Final Project Part 2: Interpreters, compilers, and virtual machines for arithmetic expressions

Oishik Ganguly

December 6, 2017

Contents

1 Introduction

In this section, we deal with a very simple calculator language (or arithmetic expressions) which represent addition and subtraction of natural numbers and errors related to these operations. The only error in the evaluation of these expressions is a numerical underflow, which concerns negative results of subtraction. Our focus in this section is to explore the two methods of evaluation of these expressions, namely (i) interpreting the expression (big step semantics) and (ii) compiling the expression into a byte code program which is them sequentially executed by a virtual machine (small step semantics), and most importantly, to prove that methods (i) and (ii) for evaluating the value of an expression are equivalent. In this report, each section will be dedicated to specifications for each of the programs discussed above (eg. interpreter, compiler, etc.), except for the final section, which deals with the commutative diagram for evaluation of expressions (i.e., the equivalence of interpretation, and compilation and virtual machine execution).

2 Evaluate and Interpret

The `specification_of_evaluate` defines a function that takes an arithmetic expression and recursively computes its final value. The soundness of the evaluate specification was easily proved, by induction over the given arithmetic expression, and subsequently considering case by case values of the sub-expressions of the `Plus _ _` and `Minus _ _` constructors.

Following this, we defined a recursive function that implemented this specification, `evaluate_v0`. We were, for want of time, unable to implement a continuation passing style evaluator. We then proved that this function met the `specification_of_evaluate`. This proof was even easier than the first, involving only using `evaluate_v0`'s unfold lemmas.

Next, we considered the specification for an interpreter, which essentially acts as a calling function to `evaluate`, after deconstructing the arithmetic expression that is passed to it wrapped in the `Source_program _` constructor. The soundness of the specification proof, the subsequent implementation (as `evaluate_v0`) , and the proof to show that `evaluate_v0` meets the specification are simple, and do not warrant discussion.

3 The virtual machine

Having explored an interpreter for arithmetic expressions, we next moved on to the virtual machine, which consists of a three-tier hierarchy. At the bottom of the hierarchy, executing a single instruction of byte code, is the `decode_execute` function. On top of this is a `fetch_decode_execute_loop` which loops thought the entire byte code instruction list, and calls the `decode_execute` function on single instructions. Finally, we have the `run` program, which accepts a byte-code program wrapped in the `Target_program _` constructor, and passes the byte-code list to the `fetch_decode_execute_loop`.

3.1 Decode and execute a single instruction

The `specification_of_decode_execute` specifies how each of the three types of byte code instruction, viz. push, add, sub, are dealt with. The proof for the soundness of the specification was direct, and involved using the hypotheses given by the specification. The implementation , `decode_execute`, and the proof that `decode_execute` satisfied the implementation was simple as well. A point of interest here was the addition of the errors regarding too few and too many values in the data stack. These errors were not dealt with when we were considering interpreters, and served as a significant obstacle when we were attempting to prove the commutative diagram.

3.2 Fetch the list of instructions and execute

Proving that the specification of the `fetch_decode_execute_loop` was sound was direct, and involved destructing the hypotheses of the specification after passing the `decode_execute` function to it. The implementation of the execution loop is standard as well, and named `fetch_decode_execute_loop`. To prove that this function meets the specification was direct as well: the only mental leap was to realize that the hypotheses provided by each conjunctive clause of the specification needed to be rewritten with the actual implementation of `decode_execute`. An example of this is as follows :

```
rewrite ->
        (specification_of_decode_execute_is_sound
           decode_execute_var
           decode_execute
           H_about_decode_execute_var

  decode_execute_satisfies_the_specification_of_decode_execute
           bci ds) in H_about_decode_execute_bci.
```

3.3 run

The proof for the soundness of `specification_of_run`, the implementation of the specification by a function name `run`, and the proof that `run` meets the given implementation were straightforward, and does not warrant discussion. However, once again, we find that the specification of run specifies errors that the interpreter does not throw. This disparity will be discussed extensively when considering the proof for the commutative diagram.

4 An interlude : Executing two byte code lists appended

This section, sandwiched between the sections for `fetch_decode_execute_loop` and `run`, deals with an interesting theorem concerning the execution of two appended byte code lists. The theorem is as follows :

```
Theorem
  relation_between_execution_of_two_bcis_and_their_appended_version :
forall (bci1s bci2s : list byte_code_instruction),
  (forall (ds : data_stack)
          (ds_new : data_stack),
      fetch_decode_execute_loop bci1s ds = OK ds_new ->
      fetch_decode_execute_loop (bci1s ++ bci2s) ds =
      fetch_decode_execute_loop (bci2s) ds_new) /\
  (forall (ds : data_stack) (s : string),
      fetch_decode_execute_loop bci1s ds = KO s ->
      fetch_decode_execute_loop (bci1s ++ bci2s) ds = KO s).
```

This theorem was essential to proving the commutative diagram and we struggled to prove this for a while. In particular, we were stuck at the following proof step in the inductive case of the proof :

```
i' : byte_code_instruction
is' : list byte_code_instruction
bci2s : list byte_code_instruction
IH_is' : forall ds ds_new : data_stack,
         fetch_decode_execute_loop is' ds = OK ds_new ->
         fetch_decode_execute_loop (is' ++ bci2s) ds =
  fetch_decode_execute_loop bci2s ds_new
ds : data_stack
ds_new : data_stack
H_when_bcis_is_cons : fetch_decode_execute_loop (i' :: is') ds = OK
  ds_new
============================
 fetch_decode_execute_loop (i' :: is' ++ bci2s) ds =
  fetch_decode_execute_loop bci2s ds_new
```

It took us a while to realize that we needed to consider the different cases of `(decode_execute i' ds)`, this allowed us to unfold the left hand side and obtain an expression similar to the L.H.S of the conclusion of the inductive hypothesis. We then used the case by case value of the above expression to rewrite the inductive hypothesis as well, and finally applied it to the hypothesis that the goal provided to obtain the required equality.

5 Auxiliary compiler and compiler

In this section, we consider the compiler that compiles a given arithmetic expression to a corresponding byte code instruction list.

The auxiliary compiler specification provides a mapping between specific arithmetic expression constructors and their byte code equivalents. The proof for the soundness of this specification was direct.

The implementation of the auxiliary compiler warrants more discussion. We implemented a standard non-accumulator based recursive auxiliary compiler, `compile_aux_v0`, that traverses left and right sub- expressions of a given arithmetic expressions, converts these sub-expressions to byte code lists, and appends the lists. That this function meets the given specification is simple to prove as well. However, we also implemented an accumulator-based auxiliary compiler, `compile_aux_v1`, implemented as follows :

```
Fixpoint compile_aux_v1 (ae : arithmetic_expression)
         (acc : list byte_code_instruction) : list byte_code_instruction
  :=
match ae with
| Literal n =>
  (PUSH n) :: acc
| Plus ae1 ae2 =>
  compile_aux_v1 ae1 (compile_aux_v1 ae2 (ADD :: acc))
| Minus ae1 ae2 =>
  compile_aux_v1 ae1 (compile_aux_v1 ae2 (SUB :: acc))
end.
```

Of course, `compile_aux_v1` will not meet the specification of `compile_aux`, since it has an accumulator. However, a compile function that calls it will still satisfy the `specification_of_compile` [1]. To prove this, we made use of the following master lemma :

```
Lemma master_lemma_about_compile_aux_v1 :
 forall (ae : arithmetic_expression)
         (acc : list byte_code_instruction),
```

[1]The soundness of this specification, an implementation of the specification using the non-accumulator based auxiliary compiler, and a proof that this implementation satisfies the specification is easily proved.

```
  compile_aux_v1 ae acc =  (compile_aux_v0 ae) ++ acc.
```

This relation between `compile_aux_v0` and `compile_aux_v1` is intuitive; the accumulator obviously contains the byte code for all sub-expressions that have thus far been compiled. If at this stage, we were to switch to `compile_aux_v0`, then it would compile the remaining expression, `ae`, and then we would append the list so obtained to the accumulator. The proof for this master lemma proceeds by induction over arithmetic expressions and is parameterized by the accumulator value.

With this lemma obtained, proving that `compile_v1` satisfies the specification of compile was straightforward. When we arrived at the following stage of the proof :

```
compile_aux_var : arithmetic_expression -> list byte_code_instruction
H_about_compile_aux_var : specification_of_compile_aux compile_aux_var
ae : arithmetic_expression
==============================
 Target_program (compile_aux_v1 ae nil) = Target_program (
 compile_aux_var ae)
```

we simply used the master lemma to replace `compile_aux_v1` with `compile_aux_v0`. The proof proceeds directly from here.

6 The commutative diagram

With the functions for interpreting, compiling, and executing specified, implemented, and properties thereof proved, we finally deal with the theme of this section of the final project : the commutative diagram.

As explained in the introduction, the commutative diagram deals with the equivalence of interpreting an arithmetic expression, and running a compiled version of the expression on a virtual machine. The statement of the commutative diagram theorem is as follows :

```
Theorem the_commutative_diagram :
  forall sp : source_program,
    interpret_v0 sp = run (compile_v0 sp).
```

To prove this, we initially unfolded all the non-recursive function implementations, and proceeded with an induction over the given arithmetic expression. However, we soon ran into a hurdle, namely, that compiling and running the expression gave more possible errors than the interpreter did for the same expression. This point was raised a number of times in earlier sections of this write-up, and we finally ran head on into the problem. From a purely mathematical standpoint, the solution was clear : if the compiled expression, when run, gave errors concerning too many or too few elements on the stack, then the expression provided in the first place could not have been a valid arithmetic expression. For instance, a byte code list where an `ADD` command is followed by just one natural value corresponds to an expression of the form `Plus (val)`, which is clearly an invalid arithmetic expression. However, we could not figure out a way of using to coq to state the `ae` $\notin$ `arithmetic_expression`.

Consequently, we pursued a different path and defined a master lemma as follows :

```
Lemma master_lemma_for_commutative_diagram :
 forall (ae : arithmetic_expression)
        (ds : data_stack),
     fetch_decode_execute_loop (compile_aux_v0 ae) ds =
     match (evaluate_v0 ae) with
     | Expressible_nat num =>
      OK (num :: ds)
     | Expressible_msg err_msg =>
       KO err_msg
     end.
```

This lemma salvaged us from the disparity in the number of errors. For now, the evaluation of some expression is *appended* to the head of the possibly erroneous data stack, thus giving the same types of error on the left and right hand sides. Thought about in another way, this lemma also follows from intuitive reasoning; if we have some sub-expression that has already been compiled and run, the result of which is stored in the data stack, then the final expressible value should be the result of evaluating the remaining part of the expression (`ae` here) and appending the result to the head of the data stack.

The proof for this lemma is long, and involves an induction over arithmetic expressions. The proof for the base (literal) case is straightforward. For the addition and subtraction cases, we must consider the different cases of `evaluate_v0 e1` and `evaluate_v0 e2` to see the proof though. For instance, here are two successive steps of the proof for the plus case, where considering cases for the first of the above mentioned expressions greatly simplifies the goal. First we have :

```
e1 : arithmetic_expression
e2 : arithmetic_expression
IH_e1_plus : forall ds : data_stack,
             fetch_decode_execute_loop (compile_aux_v0 e1) ds =
             match evaluate_v0 e1 with
             | Expressible_msg err_msg => KO err_msg
             | Expressible_nat num => OK (num :: ds)
             end
IH_e2_plus : forall ds : data_stack,
             fetch_decode_execute_loop (compile_aux_v0 e2) ds =
             match evaluate_v0 e2 with
             | Expressible_msg err_msg => KO err_msg
             | Expressible_nat num => OK (num :: ds)
             end
ds_for_lemma : data_stack
```

```
  ============================
   fetch_decode_execute_loop (compile_aux_v0 e1 ++ compile_aux_v0 e2 ++ ADD :: nil)
     ds_for_lemma =
   match
     match evaluate_v0 e1 with
     | Expressible_msg s => Expressible_msg s
     | Expressible_nat n1 =>
         match evaluate_v0 e2 with
         | Expressible_msg s => Expressible_msg s
         | Expressible_nat n2 => Expressible_nat (n1 + n2)
         end
     end
   with
   | Expressible_msg err_msg => KO err_msg
   | Expressible_nat num => OK (num :: ds_for_lemma)
   end
```

following which, implementing the following tactic :

```
case (evaluate_v0 e1) as [err_msg_e1 | nat_val_e1] eqn : val_of_eval_v0_e1.
```

we get :

```
  e1 : arithmetic_expression
  e2 : arithmetic_expression
  err_msg_e1 : string
  val_of_eval_v0_e1 : evaluate_v0 e1 = Expressible_msg err_msg_e1
  IH_e1_plus : forall ds : data_stack,
               fetch_decode_execute_loop (compile_aux_v0 e1) ds = KO err_msg_e1
  IH_e2_plus : forall ds : data_stack,
               fetch_decode_execute_loop (compile_aux_v0 e2) ds =
               match evaluate_v0 e2 with
               | Expressible_msg err_msg => KO err_msg
               | Expressible_nat num => OK (num :: ds)
               end
  ds_for_lemma : data_stack
  ============================
   fetch_decode_execute_loop (compile_aux_v0 e1 ++ compile_aux_v0 e2 ++ ADD :: nil)
     ds_for_lemma = KO err_msg_e1
```

From here on, the proof for this particular goal simply involves using the induction hypothesis (or hypotheses for subsequent goals). Thus, addition has 3 sub cases, and subtraction (owing to the numerical underflow possibility), has 4 sub cases.

With this lemma defined, we could grab the theorem `the_commutative_diagram` by its horns, and take it down in no more that 10 lines. The proof was straightforward, and makes use of the master lemma.

7 Conclusion

This section of the final project has, in many ways, been a completion of an exploration that started in Fundamentals of Programming, of interpreters, compilers, and virtual machines. We have not just unit tested, but *proved* that these programs do what they are intended to do, and shown that relations between them hold for certain. Perhaps it would be remiss of me to say that this project is the conclusion– after all, for want of time, we could not explore the Magritte compiler, the byte code verifier, and the continuation passing style interpreter. And of course, this is only the start of an exciting and enlightening journey into software verification proofs.

We have learned much from this project, not just about coq, but also about writing proofs, the nature of recursion, the pitfalls of accumulators, and powers of master lemmas. To conclude, this project made us aware of two things, one uplifting, and another demoralizing, namely, that computer programs can be tamed by the power of logic, mathematics, and proofs, but that doing so requires an incredible amount of effort, as evidenced by the fact that to prove the properties of a simple calculator language with only two arithmetic operations took almost 2000 lines of code!