in the goal we have "KO," this indicates the presence of an error message. However, (Literal l) will always evaluate to a natural number. Thus, we use the `discriminate` tactic to show contradiction. Specifically:

```
rewrite -> fold_unfold_compile_aux_lit.
rewrite -> fold_unfold_evaluate_Literal in H_eval.
discriminate H_eval.
```

(iii) Subgoal 3: (Plus ae1 ae2), OK

Under this subgoal, we have the following cases (according to the result of evaluating ae1 and ae2 respectively). To be precise, we have the following 3 cases:

(a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value:

```
injection H_eval as n1_plus_n2_is_output.
destruct (IHae1 ds) as [IHae1' _].
destruct (IHae2 (n1 :: ds)) as [IHae2' _].
assert (IHae1'' := IHae1' n1 eq_refl).
assert (IHae2'' := IHae2' n2 eq_refl).
destruct (concatenation (compile_aux ae2) (ADD :: nil) (n1 :: ds)) as
↪  [H_concat_2 _].
assert (H_concat_2' := H_concat_2 (n2 :: n1 :: ds) IHae2'').
destruct (concatenation (compile_aux ae1) (compile_aux ae2 ++ ADD ::
↪  nil) ds) as [H_concat_1 _].
assert (H_concat_1' := H_concat_1 (n1 :: ds) IHae1'').
rewrite <- H_concat_1'.
rewrite <- H_concat_2'.
Check (fold_unfold_fetch_decode_execute_loop_cons ADD nil (n2 :: n1 ::
↪  ds)).
rewrite -> (fold_unfold_fetch_decode_execute_loop_cons ADD nil (n2 ::
↪  n1 :: ds)).
unfold decode_execute.
rewrite -> n1_plus_n2_is_output.
rewrite -> fold_unfold_fetch_decode_execute_loop_nil.
reflexivity.
```

(b) evaluating ae1 gives a nat value and evaluating ae2 gives an error message:
Simply use discriminate tactic because it is "OK" in the goal whereas ae2 evaluates to an error message.

(c) evaluating ae1 gives an error message.
Simply use discriminate tactic because it is "OK" in the goal whereas ae1 evaluates to an error message.

(iv) Subgoal 4: (Plus ae1 ae2), KO

Note that compared to Subgoal 3, the 3 cases are the same but the situation is "flipped" for Subgoal 4. To be precise, for each of the three cases we prove each respectively:

(a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value:
Simply use discriminate tactic because it is "KO" in the goal whereas ae1 and ae1 both evaluate to nat values.

(b) evaluating ae1 gives a nat value and evaluating ae2 gives an error message:

```
injection H_eval as s2_is_output.
rewrite <- s2_is_output.
```

```
destruct (concatenation (compile_aux ae1) (compile_aux ae2 ++ ADD ::
↪  nil) ds) as [H_concat_1 _].
destruct (IHae1 ds) as [n1_lemma _].
Check (n1_lemma n1 eq_refl).
(* fetch_decode_execute_loop (compile_aux ae1) ds = OK (n1 :: ds) *)
assert (H_concat_1' := H_concat_1 (n1 :: ds) (n1_lemma n1 eq_refl)).
destruct (concatenation (compile_aux ae2) (ADD :: nil) (n1 :: ds)) as [_
↪  H_concat_2].
destruct (IHae2 (n1 :: ds)) as [_ IHae2'].
Check (IHae2' s2 eq_refl).
assert (H_concat_2' := H_concat_2 s2  (IHae2' s2 eq_refl)).
rewrite <- H_concat_1'.
rewrite <- H_concat_2'.
reflexivity.
```

(c) evaluating ae1 gives an error message.

```
injection H_eval as s1_is_output.
rewrite <- s1_is_output.
destruct (concatenation (compile_aux ae1) (compile_aux ae2 ++ ADD :: nil)
↪  ds) as [_ H_concat_1].
destruct (IHae1 ds) as [_ IHae1'].
assert (H_concat_1' := H_concat_1 s1 (IHae1' s1 eq_refl)).
rewrite <- H_concat_1'.
reflexivity.
```

(v) Subgoal 5: (Minus ae1 ae2), OK Under this subgoal, we have the following cases (according to the result of evaluating ae1 and ae2 respectively). Note that there is one more case in Minus here than Plus because we need to consider numerical underflow. To be precise, we have the following 4 cases:

(a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$is true (there is numerical underflow):

Simply use discriminate tactic because it is "OK" in the goal whereas there is an error message (for numerical underflow).

(b) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is false (there is NO numerical underflow error):

```
injection H_eval as n1_minus_n2_is_output.
destruct (IHae1 ds) as [IHae1' _].
destruct (IHae2 (n1 :: ds)) as [IHae2' _].
assert (IHae1'' := IHae1' n1 eq_refl).
assert (IHae2'' := IHae2' n2 eq_refl).
destruct (concatenation (compile_aux ae2) (SUB :: nil) (n1 :: ds)) as
↪  [H_concat_2 _].
assert (H_concat_2' := H_concat_2 (n2 :: n1 :: ds) IHae2'').
destruct (concatenation (compile_aux ae1) (compile_aux ae2 ++ SUB ::
↪  nil) ds) as [H_concat_1 _].
```

```
assert (H_concat_1' := H_concat_1 (n1 :: ds) IHae1'').
rewrite <- H_concat_1'.
rewrite <- H_concat_2'.
Check (fold_unfold_fetch_decode_execute_loop_cons SUB nil (n2 :: n1 ::
↪  ds)).
rewrite -> (fold_unfold_fetch_decode_execute_loop_cons SUB nil (n2 ::
↪  n1 :: ds)).
unfold decode_execute.
rewrite -> H_underflow.
rewrite -> n1_minus_n2_is_output.
rewrite -> fold_unfold_fetch_decode_execute_loop_nil.
reflexivity.
```

(c) evaluating ae1 gives a nat value and evaluating ae2 gives an error message:
Simply use discriminate tactic because it is "OK" in the goal whereas evaluating ae2 gives an error message.

(d) evaluating ae1 gives an error message:
Simply use discriminate tactic because it is "OK" in the goal whereas evaluating ae1 gives an error message.

(vi) Subgoal 6: (Minus ae1 ae2), KO Note that compared to Subgoal 5, the 4 cases are the same but the situation is "flipped" for Subgoal 6. To be precise, for each of the three cases we prove each respectively:

(a) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is true (there is numerical underflow):

```
injection H_eval as error_is_output.
 destruct (concatenation (compile_aux ae1) (compile_aux ae2 ++ SUB ::
 ↪  nil) ds) as [H_concat_1 _].
 destruct (IHae1 ds) as [n1_lemma _].
 Check (n1_lemma n1 eq_refl).
 assert (H_concat_1' := H_concat_1 (n1 :: ds) (n1_lemma n1 eq_refl)).
 destruct (concatenation (compile_aux ae2) (SUB :: nil) (n1 :: ds)) as
 ↪  [H_concat_2 _].
 destruct (IHae2 (n1 :: ds)) as [n2_lemma _].
 Check (n2_lemma n2 eq_refl).
 assert (H_concat_2' := H_concat_2 (n2 :: n1 :: ds) (n2_lemma n2
 ↪  eq_refl)).
 rewrite <- H_concat_1'.
 rewrite <- H_concat_2'.
 rewrite -> (fold_unfold_fetch_decode_execute_loop_cons SUB nil (n2
 ↪  :: n1 :: ds)).
 unfold decode_execute.
 rewrite -> H_underflow.
 rewrite <- error_is_output.
 reflexivity.
```

(b) evaluating ae1 gives a nat value and evaluating ae2 gives a nat value and $n_1 < n_2$ is false (there is NO numerical underflow error):

Simply use discriminate tactic because it is "KO" in the goal whereas ae1 and ae2 both evaluate to nat values.

(c) evaluating ae1 gives a nat value and evaluating ae2 gives an error message:

```
injection H_eval as s2_is_output.
  rewrite <- s2_is_output.
  destruct (concatenation (compile_aux ae1) (compile_aux ae2 ++ SUB ::
  ↪  nil) ds) as [H_concat_1 _].
  destruct (IHae1 ds) as [n1_lemma _].
  Check (n1_lemma n1 eq_refl).
  assert (H_concat_1' := H_concat_1 (n1 :: ds) (n1_lemma n1 eq_refl)).
  destruct (concatenation (compile_aux ae2) (SUB :: nil) (n1 :: ds)) as
  ↪  [_ H_concat_2].
  destruct (IHae2 (n1 :: ds)) as [_ s2_lemma].
  Check (s2_lemma s2 eq_refl).
  assert (H_concat_2' := H_concat_2 s2 (s2_lemma s2 eq_refl)).
  rewrite <- H_concat_1'.
  rewrite <- H_concat_2'.
  reflexivity.
```

(d) evaluating ae1 gives an error message:

```
injection H_eval as s1_is_output.
rewrite <- s1_is_output.
destruct (concatenation (compile_aux ae1) (compile_aux ae2 ++ SUB :: nil)
↪  ds) as [_ H_concat_1].
destruct (IHae1 ds) as [_ s1_lemma].
Check (s1_lemma s1 eq_refl).
assert (H_concat_1' := H_concat_1 s1 (s1_lemma s1 eq_refl)).
rewrite <- H_concat_1'.
reflexivity.
```

11 Task 10: Byte-code verification

Prove that the compiler emits code that is accepted by the verifier.

11.1 Introduction

In this task, we investigate byte-code verification. Note that the following verifier symbolically executes a byte-code program to check whether no underflow occurs during execution and whether when the program completes, there is one and one only natural number on top of the stack. The second argument of `verify_aux`is a natural number that represents the size of the stack.

11.2 Theorem 29: The main theorem for byte-code verification

The main theorem that we want to prove in this task is stated as follows:

```
Theorem the_compiler_emits_well_behaved_code :
  forall sp: source_program,
    verify (compile sp) = true.
```

which means that given any source program, the compiled result will pass the verifier. To show this, we need to define the `verify` function and prove the intermediate lemma that the compiler emits well behaved code.

11.3 The verify function

First, we implement the helper function `verify_aux` that contains the recursive part of `verify`:

```
Fixpoint verify_aux (bcis : list byte_code_instruction) (n : nat) : option nat :=
  match bcis with
    | nil =>
      Some n
    | bci :: bcis' =>
      match bci with
        | PUSH _ =>
          verify_aux bcis' (S n)
        | _ =>
          match n with
            | S (S n') =>
              verify_aux bcis' (S n')
            | _ =>
              None
          end
      end
  end.
```

We state the fold-unfold lemmas associate with `verify_aux`, according to the cases:

- base case: `fold_unfold_verify_aux_nil`

- induction case:
  (i) PUSH: fold_unfold_verify_aux_cons_push
  (ii) ADD:
    (a) add_0
    (b) add_1
    (c) add_S
  (iii) SUB:
    (a) sub_0
    (b) sub_1
    (c) sub_S

```
Lemma fold_unfold_verify_aux_nil :
  forall n : nat,
    verify_aux nil n = Some n.
Proof.
  fold_unfold_tactic verify_aux.
Qed.

Lemma fold_unfold_verify_aux_cons_push :
  forall (bcis: list byte_code_instruction)
         (n1 n2 : nat),
    verify_aux (PUSH n1 :: bcis) n2 = verify_aux bcis (S n2).
Proof.
  fold_unfold_tactic verify_aux.
Qed.

Lemma fold_unfold_verify_aux_cons_add_0 :
  forall (bcis: list byte_code_instruction),
    verify_aux (ADD :: bcis) 0 = None.
Proof.
  fold_unfold_tactic verify_aux.
Qed.

Lemma fold_unfold_verify_aux_cons_add_1 :
  forall (bcis: list byte_code_instruction),
    verify_aux (ADD :: bcis) 1 = None.
Proof.
  fold_unfold_tactic verify_aux.
Qed.

Lemma fold_unfold_verify_aux_cons_add_S :
  forall (bcis: list byte_code_instruction)
         (n : nat),
```

```
    verify_aux (ADD :: bcis) (S (S n)) = verify_aux bcis (S n).
Proof.
  fold_unfold_tactic verify_aux.
Qed.

Lemma fold_unfold_verify_aux_cons_sub_0 :
  forall (bcis: list byte_code_instruction),
    verify_aux (SUB :: bcis) 0 = None.
Proof.
  fold_unfold_tactic verify_aux.
Qed.

Lemma fold_unfold_verify_aux_cons_sub_1 :
  forall (bcis: list byte_code_instruction),
    verify_aux (SUB :: bcis) 1 = None.
Proof.
  fold_unfold_tactic verify_aux.
Qed.

Lemma fold_unfold_verify_aux_cons_sub_S :
  forall (bcis: list byte_code_instruction)
         (n : nat),
    verify_aux (SUB :: bcis) (S (S n)) = verify_aux bcis (S n).
Proof.
  fold_unfold_tactic verify_aux.
Qed.

Lemma fold_unfold_verify_aux_S :
  forall (bcis : list byte_code_instruction)
         (n n' : nat),
    verify_aux bcis n = Some n' ->
    verify_aux bcis (S n) = Some (S n').
Proof.
  intro bcis.
  induction bcis as [ | bci' bcis' IHbcis'].
  - intros n n' H.
    rewrite fold_unfold_verify_aux_nil.
    rewrite fold_unfold_verify_aux_nil in H.
    apply nat_option_value_consistency_eq in H.
    apply eq_S in H.
    rewrite H.
    reflexivity.
  - intros n n' H.
    case bci' as [ n'' | | ].
    + rewrite fold_unfold_verify_aux_cons_push in H.
```

```
      rewrite fold_unfold_verify_aux_cons_push.
      apply IHbcis'.
      exact H.
    + case n as [ | [ | n''']].
      * rewrite fold_unfold_verify_aux_cons_add_0 in H.
        discriminate H.
      * rewrite fold_unfold_verify_aux_cons_add_1 in H.
        discriminate H.
      * rewrite fold_unfold_verify_aux_cons_add_S in H.
        rewrite fold_unfold_verify_aux_cons_add_S.
        apply IHbcis'.
        exact H.
    + case n as [ | [ | n''']].
      * rewrite fold_unfold_verify_aux_cons_sub_0 in H.
        discriminate H.
      * rewrite fold_unfold_verify_aux_cons_sub_1 in H.
        discriminate H.
      * rewrite fold_unfold_verify_aux_cons_sub_S in H.
        rewrite fold_unfold_verify_aux_cons_sub_S.
        apply IHbcis'.
        exact H.
Qed.
```

Using the `verify_aux` function, we then implemented the `verify` function:

```
Definition verify (p : target_program) : bool :=
  match p with
  | Target_program bcis =>
    match verify_aux bcis 0 with
    | Some n =>
      match n with
      | 1 =>
        true
      | _ =>
        false
      end
    | _ =>
      false
    end
  end.
```

11.4 Theorem 30: the compiler emits well-behaved code

The compiler emits well-behaved code if

- no underflow occurs during execution

- when the program completes, there is one and one only natural number on top of the stack.

Thus, we formalize the theorem:

```
Lemma the_compiler_emits_well_behaved_code_aux :
  forall (ae : arithmetic_expression)
         (n : nat),
    verify_aux (compile_aux ae) n = Some (S n).
```

This is a proof by induction, we start with

```
induction ae as [n' | ae1 IHae1 ae2 IHae2 | ae1 IHae1 ae2 IHae2].
```

We thus have three cases: the base case `PUSH n`, the induction case for ADD, the induction case for SUB. We focus on each subgoal one by one:

(i) The base case: `PUSH n`

```
- intro n.
rewrite fold_unfold_compile_aux_lit.
rewrite fold_unfold_verify_aux_cons_push.
rewrite fold_unfold_verify_aux_nil.
reflexivity.
```

(ii) The induction case: `ADD`

```
 - intro n.
rewrite fold_unfold_compile_aux_plus.
remember (compile_aux ae1) as bcis1.
remember (compile_aux ae2) as bcis2.
```

Now the goal reads:

```
1 subgoal (ID 301)

  ae1, ae2 : arithmetic_expression
  bcis1 : list byte_code_instruction
  Heqbcis1 : bcis1 = compile_aux ae1
  IHae1 : forall n : nat, verify_aux bcis1 n = Some (S n)
  bcis2 : list byte_code_instruction
  Heqbcis2 : bcis2 = compile_aux ae2
  IHae2 : forall n : nat, verify_aux bcis2 n = Some (S n)
  n : nat
  ============================
  verify_aux (bcis1 ++ bcis2 ++ ADD :: nil) n = Some (S n)
```

thus we need a helper lemma, which we state and prove now:

```
      Lemma verify_aux_combine_two_lists_alt :
  forall (bcis1 bcis2: list byte_code_instruction)
         (n0 n1 : nat),
    verify_aux bcis1 n0 = Some n1 ->
    verify_aux bcis2 n1 = verify_aux (bcis1 ++ bcis2) n0.
Proof.
  intros bcis1 bcis2 n0 n1 H.
  revert n0 H.
  induction bcis1 as [ | bci1' bcis1' IHbcis1'].
  - intros n0 H.
    rewrite fold_unfold_verify_aux_nil in H.
    rewrite fold_unfold_append_nil.
```

The goal now reads:

```
1 subgoal (ID 280)

  bcis2 : list byte_code_instruction
  n1, n0 : nat
  H : Some n0 = Some n1
  ============================
  verify_aux bcis2 n1 = verify_aux bcis2 n0
```

which calls for yet another (short) helper lemma:

```
Lemma nat_option_value_consistency_eq :
  forall n1 n2: nat,
    n1 = n2 <-> Some n1 = Some n2.
Proof.
  intros n1 n2.
  split.
  - intro H_n.
    rewrite H_n.
    reflexivity.
  - intro H_ov.
    injection H_ov as H_v.
    exact H_v.
Qed.
```

Having proved the lemma above, we can proceed:

```
    rewrite <- nat_option_value_consistency_eq in H.
    rewrite H.
    reflexivity.
```

This completes the base case for the helper lemma about combining two lists. The induction case for `ADD` and `SUB` are proved respectively as follows:

```
- intros n0 H.
  rewrite fold_unfold_append_cons.
  case bci1' as [ n | | ].
  + rewrite fold_unfold_verify_aux_cons_push.
    rewrite fold_unfold_verify_aux_cons_push in H.
    apply IHbcis1'.
    exact H.
  + case n0 as [ | [ | n0' ]].
    * rewrite fold_unfold_verify_aux_cons_add_0 in H.
      discriminate H.
    * rewrite fold_unfold_verify_aux_cons_add_1 in H.
      discriminate H.
    * rewrite fold_unfold_verify_aux_cons_add_S in H.
      rewrite fold_unfold_verify_aux_cons_add_S.
      apply IHbcis1'.
      exact H.
  + case n0 as [ | [ | n0' ]].
    * rewrite fold_unfold_verify_aux_cons_sub_0 in H.
      discriminate H.
    * rewrite fold_unfold_verify_aux_cons_sub_1 in H.
      discriminate H.
    * rewrite fold_unfold_verify_aux_cons_sub_S in H.
      rewrite fold_unfold_verify_aux_cons_sub_S.
      apply IHbcis1'.
      exact H.
```

With this, we complete the helper lemma about combing two lists. Then we can proceed in our proof for theorem about compiler emitting well-behaved code. We finish the subgoal where we left off with the following steps:

```
assert (H_concat := (verify_aux_combine_two_lists bcis1 bcis2 n (S n) (S (S n)))
↪  (IHae1 n) (IHae2 (S n))).
assert (H_add := fold_unfold_verify_aux_cons_add_S nil n).
rewrite app_assoc.
exact ((verify_aux_combine_two_lists (bcis1 ++ bcis2) (ADD :: nil) n (S (S n)) (S
↪  n)) H_concat H_add).
```

(iii) The induction case: `SUB`

This subgoal is proved exactly like the subgoal for `ADD`, specifically:

```
intro n.
rewrite fold_unfold_compile_aux_minus.
remember (compile_aux ae1) as bcis1.
remember (compile_aux ae2) as bcis2.
assert (H_concat := (verify_aux_combine_two_lists bcis1 bcis2 n (S n) (S (S n)))
↪  (IHae1 n) (IHae2 (S n))).
```

```
    assert (H_sub := fold_unfold_verify_aux_cons_sub_S nil n).
    rewrite app_assoc.
    exact ((verify_aux_combine_two_lists (bcis1 ++ bcis2) (SUB :: nil) n (S (S n)) (S
    ↪  n)) H_concat H_sub).
```

Thus, we complete the proof for (Theorem 30: the compiler emits well-behaved code).

11.5 Conclusion

With that, we can complete the proof for the main theorem for byte-code verification:

```
Theorem the_compiler_emits_well_behaved_code :
  forall sp: source_program,
    verify (compile sp) = true.
    Proof.
  intros [ae].
  unfold verify, compile.
  rewrite the_compiler_emits_well_behaved_code_aux.
  reflexivity.
Qed.
```

12 Task 11: The world seen in Magritte

12.1 Introduction

In this task, we investigate the Magritte version of the interpreter for the source program and target program respectively. While the standard interpreters operate on natural numbers, the Magritte interpreters operate on arithmetic expressions which are syntactic representations of natural numbers.

12.2 11a: Magritte interpreter for the source program

a. Write a Magritte interpreter for the source language that does not operate on natural numbers but on syntactic representations of natural numbers.

Recall that the standard source interpreter in Task 1 has type:

```
arithmetic_expression -> expressible_value
```

Now, the Magritte interpreter has type:

```
arithmetic_expression -> arithmetic_expression
```

Given any arithmetic expression, we have the following cases:

- The Magritte interpreter maps `Literal n` to `Literal n`
- If the Magritte interpreter has evaluated ae1 into ae1' and ae2 into ae2', then it maps `Plus ae1 ae2` to `Plus ae1' ae2'`.
- If the Magritte interpreter has evaluated ae1 into ae1' and ae2 into ae2', then it maps `Minus ae1 ae2` to `Minus ae1' ae2'`.

Thus, we define the type `Expressible_value_m` for the output of the Magritte interpreter.

```
Inductive Expressible_value_m : Type :=
| Expressible_nat_m : arithmetic_expression -> Expressible_value_m
| Expressible_msg_m : string -> Expressible_value_m.
```

Now we reformulate the specification of the standard `evaluate` function into the Magritte version. Note that the structure is inherited from standard version with the only difference being that in the Magritte world, nothing is computed. Therefore,

- `Literal n` evaluates to `Literal n` (instead of n).
- `PLUS (ae1 ae2)` evaluates to another arithmetic expression `PLUS (ae1' ae2')` (instead of the result of addition which is a nat value as in the standard interpreter).
- Similarly, `MINUS (ae1 ae2)` evaluates to another arithmetic expression `MINUS (ae1' ae2')` (instead of the result of subtraction which is a nat value as in the standard interpreter).

```
Definition specification_of_evaluate_m (evaluate_m : arithmetic_expression ->
↪  Expressible_value_m) :=
  (forall n : nat,
```

```
    evaluate_m (Literal n) = Expressible_nat_m (Literal n))
  /\
  ((forall (ae1 ae2 : arithmetic_expression)
           (s1 : string),
       evaluate_m ae1 = Expressible_msg_m s1 ->
       evaluate_m (Plus ae1 ae2) = Expressible_msg_m s1)
   /\
   (forall (ae1 ae2 ae1' : arithmetic_expression)
           (s2 : string),
       evaluate_m ae1 = Expressible_nat_m ae1' ->
       evaluate_m ae2 = Expressible_msg_m s2 ->
       evaluate_m (Plus ae1 ae2) = Expressible_msg_m s2)
   /\
   (forall (ae1 ae2 ae1' ae2' : arithmetic_expression),
       evaluate_m ae1 = Expressible_nat_m ae1' ->
       evaluate_m ae2 = Expressible_nat_m ae2' ->
       evaluate_m (Plus ae1 ae2) = Expressible_nat_m (Plus ae1'  ae2')))
  /\
  ((forall (ae1 ae2 : arithmetic_expression)
            (s1 : string),
       evaluate_m ae1 = Expressible_msg_m s1 ->
       evaluate_m (Minus ae1 ae2) = Expressible_msg_m s1)
   /\
   (forall (ae1 ae2 ae1' : arithmetic_expression)
           (s2 : string),
       evaluate_m ae1 = Expressible_nat_m ae1' ->
       evaluate_m ae2 = Expressible_msg_m s2 ->
       evaluate_m (Minus ae1 ae2) = Expressible_msg_m s2)
   /\
   (forall (ae1 ae2 ae1' ae2' : arithmetic_expression),
       evaluate_m ae1 = Expressible_nat_m ae1' ->
       evaluate_m ae2 = Expressible_nat_m ae2' ->
       evaluate_m (Minus ae1 ae2) = Expressible_nat_m (Minus ae1' ae2'))).
```

Given the above specification, we implement the Magritte version for `evaluate` function and state its associated fold-unfold lemmas:

```
Fixpoint evaluate_m (ae : arithmetic_expression) : Expressible_value_m :=
 match ae with
  | Literal n =>
    Expressible_nat_m (Literal n)
  | Plus ae1 ae2 =>
    match evaluate_m ae1 with
    | Expressible_nat_m ae1' =>
      match evaluate_m ae2 with
      | Expressible_nat_m ae2' =>
```

```
        Expressible_nat_m (Plus ae1' ae2')
      | Expressible_msg_m s =>
        Expressible_msg_m s
      end
    | Expressible_msg_m s =>
      Expressible_msg_m s
    end
  | Minus ae1 ae2 =>
    match evaluate_m ae1 with
    | Expressible_nat_m ae1' =>
      match evaluate_m ae2 with
      | Expressible_nat_m ae2' =>
        Expressible_nat_m (Minus ae1' ae2')
      | Expressible_msg_m s =>
        Expressible_msg_m s
      end
    | Expressible_msg_m s =>
      Expressible_msg_m s
    end
 end.

Lemma fold_unfold_evaluate_m_Literal :
  forall n : nat,
    evaluate_m (Literal n) = Expressible_nat_m  (Literal n).
Proof.
  fold_unfold_tactic evaluate_m.
Qed.

Lemma fold_unfold_evaluate_m_Plus :
  forall (ae1 ae2 : arithmetic_expression),
   evaluate_m (Plus ae1 ae2) =
    match evaluate_m ae1 with
    | Expressible_nat_m  ae1' =>
      match evaluate_m ae2 with
      | Expressible_nat_m  ae2' =>
        Expressible_nat_m  (Plus ae1' ae2')
      | Expressible_msg_m s =>
        Expressible_msg_m s
      end
    | Expressible_msg_m s =>
      Expressible_msg_m s
    end.
Proof.
  fold_unfold_tactic evaluate_m.
Qed.
```

```
Lemma fold_unfold_evaluate_m_Minus :
  forall (ae1 ae2 : arithmetic_expression),
    evaluate_m (Minus ae1 ae2) =
    match evaluate_m ae1 with
    | Expressible_nat_m  ae1' =>
      match evaluate_m ae2 with
      | Expressible_nat_m  ae2' =>
        Expressible_nat_m  (Minus ae1' ae2')
      | Expressible_msg_m s =>
        Expressible_msg_m s
      end
    | Expressible_msg_m s =>
      Expressible_msg_m s
    end.
Proof.
  fold_unfold_tactic evaluate_m.
Qed.
```

Instantiating `evaluate_m` with arithmetic expression from the source program, we get the Magritte source interpreter:

```
Definition specification_of_interpret_m (interpret_m : source_program ->
↪  Expressible_value_m) :=
  forall evaluate_m : arithmetic_expression -> Expressible_value_m,
    specification_of_evaluate_m evaluate_m ->
    forall ae : arithmetic_expression,
      interpret_m (Source_program ae) = evaluate_m ae.

Definition interpret_m (sp : source_program) : Expressible_value_m :=
  match sp with
  | Source_program ae =>
    evaluate_m ae
  end.
```

12.3 11b: Magritte interpreter for the target program

b. Write a Magritte interpreter for the target language that does not operate on natural numbers but on syntactic representations of natural numbers.

Similar to the comparison between the Magritte source interpreter and the standard source interpreter, the standard target interpreter operates with a stack of nats, and the Magritte target interpreter operates with a stack of representations of nats, i.e., a stack of arithmetic expressions. Thus we define the Magritte data stack:

```
Definition data_stack_m := list arithmetic_expression.
```

Given any list of byte code instructions (bcis), we have the following cases:

- given `PUSH n` and `ds`, it yields `(Literal n) :: ds`.
- given `ADD` and `Plus ae1 ae2`, it yields `(Plus a1' ae2') :: ds`.
- given `SUB` and `Minus ae1 ae2`, it yields `(Plus a1' ae2') :: ds`.

As such, we define the inductive type for the the Magritte counterpart of the `result_of_decoding_and_executio`

```
Inductive result_of_decoding_and_execution_m : Type :=
| OK_m : data_stack_m -> result_of_decoding_and_execution_m
| KO_m : string -> result_of_decoding_and_execution_m.
```

With these definitions formalized in Coq, we are now equipped to state the specification of the Magritte `decode_execute` and implement the Magritte `decode_execute`. Similar to what we have done for the Magritte source interpreter, for the Magritte target interpreter, the structure is inherited from standard version with the only difference being that in the Magritte world, nothing is computed. Therefore,

- executing `(PUSH n) ds` returns `OK_m ((Literal n) :: ds)` (instead of `OK_m (n :: ds)`).
- executing `ADD (ae2 :: ae1 :: ds)` returns `OK_m ((Plus ae1 ae2) :: ds))` (instead of `OK ((n1 + n2) :: ds` in the standard target interpreter).
- Similarly, executing `SUB (ae2 :: ae1 :: ds)` returns `OK_m ((Minus ae1 ae2) :: ds)` (instead of `OK ((n1 - n2) :: ds` in the standard target interpreter).

The specific implementation is shown below:

```
Definition specification_of_decode_execute_m (decode_execute_m : byte_code_instruction
↪  -> data_stack_m -> result_of_decoding_and_execution_m) :=
  (forall (n: nat)
          (ds : data_stack_m),
     decode_execute_m (PUSH n) ds = OK_m ((Literal n) :: ds))
  /\
  ((decode_execute_m ADD nil = KO_m "ADD: stack underflow")
   /\
   (forall (ae1 : arithmetic_expression),
       decode_execute_m ADD (ae1 :: nil) = KO_m "ADD: stack underflow")
   /\
   (forall (ae1 ae2 : arithmetic_expression)
           (ds : data_stack_m),
       decode_execute_m ADD (ae2 :: ae1 :: ds) = OK_m ((Plus ae1 ae2) :: ds)))
  /\
  ((decode_execute_m SUB nil = KO_m "SUB: stack underflow")
   /\
   (forall (ae2 : arithmetic_expression),
       decode_execute_m SUB (ae2 :: nil) = KO_m "SUB: stack underflow")
   /\
   (forall (ae1 ae2 : arithmetic_expression)
```

```
          (ds : data_stack_m),
        decode_execute_m SUB (ae2 :: ae1 :: ds) = OK_m ((Minus ae1 ae2) :: ds))).

Definition decode_execute_m (bcis : byte_code_instruction) (ds : data_stack_m) :
↪  result_of_decoding_and_execution_m :=
  match bcis with
  | PUSH n =>
    OK_m ((Literal n) :: ds)
  | ADD =>
    match ds with
    | nil =>
      KO_m "ADD: stack underflow"
    | ae2 :: nil =>
      KO_m "ADD: stack underflow"
    | ae2 :: ae1 :: ds =>
      OK_m ((Plus ae1 ae2) :: ds)
    end
  | SUB =>
    match ds with
    | nil =>
      KO_m "SUB: stack underflow"
    | ae2 :: nil =>
      KO_m "SUB: stack underflow"
    | ae2 :: ae1 :: ds =>
      OK_m ((Minus ae1 ae2) :: ds)
    end
  end.
```

Following the motion that we has gone through in the previous tasks for the standard target interpreter, we state the specification of the Magritte `fetch_decode_execute_loop` (aka the Magritte virtual machine) and implement the Magritte `fetch_decode_execute_loop`:

```
Definition specification_of_fetch_decode_execute_loop_m (fetch_decode_execute_loop_m :
↪  list byte_code_instruction -> data_stack_m -> result_of_decoding_and_execution_m) :=
  forall decode_execute_m : byte_code_instruction -> data_stack_m ->
  ↪  result_of_decoding_and_execution_m,
    specification_of_decode_execute_m decode_execute_m ->
    (forall ds : data_stack_m,
        fetch_decode_execute_loop_m nil ds = OK_m ds)
    /\
    (forall (bci : byte_code_instruction)
            (bcis' : list byte_code_instruction)
            (ds ds' : data_stack_m),
        decode_execute_m bci ds = OK_m ds' ->
        fetch_decode_execute_loop_m (bci :: bcis') ds =
        fetch_decode_execute_loop_m bcis' ds')
```

```
    /\
    (forall (bci : byte_code_instruction)
            (bcis' : list byte_code_instruction)
            (ds : data_stack_m)
            (s : string),
        decode_execute_m bci ds = KO_m s ->
        fetch_decode_execute_loop_m (bci :: bcis') ds =
        KO_m s).

Fixpoint fetch_decode_execute_loop_m (bcis : list byte_code_instruction) (ds :
↪  data_stack_m) : result_of_decoding_and_execution_m :=
  match bcis with
  | nil => OK_m ds
  | bci :: bcis' =>
    match (decode_execute_m bci ds) with
    | OK_m ds' => fetch_decode_execute_loop_m bcis' ds'
    | KO_m s => KO_m s
    end
  end.

Lemma fold_unfold_fetch_decode_execute_loop_m_nil :
  forall ds : data_stack_m,
    fetch_decode_execute_loop_m nil ds = OK_m ds.
Proof.
  fold_unfold_tactic fetch_decode_execute_loop_m.
Qed.

Lemma fold_unfold_fetch_decode_execute_loop_m_cons :
  forall (bci: byte_code_instruction)
         (bcis': list byte_code_instruction)
         (ds : data_stack_m),
    fetch_decode_execute_loop_m (bci :: bcis') ds =
    match decode_execute_m bci ds with
    | OK_m ds' => fetch_decode_execute_loop_m bcis' ds'
    | KO_m s => KO_m s
    end.
Proof.
  fold_unfold_tactic fetch_decode_execute_loop_m.
Qed.
```

Finally, as has been done for the standard target interpreter, we state the specification of the Magritte `run` and implement the Magritte `run` has follows:

```
Definition specification_of_run_m (run_m : target_program -> Expressible_value_m) :=
  forall fetch_decode_execute_loop_m : list byte_code_instruction -> data_stack_m ->
  ↪  result_of_decoding_and_execution_m,
```

```
    specification_of_fetch_decode_execute_loop_m fetch_decode_execute_loop_m ->
    (forall (bcis : list byte_code_instruction),
       fetch_decode_execute_loop_m bcis nil = OK_m nil ->
       run_m (Target_program bcis) = Expressible_msg_m "no result on the data stack")
    /\
    (forall (bcis : list byte_code_instruction)
            (ae : arithmetic_expression),
       fetch_decode_execute_loop_m bcis nil = OK_m (ae :: nil) ->
       run_m (Target_program bcis) = Expressible_nat_m  ae)
    /\
    (forall (bcis : list byte_code_instruction)
            (ae ae' : arithmetic_expression)
            (ds'' : data_stack_m),
       fetch_decode_execute_loop_m bcis nil = OK_m (ae :: ae' :: ds'') ->
       run_m (Target_program bcis) = Expressible_msg_m "too many results on the data
       ↪  stack")
    /\
    (forall (bcis : list byte_code_instruction)
            (s : string),
       fetch_decode_execute_loop_m bcis nil = KO_m s ->
       run_m (Target_program bcis) = Expressible_msg_m s).

Definition run_m (tp : target_program) : Expressible_value_m :=
  match tp with
  | Target_program bcis =>
    match (fetch_decode_execute_loop_m bcis nil) with
    | OK_m nil =>
      Expressible_msg_m "no result on the data stack"
    | OK_m (ae :: nil) =>
      Expressible_nat_m  ae
    | OK_m (ae :: ae' :: ds'') =>
      Expressible_msg_m "too many results on the data stack"
    | KO_m s =>
      Expressible_msg_m s
    end
  end.
```

And with this we conclude part b of task 11.

12.4 11c: The Magritte commutative diagram

c. Prove that interpreting an arithmetic expression with the Magritte source interpreter gives the same result as first compiling it and then executing the compiled program with the Magritte target interpreter over an empty data stack.

An overview for the proof of the Magritte evaluate ultimate theorem is given in Figure 8:

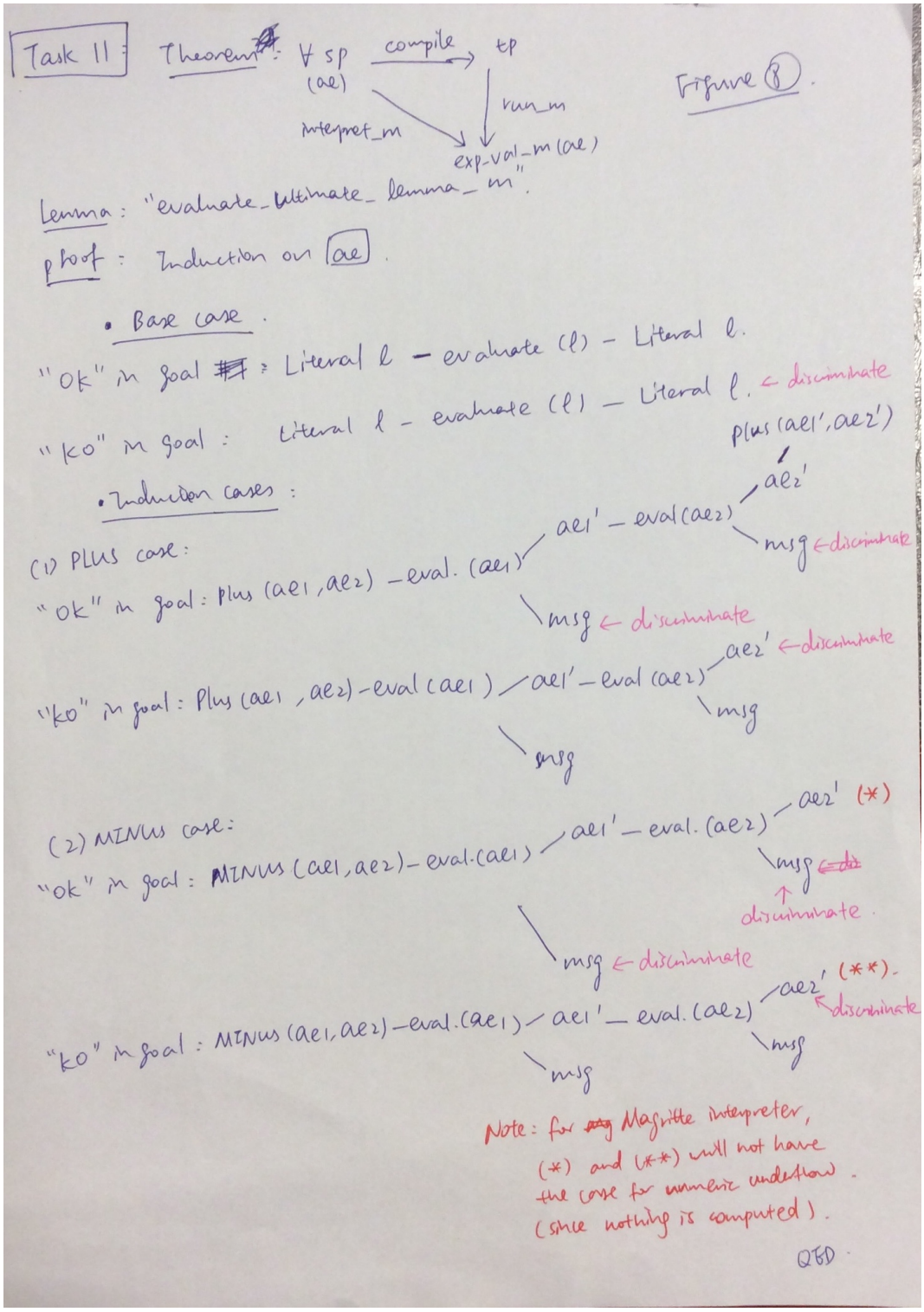


Figure 8: Figure 8

The proof for the Magritte commutative diagram has the same structure as that for the standard commutative diagram. One significant difference worth noting here is that there will not be any error message for numerical underflow, again, because nothing is computed in the Magritte world. Thus the cases are slightly different, as can be observed from the diagram in Figure 8.

And the specifics of the proof is shown below for the sake of easier reference and completeness.

```
Theorem concatenation_m :
  forall (bcis1 bcis2 : list byte_code_instruction)
         (ds: data_stack_m) ,
  (forall (ds': data_stack_m),
    (fetch_decode_execute_loop_m bcis1 ds) = OK_m ds'
    ->
    (fetch_decode_execute_loop_m bcis2 ds') = (fetch_decode_execute_loop_m (bcis1 ++
    ↪  bcis2) ds))
  /\
  (forall (s' : string),
      (fetch_decode_execute_loop_m bcis1 ds) = KO_m s'
      ->
      KO_m s' = (fetch_decode_execute_loop_m (bcis1 ++ bcis2) ds)).
Proof.
  unfold specification_of_fetch_decode_execute_loop_m.
  intro bcis1.
  induction bcis1 as [ | bci bcis1' IHbcis1 ].
  + intros bcis2 ds.
    split.
  - intros ds' H_OK_nil.
    rewrite -> (fold_unfold_fetch_decode_execute_loop_m_nil) in H_OK_nil.
    rewrite -> (fold_unfold_append_nil bcis2).
    injection H_OK_nil.
    intro H_equal.
    rewrite -> (H_equal).
    reflexivity.
  - intros s' H_KO_nil.
    rewrite -> (fold_unfold_fetch_decode_execute_loop_m_nil) in H_KO_nil.
    discriminate H_KO_nil.
  + intros bcis2 ds.
    split.
    - intros ds' H_OK_cons.
      rewrite -> (fold_unfold_fetch_decode_execute_loop_m_cons) in H_OK_cons.
      Check (fold_unfold_append_cons).
      rewrite -> (fold_unfold_append_cons bci bcis1' bcis2).
      Check (fold_unfold_fetch_decode_execute_loop_m_cons).
     rewrite -> (fold_unfold_fetch_decode_execute_loop_m_cons bci (bcis1' ++ bcis2) ds).
      destruct (decode_execute_m bci ds) as [ds'' | s'].
      -- destruct (IHbcis1 bcis2 ds'') as [IHds IHs].
         rewrite (IHds ds' H_OK_cons).
```

```
      reflexivity.
    -- discriminate.
  - intros s' H_KO_cons.
    rewrite -> (fold_unfold_fetch_decode_execute_loop_m_cons) in H_KO_cons.
    Check (fold_unfold_append_cons).
    rewrite -> (fold_unfold_append_cons bci bcis1' bcis2).
    Check (fold_unfold_fetch_decode_execute_loop_m_cons).
    rewrite -> (fold_unfold_fetch_decode_execute_loop_m_cons bci (bcis1' ++ bcis2) ds).
    destruct (decode_execute_m bci ds) as [ds'' | s''].
    -- destruct (IHbcis1 bcis2 ds'') as [IHds IHs].
      rewrite (IHs s' H_KO_cons).
      reflexivity.
    -- symmetry.
      exact H_KO_cons.
Qed.

Lemma evaluate_ultimate_lemma_m :
  forall (ae : arithmetic_expression)
         (ds : data_stack_m),
    (forall (ae' : arithmetic_expression),
      evaluate_m ae = Expressible_nat_m ae' -> fetch_decode_execute_loop_m (compile_aux
      ↪  ae) ds = OK_m (ae' :: ds)) /\
  (forall (s : string),
     evaluate_m ae = Expressible_msg_m s -> fetch_decode_execute_loop_m (compile_aux ae)
      ↪  ds = KO_m s).
Proof.
  induction ae as [ l | ae1 IHae1 ae2 IHae2 | ae1 IHae1 ae2 IHae2]; intro ds; split;
  ↪  intros output H_eval.
  - rewrite -> fold_unfold_compile_aux_lit.
    rewrite -> fold_unfold_evaluate_m_Literal in H_eval.
    injection H_eval as l_is_output.
    rewrite -> fold_unfold_fetch_decode_execute_loop_m_cons.
    unfold decode_execute_m.
    rewrite -> l_is_output.
    rewrite -> fold_unfold_fetch_decode_execute_loop_m_nil.
    reflexivity.
  - rewrite -> fold_unfold_compile_aux_lit.
    rewrite -> fold_unfold_evaluate_m_Literal in H_eval.
    discriminate H_eval.
  - rewrite -> fold_unfold_compile_aux_plus.
    rewrite -> fold_unfold_evaluate_m_Plus in H_eval.
    destruct (evaluate_m ae1) as [ae1' | s1] eqn:H_ae1.
    + destruct (evaluate_m ae2) as [ae2' | s2] eqn:H_ae2.
      * injection H_eval as ae1'_plus_ae2'.
        destruct (IHae1 ds) as [IHae1' _].
```

```
      destruct (IHae2 (ae1' :: ds)) as [IHae2' _].
      assert (IHae1'' := IHae1' ae1' eq_refl).
      assert (IHae2'' := IHae2' ae2' eq_refl).
      destruct (concatenation_m (compile_aux ae2) (ADD :: nil) (ae1' :: ds)) as
      ↪  [H_concat_2 _].
      assert (H_concat_2' := H_concat_2 (ae2' :: ae1' :: ds) IHae2'').
      destruct (concatenation_m (compile_aux ae1) (compile_aux ae2 ++ ADD :: nil) ds)
      ↪  as [H_concat_1 _].
      assert (H_concat_1' := H_concat_1 (ae1' :: ds) IHae1'').
      rewrite <- H_concat_1'.
      rewrite <- H_concat_2'.
     Check (fold_unfold_fetch_decode_execute_loop_m_cons ADD nil (ae2' :: ae1' :: ds)).
      rewrite -> (fold_unfold_fetch_decode_execute_loop_m_cons ADD nil (ae2' :: ae1' ::
      ↪  ds)).
      unfold decode_execute_m.
      rewrite -> ae1'_plus_ae2'.
      rewrite -> fold_unfold_fetch_decode_execute_loop_m_nil.
      reflexivity.
    * discriminate H_eval.
  + discriminate H_eval.
- rewrite -> fold_unfold_compile_aux_plus.
  rewrite -> fold_unfold_evaluate_m_Plus in H_eval.
  destruct (evaluate_m ae1) as [ae1' | s1] eqn:H_ae1.
  + destruct (evaluate_m ae2) as [ae2' | s2] eqn:H_ae2.
    * discriminate H_eval.
    * injection H_eval as s2_is_output.
      rewrite <- s2_is_output.
      destruct (concatenation_m (compile_aux ae1) (compile_aux ae2 ++ ADD :: nil) ds)
      ↪  as [H_concat_1 _].
      destruct (IHae1 ds) as [ae1'_lemma _].
      Check (ae1'_lemma ae1' eq_refl).
      assert (H_concat_1' := H_concat_1 (ae1' :: ds) (ae1'_lemma ae1' eq_refl)).
      destruct (concatenation_m (compile_aux ae2) (ADD :: nil) (ae1' :: ds)) as [_
      ↪  H_concat_2].
      destruct (IHae2 (ae1' :: ds)) as [_ IHae2'].
      Check (IHae2' s2 eq_refl).
      assert (H_concat_2' := H_concat_2 s2  (IHae2' s2 eq_refl)).
      rewrite <- H_concat_1'.
      rewrite <- H_concat_2'.
      reflexivity.
  + injection H_eval as s1_is_output.
    rewrite <- s1_is_output.
    destruct (concatenation_m (compile_aux ae1) (compile_aux ae2 ++ ADD :: nil) ds) as
    ↪  [_ H_concat_1].
    destruct (IHae1 ds) as [_ IHae1'].
```

```
    assert (H_concat_1' := H_concat_1 s1 (IHae1' s1 eq_refl)).
    rewrite <- H_concat_1'.
    reflexivity.
- rewrite -> fold_unfold_compile_aux_minus.
  rewrite -> fold_unfold_evaluate_m_Minus in H_eval.
  destruct (evaluate_m ae1) as [ae1' | s1] eqn:H_ae1.
  + destruct (evaluate_m ae2) as [ae2' | s2] eqn:H_ae2.
    * injection H_eval as ae1'_minus_ae2'_is_output.
      destruct (IHae1 ds) as [IHae1' _].
      destruct (IHae2 (ae1' :: ds)) as [IHae2' _].
      assert (IHae1'' := IHae1' ae1' eq_refl).
      assert (IHae2'' := IHae2' ae2' eq_refl).
      destruct (concatenation_m (compile_aux ae2) (SUB :: nil) (ae1' :: ds)) as
      ↪  [H_concat_2 _].
      assert (H_concat_2' := H_concat_2 (ae2' :: ae1' :: ds) IHae2'').
      destruct (concatenation_m (compile_aux ae1) (compile_aux ae2 ++ SUB :: nil) ds)
      ↪  as [H_concat_1 _].
      assert (H_concat_1' := H_concat_1 (ae1' :: ds) IHae1'').
      rewrite <- H_concat_1'.
      rewrite <- H_concat_2'.
     Check (fold_unfold_fetch_decode_execute_loop_m_cons SUB nil (ae2' :: ae1' :: ds)).
     rewrite -> (fold_unfold_fetch_decode_execute_loop_m_cons SUB nil (ae2' :: ae1' ::
      ↪  ds)).
      unfold decode_execute_m.
      rewrite -> ae1'_minus_ae2'_is_output.
      rewrite -> fold_unfold_fetch_decode_execute_loop_m_nil.
      reflexivity.
      * discriminate H_eval.
  + discriminate H_eval.
- rewrite -> fold_unfold_compile_aux_minus.
  rewrite -> fold_unfold_evaluate_m_Minus in H_eval.
  destruct (evaluate_m ae1) as [ae1' | s1] eqn:H_ae1.
  + destruct (evaluate_m ae2) as [ae2' | s2] eqn:H_ae2.
    * discriminate H_eval.
    * injection H_eval as s2_is_output.
      rewrite <- s2_is_output.
      destruct (concatenation_m (compile_aux ae1) (compile_aux ae2 ++ SUB :: nil) ds)
      ↪  as [H_concat_1 _].
      destruct (IHae1 ds) as [ae1'_lemma _].
      Check (ae1'_lemma ae1' eq_refl).
      assert (H_concat_1' := H_concat_1 (ae1' :: ds) (ae1'_lemma ae1' eq_refl)).
      destruct (concatenation_m (compile_aux ae2) (SUB :: nil) (ae1' :: ds)) as [_
      ↪  H_concat_2].
      destruct (IHae2 (ae1' :: ds)) as [_ s2_lemma].
      Check (s2_lemma s2 eq_refl).
```

```
      assert (H_concat_2' := H_concat_2 s2 (s2_lemma s2 eq_refl)).
      rewrite <- H_concat_1'.
      rewrite <- H_concat_2'.
      reflexivity.
  + injection H_eval as s1_is_output.
    rewrite <- s1_is_output.
    destruct (concatenation_m (compile_aux ae1) (compile_aux ae2 ++ SUB :: nil) ds) as
    ↪  [_ H_concat_1].
    destruct (IHae1 ds) as [_ s1_lemma].
    Check (s1_lemma s1 eq_refl).
    assert (H_concat_1' := H_concat_1 s1 (s1_lemma s1 eq_refl)).
    rewrite <- H_concat_1'.
    reflexivity.
Qed.


Theorem the_commutative_diagram_m :
  forall sp : source_program,
    interpret_m sp = run_m (compile sp).
Proof.
  intros [ ae ].
  unfold interpret_m.
  unfold compile.
  unfold run_m.
  destruct (evaluate_m ae) as [ae' | s] eqn : H_evaluate.
  - destruct (evaluate_ultimate_lemma_m ae nil) as [ae'_lemma _].
    rewrite -> (ae'_lemma ae').
    reflexivity.
    exact H_evaluate.
  - destruct (evaluate_ultimate_lemma_m ae nil) as [_ s_lemma].
    rewrite -> (s_lemma s).
    reflexivity.
    exact H_evaluate.
Qed.
```

12.5 11d: Decompile theorem

d. Prove that the Magritte target interpreter is (essentially) a left inverse of the compiler, i.e., it is a decompiler.

This theorem is a direct corollary of the commutative diagram. To see why, refer to Figure 9. By the commutative diagram, for all source program, `interpret_m` = `run_m` ∘ `compile` (where ∘ is function composition). Then, in particular when `interpret_m` = I (the identity function), i.e., when for all source program sp, `interpret_m` (sp) = sp, we have:

$$\Longrightarrow$$

which implies that `run_m` the Magritte source interpreter is a decompiler.

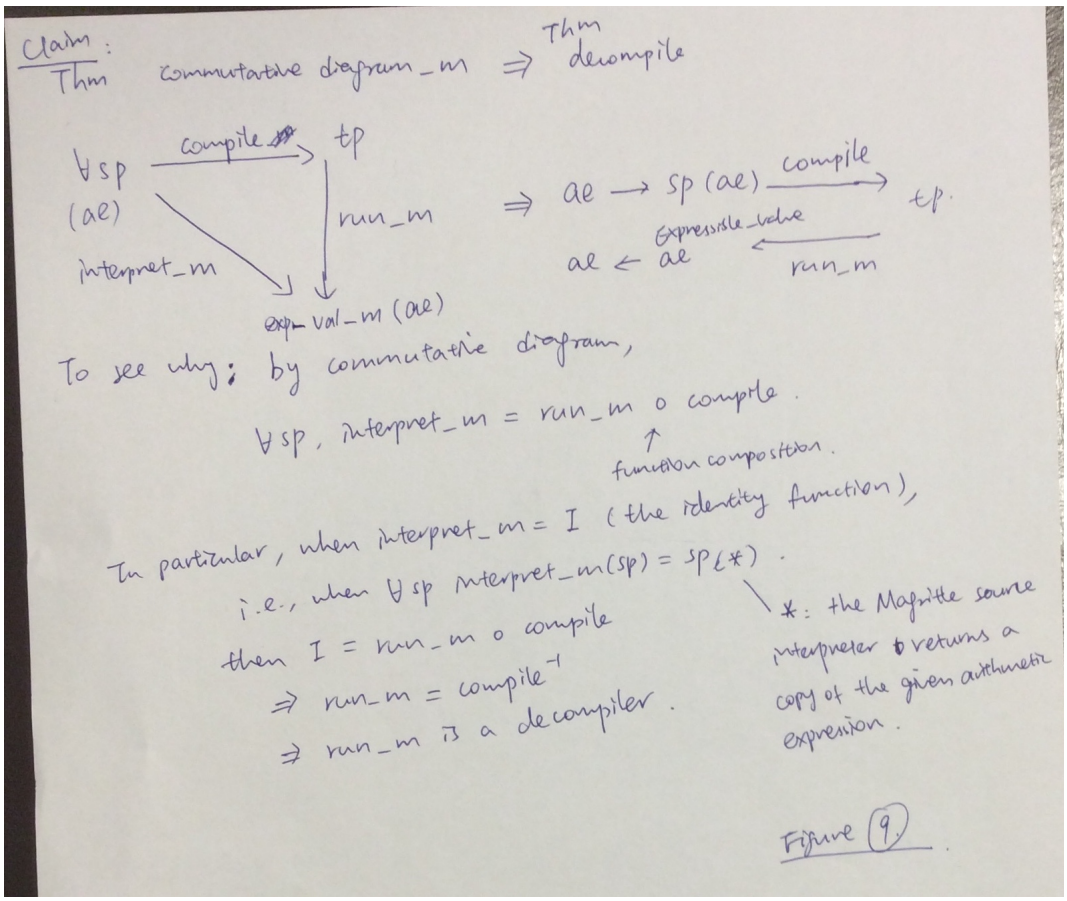


Figure 9: Figure 9

```
Proposition about_evaluate_m :
  forall ae : arithmetic_expression,
    evaluate_m ae = Expressible_nat_m ae.
Proof.
  induction ae as [ l | ae1 IHae1 ae2 IHae2 | ae1 IHae1 ae2 IHae2].
  - rewrite -> (fold_unfold_evaluate_m_Literal l).
    reflexivity.
  - rewrite -> (fold_unfold_evaluate_m_Plus ae1 ae2 ).
    rewrite -> IHae1.
    rewrite -> IHae2.
    reflexivity.
```

```
  - rewrite -> (fold_unfold_evaluate_m_Minus ae1 ae2).
    rewrite -> IHae1.
    rewrite -> IHae2.
    reflexivity.
Qed.

Theorem decompile :
  forall ae : arithmetic_expression,
    run_m (compile (Source_program ae)) = Expressible_nat_m ae.
(*
note to self:
run_m compile ae => ae
run_m is a left inverse of compile
run_m is a decompiler
*)
Proof.
  intro ae.
  Check (the_commutative_diagram_m (Source_program ae)).
  rewrite <- (the_commutative_diagram_m (Source_program ae)).
  unfold interpret_m.
  Check (about_evaluate_m).
  apply (about_evaluate_m).
Qed.
```

12.6 Some Concluding Remarks

12.6.1 A Simplification Not Yet Implemented

In retrospect we realized that the implementation can be simplified further: the message constructor can be removed since the Magritte source interpreter does not compute and thus does not produce error message. Due to time constraints this was not reflected in the previous subtasks, but we have instead proved that if we define the simplified type:

```
Definition Expressible_value_m' := arithmetic_expression.
```

and

```
Definition specification_of_evaluate_m' (evaluate_m' : arithmetic_expression ->
↪  arithmetic_expression) :=
  (forall n : nat,
     evaluate_m' (Literal n) = Literal n)
  /\
  (forall (ae1 ae2 ae1' ae2' : arithmetic_expression),
      evaluate_m' ae1 = ae1' ->
      evaluate_m' ae2 = ae2' ->
      evaluate_m' (Plus ae1 ae2) = Plus ae1' ae2')
  /\
```

```
  (forall (ae1 ae2 ae1' ae2' : arithmetic_expression),
      evaluate_m' ae1 = ae1' ->
      evaluate_m' ae2 = ae2' ->
      evaluate_m' (Minus ae1 ae2) = Minus ae1' ae2').


Fixpoint evaluate_m' (ae : arithmetic_expression) : Expressible_value_m' :=
 match ae with
  | Literal n =>
    (Literal n)
  | Plus ae1 ae2 =>
    match evaluate_m' ae1 with
    | ae1' =>
      match evaluate_m' ae2 with
      | ae2' =>
         (Plus ae1' ae2')
      end
    end
  | Minus ae1 ae2 =>
    match evaluate_m' ae1 with
    |  ae1' =>
      match evaluate_m' ae2 with
      | ae2' =>
          (Minus ae1' ae2')
      end
    end
 end.

Lemma fold_unfold_evaluate_m'_Literal :
  forall n : nat,
    evaluate_m' (Literal n) = (Literal n).
Proof.
  fold_unfold_tactic evaluate_m'.
Qed.

Lemma fold_unfold_evaluate_m'_Plus :
  forall (ae1 ae2 : arithmetic_expression),
    evaluate_m' (Plus ae1 ae2) =
    match evaluate_m' ae1 with
    | ae1' =>
      match evaluate_m' ae2 with
      | ae2' =>
        (Plus ae1' ae2')
      end
    end.
```

```
Proof.
  fold_unfold_tactic evaluate_m'.
Qed.

Lemma fold_unfold_evaluate_m'_Minus :
  forall (ae1 ae2 : arithmetic_expression),
    evaluate_m' (Minus ae1 ae2) =
    match evaluate_m' ae1 with
    | ae1' =>
      match evaluate_m' ae2 with
      | ae2' =>
        (Minus ae1' ae2')
      end
    end.
Proof.
  fold_unfold_tactic evaluate_m'.
Qed.


Theorem evaluate_m'_satisfies_the_specification_of_evaluate_m' :
  specification_of_evaluate_m' evaluate_m'.
Proof.
  unfold specification_of_evaluate_m'.
  split.
  - intro n.
    exact (fold_unfold_evaluate_m'_Literal n).
  - split.
    * intros ae1 ae2 ae1' ae2'.
      rewrite -> fold_unfold_evaluate_m'_Plus.
      intros H_ae1 H_ae2.
      rewrite -> H_ae1.
      rewrite -> H_ae2.
      reflexivity.
      * intros ae1 ae2 ae1' ae2'.
      rewrite -> fold_unfold_evaluate_m'_Minus.
      intros H_ae1 H_ae2.
      rewrite -> H_ae1.
      rewrite -> H_ae2.
      reflexivity.
Qed.

Definition specification_of_interpret_m' (interpret_m' : source_program ->
↪  source_program) :=
  forall evaluate_m' : arithmetic_expression -> arithmetic_expression,
    specification_of_evaluate_m' evaluate_m' ->
```

```
forall ae : arithmetic_expression,
  interpret_m' (Source_program ae) = Source_program (evaluate_m' ae).
```

12.6.2 Connections to Homomorphisms

(this subsection is inspired from one email exchange with Prof Danvy) There is an homomorphism between things and the representations of things. This is because taking an evaluation/computation at a point can be thought of as a homomorphism.

In this case, there are several homomorphisms:

- there exists a homomorphism between the natural numbers and the syntactic representations of natural numbers, i.e., the arithmetic expressions.
- there exists a homomorphism between the source programs and the representations of source programs, i.e., the target programs.
- there exists a homomorphism between the Magritte interpreter and the standard interpreter
- so perhaps it makes sense to also say that there exists a homomorphism from the Magritte world to the standard world (and the homomorphism is simply the function that performs the computation of addition/subtraction)

This increases the level of abstraction while preserving the structure of the binary operation (in this case addition and subtraction) since homomorphisms preserve the operators. This is nice because it enables us to instantiate the representations with many other things apart from merely natural numbers, allowing us to implement much more interesting things.

An attempt at visualizing the relations involved is shown in Figure 10.

The famous pipe. How people reproached me for it! And yet, could you stuff my pipe? No, it's just a representation, is it not? So if I had written on my picture "This is a pipe", I'd have been lying!
– René Margritte

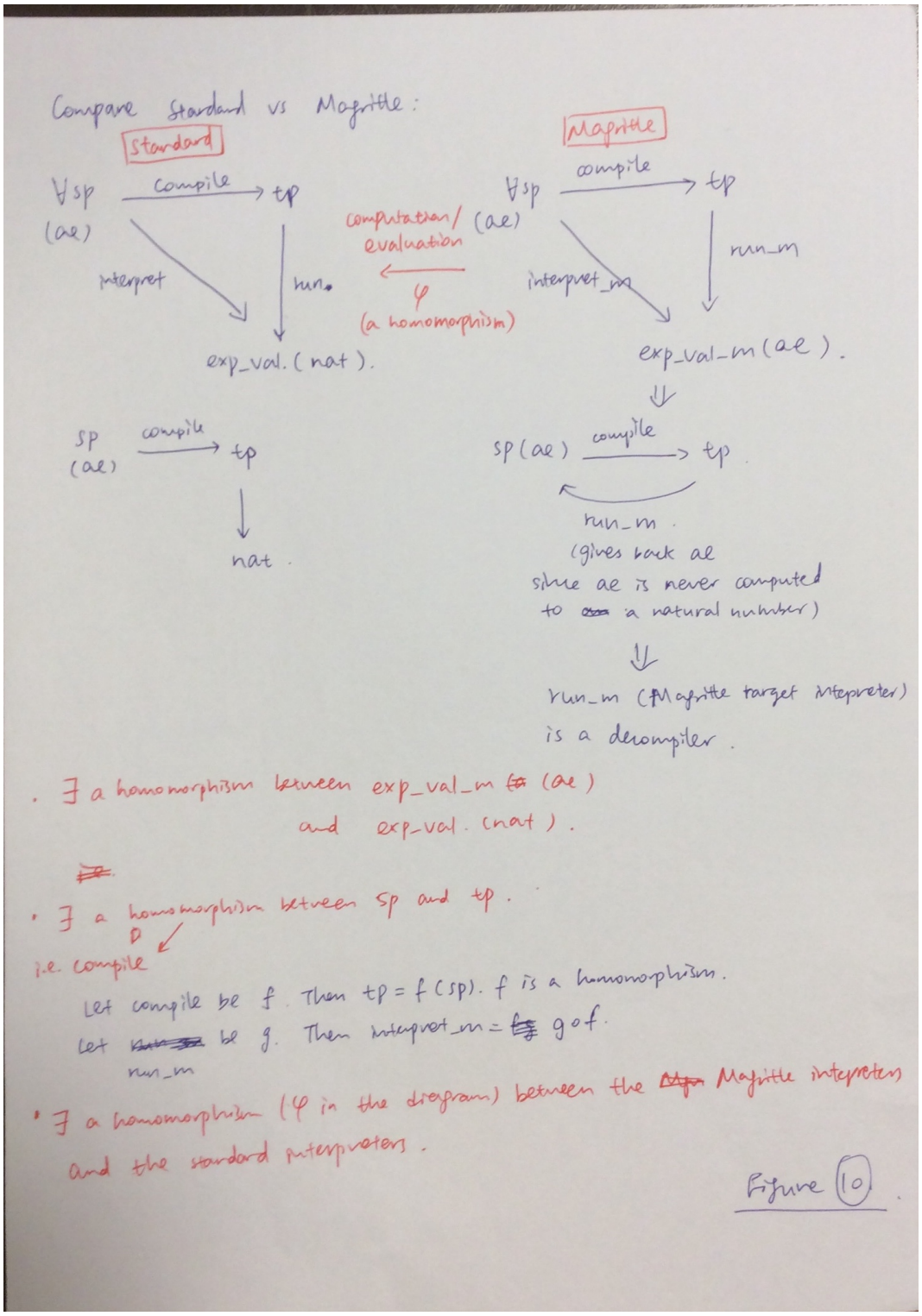


Figure 10: Figure 10

13 Possible extensions

There are two possible extension that unfortunately was not yet implemented at the time of this report:

13.1 The world seen from right to left

Although we do not have time to implement this, the basic idea is to change the order of evaluation for ae1 and ae2.

The right to left evaluate would have the following case structure:

- Base case: (Literal n)
- Induction case:
  - (i) (Plus ae1 ae2)
    - (a) evaluating ae2 gives a nat value and evaluating ae1 gives a nat value
    - (b) evaluating ae2 gives a nat value and evaluating ae1 gives an error message
    - (c) evaluating ae2 gives an error message
  - (ii) (Minus ae1 ae2)
    - (a) evaluating ae2 gives a nat value and evaluating ae1 gives a nat value and $n_2 < n_1$ is true (there is numerical underflow);
    - (b) evaluating ae2 gives a nat value and evaluating ae1 gives a nat value and $n_2 < n_1$ is false (there is NO numerical underflow);
    - (c) evaluating ae2 gives a nat value and evaluating ae1 gives an error message;
    - (d) evaluating ae2 gives an error message;

The right to left `decode_execute` would have the following case structure:

(i) PUSH n

In this case, we will always return OK and add n iteself to the data stack.

(ii) ADD

If there are only zero or one element in the stack, returns KO and an error message indicating stack underflow.

If there are two elements, returns OK and add the result of addition in the data stack.

(iii) SUB

If there are only zero or one element in the stack, returns KO and an error message indicating stack underflow (KO).

If there are two elements, check if $n_1 < n_2$. If yes, returns KO and an error message indicating numerical underflow. Otherwise, returns OK and add the result of subtraction in the data stack.

The right to left compiler (take the accumulator version for example) would now be:

```
Fixpoint compile_aux_alt (ae : arithmetic_expression) (a : list byte_code_instruction) :
↪  list byte_code_instruction :=
  match ae with
  | Literal n => PUSH n :: a
  | Plus ae1 ae2 => compile_aux_alt ae2 (compile_aux_alt ae1 (ADD :: a))
  | Minus ae1 ae2 => compile_aux_alt ae2 (compile_aux_alt ae1 (SUB :: a))
  end.
```

instead of

```
Fixpoint compile_aux_alt (ae : arithmetic_expression) (a : list byte_code_instruction) :
↪  list byte_code_instruction :=
  match ae with
  | Literal n => PUSH n :: a
  | Plus ae1 ae2 => compile_aux_alt ae1 (compile_aux_alt ae2 (ADD :: a))
  | Minus ae1 ae2 => compile_aux_alt ae1 (compile_aux_alt ae2 (SUB :: a))
  end.
```

13.2 The world seen in fold right

In intro to CS, we have implemented the fold right function for arithmetic expression. And using that, we define the fold right counterpart for `evaluate`,`fetch_decode_execute_loop` and `compiler`. It is not insensible that we wonder if we could potentially also implement AND prove everything in fold right version?

14 Conclusion

14.1 Problem solving reflections

- what and how

  Keep in mind of the "what" to guide the direction, so that we don't get lost in the sea of details, especially when the proof involves many cases and subcases. In this project in particular, it helps significantly if we have a mental picture of the overall structure of the proof when solving the problems. And this is especially nice when many tasks in this project actually share the same structure!

  Be knowledgeable and confident in the "how" aka the specific syntax and tactics to implement the ideas. This also provides a sense of comfort in knowledge and the inertia that we can fall back on.

- Keep to the standard workflow (like putting the recursive part in an aux functions, stating fold-unfold lemmas after a recursive definition, consistent naming and notations). With these habits we can avoid unnecessary distractions so that we can focus our attention on more important things.

- have a bag of tricks when proving

  Some examples encountered in the term project and other assignments include: light of inductil, proof by cases, and the Eureka lemmas.

14.2 The Term Project

This term project is a significant step in bridging the connections between mathematics and computer science. This project is the dual of the term project of the same title in Intro to CS. Back then, we can only test our implementations, but now we can prove that each functions satisfy their respective specifications. Even more significantly, we have proved the commutative diagram that we have only tested in Intro to CS. Eventually, we take a higher view from the Magritte world where nothing is computed and instead everything is a representation of something. The process of generalizing patterns is a recurring (perhaps even unifying) theme in the this course (and similarly in intro to CS). Thus the cover image for this term project is meant to be a departure point for further reflections on pattern, representation, and abstraction, among many more interesting things in the vast universe of computer science, the visible ones and the hidden ones.