YSC3236: Functional Programming and Proving

Term project

A language processors for arithmetic expressions

Karolina Bargiel

November 29th, 2019

Table of contests

Introduction

The following project focuses on the representation on virtual machine in the Coq Proof Assistant. The main goal of the project is to formalize an interpreter for arithmetic expressions, a compiler from arithmetic expressions to byte-code, and an interpreter for byte-code. Additionally, we also prove that for any given arithmetic expression, interpreting an arithmetic expression and compiling this arithmetic expression, and then running the resulting byte-code program yields the same result, be it a natural number or an error message.

The below discussion is a formalization of the project which was conducted in Introduction to Computer Science in the previous year.

In the first chapter we focus on the implementation of the evaluate and interpret function. We will also formally show that our implementation meets the given specification, and that there is at most one function that satisfies the specifications.

In the second chapter our focus is on decode_execute and fetch decode_execute functions alongside with its byte-code properties.

In the third chapter we study the virtual machine for byte-code instruction.

In the fourth chapter discover the nature of compiler, with its multiple implementations both with and without acumulator.

In the fifth chapter we have a closer look at the capstone property which is a main formal property to be shown in this project.

In the sixth chapter we implement the function verify with all of its properties.

Finally, we try to combine all of the above debate with Surrealism, by showing the Magritte implementation of the virtual machine functions.

Chapter 1 Evaluate and interpret

In this chapter we have a closer look at the source interpreter and evaluate function.

Before we do so lets first introduce the source language of arithmetic expression which is used throughout this project. When we refer to arithmetic expression we mean an inductive type which can be either a literal representation of natural number, an addition, subtraction, multiplication or division. By source program we understand a type which in its constructor takes arithmetic expression and returns source program. By expressive value we mean either an expressible nat which takes a natural number and returns an expressible value or expressible message which takes string and return expressible value.

Equipped with the above we can move to introducing the specification of the evaluate. These specifications were provided in the statement of the project. Evaluate is a function which takes an arithmetic expression and returns the expressible value. Depending on the arithmetic expression, the function can return either Expressible_nat - which is a type declared from natural number n to expressible_value, or expressible message which is a type declared from string to expressible_value. The function is recursive. In the below reasoning we will take under consideration five cases that evaluate can handle. Those are:

- Literal n - when the given arithmetic expression is just a representation of a natural number.

- Plus - which adds two arithmetic expressions. In this case we check if the first value is a natural number. If it is not, the second value will not be evaluated and an error message in a form of expressible message. If the first value is evaluated as a natural number, then the program evaluates the second argument— if it's a natural number, then the arithmetic expression will be evaluated (into expressible nat), and if it's not, then an error message will be returned. This process of checking in both values are numbers is applied in all of the following operations.

- Minus - which subtract the first arithmetic expression from the other. This case also takes under consideration the error message which is displayed in case the output of the subtraction of two natural numbers will be negative.

- Multiply- which multiplies two arithmetic expressions.

-Divide- which divides arithmetic expression, in this case we also produce the error message in case we divide by 0.

Our debate starts by proving that there is at most one function that satisfies the specification of evaluate. As we remember from the mid term project this proof leads us to the conclusion that all of the implementations that satisfy the same specifications are structurally equivalent. We try to show that :

Proposition there_is_at_most_one_specification_of_evaluate :
forall f g,
specification_of_evaluate f ->
specification_of_evaluate g ->
forall e,
f e = g e.

We conduct the above proof by induction on e which is our arithmetic expression. E can evaluate to 5 cases literal n, plus, minus, multiply and divide.

induction e as [n| ae1 IHae1 ae2 IHae2| ae1 IHae1 ae2 IHae2| ae1 IHae1 ae2 IHae2| ae1 IHae1 ae2 IHae2].

The base case gets solved by rewriting the correct pair of specifications both in f and g cases. With all of the induction steps we include two different arithmetic expressions in each of them. We are conducting proof by case within each operator. In each case we take under consideration whether arithmetic expressions were destructed into natural numbers and strings. Additionally, in divide and minus we also enumerated cases where errors occur (e.g. is n2 zero or not?) In error cases in minus and divide, we had to inject into the hypothesis, and then substitute into condition and only then can we discriminate because the contradiction becomes obvious. As we can observe, the proof uses familiar tactics such as injection, discrimination and use of induction hypothesis.

After the above discussion we can now implement the direct version of evaluate. This function as mentioned above is a recursive function which takes an arithmetic expression and returns an expressible value. Effectively it evaluates the given expression into either the result of the given operations or the error message, which will indicate what made the function fail to obtain the result. We match given arithmetic expression with one of 5 cases literan, plus, minus, multiply and divide. Then we call recursively the function on arithmetic expressions which are nested in ae, and if the error occurred such as one of the numbers is not implemented, n1<= n2 in case of minus or we try to divide by 0, we return expressible message. Otherwise we get the expressible nat as the final output.

Under the fixpoint definition we can find accompanying fold unfold lemmas for our five operations. Fold unfold lemmas can be used as a reusable tool to proof more complex proofs such as a problem of whether our implementation satisfies a certain specification.

We sum up our debate about evaluate with a proof that evaluate satisfies its specification. We show that:

Theorem evaluate_satisfies_its_specification:
specification_of_evaluate evaluate.

We will conduct this proof by cases. We first start by unfolding specification of evaluate and then we split the given equation. This lets us focus on a primary goal which can be solved using fold unfold lemma for literal case and reflexivity. Then we continue to split and we use other fold unfold lemmas accordingly to the case we currently are in. In the parts where our goal contains induction, for example for the second subgoal:

evaluate ae1 =
Expressible_msg s1 ->
evaluate
(Plus ae1 ae2) =
Expressible_msg s1

We introduce the hypothesis H, which will be the left part of induction and then solve the equation by rewriting the hypothesis (or hypotheses) into the main goal. The repetition of that tactic summarizes the proof. Formally shown that the specification satisfies, let us skip the step of writing unit tests. Unit testing only show us the presence of bugs not its absence while the above proof ensures that the implementation is correct.

The second function which we have the pleasure to discuss in this chapter is an interpreter. Specifications of this function are defined as follows:

Definition specification_of_interpret (interpret : source_program -> expressible_value) :=
forall evaluate : arithmetic_expression -> expressible_value,
specification_of_evaluate evaluate ->
forall ae : arithmetic_expression,
interpret (Source_program ae) = evaluate ae.

In words, interpret is a function which takes a source program and returns the espressible value.

Similarly as described above we begin the reasoning by proving that there is at most one function that meets the specification of interpreting :

Proposition there_is_at_most_one_specification_of_interpret :
forall f g,
specification_of_interpret f ->
specification_of_interpret g ->
forall sp : source_program,
f sp = g sp .

We conduct the proof by firstly introducing f and g, then we unfold the specification of interpret, and introduce naiming for the specification S_f and S_g. We also introduce ae. Here for the first time a new way of introduction variables appear. As we know that in the constructor of source program ae is the input we introduce ae in square brackets . This technique let us “unfold” the constructor in a way that in case of sp which states for source program we can work with Source_program ae. After that we conduct the two below lines:

rewrite -> (S_f evaluate evaluate_satisfies_its_specification ae).
rewrite -> (S_g evaluate evaluate_satisfies_its_specification ae).

These lines provide to the specification of evaluate the function evaluates satisfies its specification. Effectively they rewrite our subgoal to the form:

evaluate ae = evaluate ae

Which can be solved by reflexivity. Below that, we implement the definition of interpret directly.

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

In the final part of this chapter, similar to what we did for evaluate, we will prove the theorem that interpret_satisfies_the_specification_of_interpret.

In this proof we also use the evaluate function. After unfolding and introducing all the variables we are left with the equation:

```
1 subgoal
evaluate1 : arithmetic_expression ->
expressible_value
S_evaluate1 : specification_of_evaluate
evaluate1
ae : arithmetic_expression
_____________________________________(1/1)
interpret
(Source_program ae) =
evaluate1 ae
```

Now we need to substitute evaluate1 with our implementation of evaluate. Fortunately, we can easily do it due to the fact that at the beginning of this chapter we showed that there_is_at_most_one_specification_of_evaluate, which means that all the implementations are equivalent. So, in the below line we use the above described position to substitute evaluate1 with evaluate.

rewrite -> (there_is_at_most_one_specification_of_evaluate evaluate1 evaluate S_evaluate1 evaluate_satisfies_its_specification ae).

That leaves us with:

Interpret (Source_program ae) = evaluate ae

Here we are one step from reflexivity. The last part only requires us to unfold interpret, which finishes the proof.

The above functions are the first step to our capstone proportion which will be further described in the project. During the development process we decided not to implement unit tests, as we are sure that our implementations of functions are correct, because we formally proved that they meet the specification.

If we were about to implement the two above functions in reverse, then we would only have to change the order of match statements in the evaluate function. If that was implemented we can easily show that those two versions commute with each other. Unfortunately, we were not able to show that here formally due to the limited time, but it is worth acknowledging that no matter from which side we will evaluate our statement, the error message will appear eventually. Even though the expressible natural value will always give the same result no matter if we evaluate from left or from the right side the error message can differ. Let's consider the example of arithmetic expression:

Plus ((Divide (Literal 3) (Literal 0)) (Minus (Literal 3) (Literal 4))

Above if evaluated left to write will give us the error message “division by 0”, while evaluated right to left “subtraction gives a negative number “. As we can see, even though the two implementations commute with each other they are not structurally equivalent as they are some cases when the result of computation will be different.

Chapter 2 Processor of byte-code instructions

In this chapter we will focus on the implementation and properties of functions decode_execute and fetch_decode_execute_loop.

Before we jump into the implementations, let's first discuss the new declared types that will be used throughout the rest of the project. The byte-code instruction is defined inductively as a type which consists of 5 cases: PUSH, which takes a natural number and returns byte-code instruction and ADD, SUB, MUL, DIV which are byte-code instructions themselves. This type will be used in a target program constructor and enables us to push the given instructions onto a stack. Additionally, the .v file also has a definition of data stack which is a list of natural numbers, as well as the result of decoding execution which can either evaluate to OK (given data stack) or KO (given string). OK will be returned when there are enough variables on the data stack to perform the execution, while KO will return an error message in case some variables are missing or we are trying to make an operation which cannot be executed (e.g. division by 0).

Now equipped with above we can define a decode execute function. This is a direct implementation of given specifications. Decode execute is a function which takes bytecode instruction and returns above defined result of decoding execution. The function matches bcis either with PUSH n - and then returns the approval message OK with n added to the data stack. As we are dealing with the stack list the n will be pushed on the head of the list. Otherwise bcis is matched with one of the other operation names. Within each operation (ADD, SUB, MUL, DIV) it matches the given data stack either with an empty list, the list with one natural number as an argument or two natural numbers pushed at the head of data stack.

Below the implementation we can find a theorem that decode execute satisfies its specification. This theorem can ensure that our implementation is correct as it meets the specification given. This is a formal way of showing the correctness, however just as a step we also computed multiple examples. Nevertheless, these just show that our function gives the expected output, not that the implementation is fully correct.

The proof is mechanical, and we use the same old tactics such as intro, rewrite, split, and reflexivity.

The final step to be conducted is to show that there is at most one function that satisfies the specification of decode execute. This property will ensure us formally that all of the future implementations of decode_exdcute are actually structurally equivalent to the one described above.

This proof uses the familiar tactic of introducing functions f and g which we assume are equal. The proof is conducted by induction over byte-code instruction. Then we conduct all the possible cases using destructure tactic. We apply destruct on the specification and then depends on the case we name only the hypothesis which is required in the current subgoal. As an example:

destruct S_decode_execute_f as [_ [[H_f_plus_nil [_ _]] [_ [_ _]]]]

We do this step as the specification of execute consists of many cases, and as we conduct the current proof by case, in each subgoal we only need a specific hypothesis. So, based on the piece of code above, in that

case we will only name H_f_plus_nil , which is a case where we match byte-code instruction with Plus applied on nil and the rest can be unnamed.

The above tactic combined with mechanical rewrite statement finishes the prove. By proofing the above we showed that all of the implementations of decode execute are equivalent.

In the next part of this chapter we will focus on fetch decode execute loop function. This function has a mirror functionality to the one described above. Instead of operating on one byte-code instruction it operates on the list of byte-code instructions. That enables us to decode execute longer instructions, as we can push the whole list with multiple operations onto the loop.

Fetch_decode_execute_loop is a recursive function which calls decode_execute inside it.

The implementation presented is a direct implementation form definition. In the base case we match empty lists with OK message which returns data stack. In the induction step we match the new elempt of the list with the reminder and the call a decode execute function.

The above definition is followed by accompanying fold unfold lemmas, as always when we deal with recursive functions.

Similar to the above we will show that fetch decode execute loop satisfies its specification, which will formally check if our definition is correct. The proof is mostly mechanical and it consists of familiar tactics. The only creative moment is to destruct the decode_execute bci ds to ds_OK and ds_KO. This step is a formal deconstruction of the second match statement and let us look separately at the cases of : recursive call, which returns an OK message and passes the data stack, and error message KO with string as an argument. The other tactic worth mentioning is the substitution of decode_execute' by decode_execute using that there is most on decode_execute_function which was proved earlier in this chapter.

Equipped with the above we can now decode and execute not just an individual byte-code instruction, but the list of those with respect to all the possible errors that might occur. Moving on to the properties related to fetch decode execute loop we can look into the property which speaks about byte-code append, which states that executing the concatenation of bcis1 and 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.

```
Proposition byte_code_append_property :
forall (bcis1 bcis2 : list byte_code_instruction) (ds : data_stack),
fetch_decode_execute_loop (bcis1 ++ bcis2) ds = match (fetch_decode_execute_loop bcis1 ds) with
| OK ds' => fetch_decode_execute_loop bcis2 ds'
| KO s => KO s end.
```

To accomplish this proof we will use two fold unfold lemmas for appending lists of instructions. One refers to nil case and states that nil concatenated on the left with the list is neutral and the second one is a cons case which states that the connection of two lists is equivalent to concationtio of the tail of the first list with the second list and then adding the head. These lemmas comes in handy in the later reasoning.

The rest of the proof is mechanical. The above function returns the stack of type resut_of_decoding_execution which will be used in later reasoning.

Chapter 3 Virtual machine of byte-code instructions

In this chapter we will study a virtual machine for byte-code instruction. Virtual machine is a step which follows our elaboration from the previous chapter. This program runs the stack which is of type:

result_of_decoding_and_execution, which can be obtained from fetch_decode_ececute_loop_function and turns it into an expressible value. In the .v file we named this function run. Run is a function that takes a target program as an input and returns expressible value - which is an expressible natural number or expressible message. The direct implementation of this function based on the provided specification uses a match statement applied on the target program. As mentioned in the first chapter, target program is a constructor that takes a list of byte-code instructions and turns it into target program.

```
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.
```

As we can see from the above implementation run returns a natural number only if the result of fetch decoding is expressed as an OK message with n as the head of the list and nil as its tail. That execution implies that fetch decode execute was able to execute the byte-code instruction. In all of the other cases the error message is returned. Now to confirm that our implementation is correct we formally show that the function meets its specification. This proof has been conducted many times in the previous chapters so I will not go into the details of its explanation.

After that we move to the part where we formally show that all the implementations of function run are equivalent. This proposition will ensure that all of the future implementations of run are equivalent to the one we implemented. The proof is conducted by cases. We can distinguish 4 main parts of what happens if stack is empty, if stack has only one argument, if stack has more arguments and if there is a string instead of a natural number.

```
Proposition there_is_at_most_one_specification_of_run :
 forall f g,
 specification_of_run f ->
 specification_of_run g ->
 forall (tp : target_program ),
 f tp = g tp.
```

By conducting the above we obtained a processor for byte-code instructions, which is an exact background we needed to show the capstone property in the next chapter.

Chapter 4 Compile

In the following chapter we will focus on the compile function and its different implementations. A compiler is a special program that processes source programs and turns them into target programs. In our debate that means that the arithmetic expression given in source program is compiled into byte-code instruction in the target program, for example given Divide (Literal n1) (Literal n2) , it will return DIV :: PUSH n1 :: PUSH :: n2:: nil.

Let's begin the discussion about implementation by introducing the specification of function compile aux which is an auxiliary version of the compile. Compile aux is a recursive function which takes an arithmetic expression and returns list of bytecode instructions. The byte-code instruction depends on the arithmetic expression that has been given. If we deal with literal n then the function will return PUSH n :: nil, otherwise we will recursively call the function on two arithmetic expressions and then concatenate the obtained results with each other and with the name of the operation (ADD, SUB, MUL or DIV).

Below the specification in the .v file we can find a direct implementation of a given function. The fixpoint contains of match statement which matches ae1 as described above. We also provided accompanying fold_unfold lemmas for five cases of arithmetic expressions.

Now in order to check if our implementation is correct we will prove that compile aux satisfies the specification of compile aux. To wit,

Theorem compile_aux_satisfies_its_specification :
specification_of_compile_aux compile_aux.

The proof of this theorem is mostly mechanical and is conducted as a proof by case. We primarily unfold specification of compile aux and successfully split the statement until we achieve a subgoal which can be extracted using one of the fold unfold lemmas. By repeating the above for all the cases we can accomplish the proof.

As usual to ensure that all the implementations are structurally, we show that there is at most one compile aux function. We conduct the above proof similar to the one in the previous chapter, by using the induction on ae1 (arithmetic expression) and taking under consideration all 5 possible cases. Then by destructing the specification we can obtain a specific hypothesis which can be rewritten. That step combined with the use of induction hypothesis finishes the proof.

Now equipped with the auxiliary version of the function let’s try to define the compile itself. According to specification compile is a function that translates the source program into target program. In simple words, in the environment of the computer science world compiler is a function that translates from the human understandable language of arithmetic expressions into machine code which is expressed as a list of byte-code instructions.

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

As usual we will test if our implementation is correct by formally showing that it meets the given specification.

This proof uses the familiar tactic of substituting one implementation with the other using the theorem that there is at most one function that satisfies the specification for compile aux. Here we can observe how this property come to use in action. As we know that all of the implementations are equivalent we can substitute the compile_aux' with the compile_aux.

rewrite (there_is_at_most_one_compile_aux_function compile_aux compile_aux' compile_aux_satisfies_its_specification H_compile_aux ae).

The above followed by reflexivity finishes the proof.

Now we can show that there is at most one compile function, to ensure that all of the implementations are equivalent. We conduct this proof in a standard way, primarily we introduce the name of the function f and g and then the two specifications. We also name arithmetic expression ae which is used in the constructor of source program. That leaves us with the statement :

f (Source_program ae) = g (Source_program ae)

As in our goal window we can observe that the specification of compile requires the implementation of compile aux, in order to rewrite, we need to provide it with one. In the following lines we then write S_f_compile with the compile aux and the theorem that compile aux satisfies its specification an ae as an argument . This operation applied both on function f and g finishes the proof.

In the remaining part of the chapter we will copy the above reasoning, however this time we will try to implement the compile function using acumulator. We will try to show that both implementations are structurally equivalent and can serve us as the "translator" between source and target programs.

Let's begin by implementing a function complie_aux' which is a recursive fixpoint which takes an arithmetic expression and list of byte-code instructions as arguments. The second one is going to be our acumulator. Similar to what we saw at the beginning of the chapter we match ae with 5 different cases . If ae is a Literal representation of natural number we return PUSH n :: bcis. Otherwise we will match the ae with either Plus, Minus, Multiply or Divide. In all of those cases we will match the input with recursive call

compile_aux' ae1 (compile_aux' ae2 (_ :: bcis)),

where "_ " is the respective name of the operation (ADD, SUB, MUL or DIV). Below the above described implementation, as usual a fold unfold lemmas can be found for all 5 cases.

Now equipped with the new version of compile aux, let's rewrite the definition of compile itself :

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

As we can observe compile' is almost the exact copy of compile. The only difference is in the compile aux call this time we refer the the acumulaotr version, and we pass nil as the value of the original acumulator.

Below the above definitions we introduced the lemma about_compile_aux'_and_compile_aux. This lemma is helpful in later reasoning about compiler'. The lemma states that

```
forall (ae : arithmetic_expression)
(bcis : list byte_code_instruction),
compile_aux' ae bcis = compile_aux ae ++ bcis.
```

In words the acumulator version of compile aux applied on arithmetic expression ae and list of byte-code instructions an accumulator is equivalent to non-acumulator version applied on ae concatenated with a list of byte-code instructions.

We prove the above statement by induction on ae. In the base case we use fold unfold lemmas for literal both about compile aux and compile aux '. We also make use of the property append cons and append nil which has been described in the previous chapters. In the first induction step we also make use of fold unfold lemmas. After that we rewrite the induction hypothesis 1 with (compile_aux' ae2 (ADD :: bcis)) as an argument and induction hypothesis 2 with (ADD :: bcis) as an argument. That leaves us with the goal :

compile_aux ae1 ++compile_aux ae2 ++ ADD :: bcis =(compile_aux ae1 ++compile_aux ae2 ++ ADD :: nil) ++bcis

At this stage we make use of the built in property List.app_assoc which states that :

```
forall (A : Type)
(l m n : list A),
l ++ m ++ n = (l ++ m) ++ n
```

That tactic combined with fold_unfold_append and fold_unfold_nil leads us to reflexivity. The induction steps for minus, multiply and divide are an exact copy of the reasoning introduced for plus.

Now the final property to be proved in this chapter is the theorem that compile' satisfies the specification of compile. This proof will ensure that the implementation with accumulator plays the same role as the one without, so they are structurally equivalent. In this proof the above lemma comes in handy. After unfolding

the specifications and the function itself and naming all components we can conduct our standard tactic of substituting the unknown implementation with the one we provide on the complie0 (see previous chapters).

Target_program (compile_aux' ae nil) = Target_program (compile_aux ae)

Here we can rewrite our master lemma, which talks about the equivalence of the auxiliary versions of compile. Then using built in function List.app_nil_r which states that the concatenation of nil in the left is neutral finishes the proof.

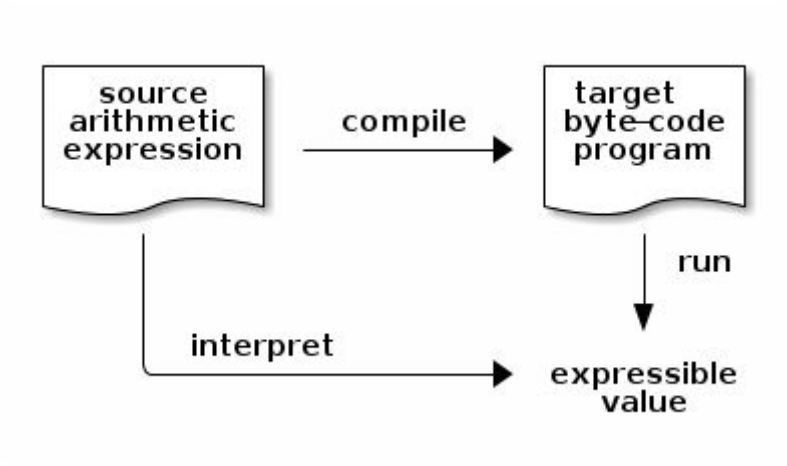
Overall in this chapter we experienced a chance to observe the nature of the function compile, which is a formal translator between source and target program As we can see the function uses its auxiliary recursive version inside the Target program constructor.

Chapter 5 Capstone

In this chapter we combine all functions discussed above in order to prove the main point of this project which is the capstone theorem that states that interpreting a source program yields the same result as running the output of compiling the same source program.

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

Let's begin our debate by discussing the meaning of the above property. The diagram below shows that if we take a source arithmetic expression interpret in the result will show the expressible value. If we recall our implementation of the function interpret from the first chapter we can clearly see that this is a case. If we compile the same source program we will get a target byte-code result which was shown in the previous chapter, and then if we run that result we will get an expressible value. Looking at the example, if our arithmetic expression is Literal 3, then interpreting it will yield a Expressible 3 as an answer. Then compiling it will give us PUSH 3 :: nil, and running that will leave us with Expressible 3.



The formalization of the above will be shown in our capstone. The proof will use a master lemma, which is formulated below:

```
Lemma capstone_aux :
 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))
```

In words, we are trying to show that evaluate ae which evaluates to expressible nat implies that fetch decode execute loop applied on complie_aux ae ds evaluates to OK (n :: ds). And evaluate ae which evaluates to expressible message implies that fetch decode application returns KO s. The lemma above is formulated based on the previous discussion about the diagram . We know that the result of the evaluation can either be a natural number or an error message. If it is a natural number that means that the ds has the correct amount of value, hence we can imply that the result of compiling and fetch decoding will be an OK message with n as a head of our stack. Otherwise, if we evaluate to the error message we know that the result of decoding and compiling with yield and KO string error.

We conduct the above lemma by induction on ae. Then, we solve a base case using fold unfold lemmas. Then, evaluation of an arithmetic expression can either be a natural number or a string so we destructure these two cases. We use the built in function called eq_refl to show that the function is equal to itself. The rest of the proof is mechanical and uses already introduced tactics such as injection, destruct and rewrite.

Equipped with the above we can have a final look at the capstone property. We destructure capstone_aux with regard to two cases H_OK for the data stack and H_KO for string. We also destructured evaluate ae to address the natural number and the string case and give it a name — H_ae. Then we can rewrite our hypothesis with the correct arguments which finishes the proof.

That summarizes the most important property of this project, the fact that interpret applied to a source program yields the same result as run applied on compile applied on a source program.

Chapter 6 Verify

In this chapter we will focus on function verify. This function merely symbolically executes a bytecode program to confirm (1) whether no underflow occurs and (2) upon completion of the program, there is exactly one natural number on top of the stack. The verify is a program that will check if the compiler worked properly based on the length of the stack.

This function takes two arguments: a list of byte-code instructions and a natural number which represents the size of the stack.

We begin our debate by writing a direct implementation of verify_aux.

```
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.
```

As we can observe above, this is a recursive function that matches bcis either with either empty list in a base case or byte-code instruction as a head of the list of byte-code instructions in the induction step. The natural number is an accumulator. This number represents the length of the list. It is originally indicated to 0 and then grows by 1 with every byte-code instruction being taken off the stack.

Now let's move to properties about the verify. We are trying to show that if we take two lists of byte-code instructions, then the verification will happen as follows. This proof refers to the property about decoding and executing two concatenated lists of byte-code instructions, which was shown earlier in the project. We show that if the first stack is verified and it has some height h1 then we can proceed to verifying the concatenation of first and second. Otherwise we yield and error after first verification as there is no point of checking the second one if the first one is not verified correctly. To wit:

```
Lemma byte_code_append_property_verify:
 forall (bcis1 bcis2 : list byte_code_instruction)
 (h : nat),
 (forall h1 : nat,
 verify_aux bcis1 h = Some h1 ->
 verify_aux (bcis1 ++ bcis2) h = verify_aux bcis2 h1)
/\
 (verify_aux bcis1 h = None ->
 verify_aux (bcis1 ++ bcis2) h = None).
```

We prove the above by conducting the induction on bcis and then dividing the proof into 5 cases: push, plus, minus, multiply and divide. When we arrive at the stage when we need to deal with the second list bcis2, we destruct the h (which in this proof stands for high) and continue wit subcases.

Mirroring the above reasoning for all the cases we summarize the proof.

Now, equipped with above we can prove the theorem about verify aux which will be needed in later reasoning. The lemma states that the function verify applied on the result of compilation will always yield some hight S h. That means that whatever is compiled passes the verification.

```
Lemma about_verify_aux:
 forall (ae : arithmetic_expression)
 (h : nat),
 verify_aux (compile_aux ae ) h = Some (S h)
```

This proof can be solved by induction over arithmetic expression. The proof is mechanical and uses familiar tactics to systematically step by step lead to reflexivity for all the cases. Now we can move to proving that the compiler emits well behaved code. This proof is trivial once we have access to the about verify aux lemma. The most difficult part is to formulate the lemma itself.

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

The above theorem shows us the verification of compiler based on its length. Its significance comes with the ease of verification. Verify checks the whole datastack and ensures the machine that we are not missing any items of byte-code instruction list. Without this function we will have to check the whole data stack one by one which is very time consuming and computationally expensive.

Chapter 7 Magritte

In the following chapter we will write the magritte versions of the operations which we discussed in the whole project. Before we jump into it let us focus for a moment on a little bit of context.

Rene Fransois Ghislain Magritte was a surrealist artist. He became well known for depicting ordinary objects in an unusual context, one of which is his painting “This is not a pipe”. His philosophy around that piece of art was as follows: it is not a pipe is just a representation of it .

Taking the surrealism into CPA we will now write a Magritte interpreter for the source language that does not operate on natural numbers but on syntactic representations of natural numbers. However as interpret uses the evaluate let’s first implement the evaluate magritte function. As the standard to evaluate this is a recursive function that matches the given arithmetic expression with one of the five cases. However, this time if our arithmetic expression is literal n we match it with literal n. If it is any other operation we match it with the name of the operation and recursive call on ae1 as a first argument with recursive call on ae2 as a second argument. Effectively we can observe that our evaluate magritte is a simple mirror function, as it will return exactly what is given.

Below the implementation the accoming fold unfold lemmas for all 5 cases are presented.

As mentioned above we can observe that the evaluate magritte is effectively a mirror function. Now let's formally prove that observation in the about evaluate magritte lemma. Formally we are showing that:

forall ae : arithmetic_expression,
evaluate_magritte ae = ae.

We conduct the above proof by induction on ae. The proof is mechanical and uses familiar tactics such as rewriting fold unfold lemmas and induction hypothesis.

The definition of the source interpret requires the expresible value as an output type. However, because in our magritte function we changed the evaluate function we require to implement the new debition of expressible value which in this project will be called an expressible valued magritte. And expressible value magritte is defined as :

Inductive expressible_value_magritte : Type :=
| Expressible_nat_magritte : arithmetic_expression -> expressible_value_magritte
| Expressible_msg_magritte : string -> expressible_value_magritte.

In words, expressible value magritte is a type which can be constructed in two ways. Either as an Expressible nat which takes arithmetic expression and returns expressible value magritte or as expressible message which takes string and returns expressible value magritte.

For the further reasoning we also need to change the type of data stack. Data stack magritte is defined as a list of arithmetic expressions.

Definition data_stack_magritte := list arithmetic_expression.
One additional type which requires changing is the result of decoding execution:
Inductive result_of_decoding_and_execution_magritte : Type :=
| OK_magritte : data_stack_magritte -> result_of_decoding_and_execution_magritte
| KO_magritte : string -> result_of_decoding_and_execution_magritte.

In words, the result_of_decoding_and_execution_magritte is a type that can be contracted wither as OK_magritte, which takes data stack or KO_magrite which takes string. Equipped with all of the adjusted data types we can proceed to writing the definition of interpret_magritte_s. This definition is exactly the same as normal interpret the only difference is in types. Here we return the Expressible nat magritte on the product of evaluate magritte .

A similar situation happens in the definition of decode execute. The magritte version uses just different types but the general layout of definition stays the same. In decode execute function the one import thing to point out is the fact that when we match the data stack with 3 items at the top of the list we have ae2 :: ae1 not ae1 :: ae2 . That is due to the fact that we deal with stack list here, which means that the last item which entered the kist is the first to get out. The last item which we added is ae2, hence it is at the head of our list.

In the remaining part of the chapter we also show the target interpreter of run magritte. This function have an exact same structure as the run described in the earlier part of the project. However, outside of run_magritte it has one another hidden name. As the function takes an target_program as an input and returns expressible value magritte which can be an arithmetic expression, it is effectively a decompiler. This function translates the target language into source program, so we can say that we have built a reverse compiler just by changing the types of our target interpreter.

Diving deeper into the word of magritte we can also show that the capstone of the project which shows that interpreting a source program yields the same result as running the output of compiling the same source program, also holds for the magritte implementation. As we perform the same operations just using different data types the structure of the proof is exactly the same.

The last proof to be shown in this project is a corally of capstone magritte which states that :

```
forall (ae : arithmetic_expression)
    (tp : target_program),
  compile (Source_program ae) = tp ->
  run_magritte tp = Expressible_nat_magritte ae.
```

This proof consists of three rewrite systems, so its structure is not crucial for us. Nevertheless it shows us what was already discussed in the run version of magritte. That effectively we implemented a decompiler (left side of the corollary).

Conclusion

Overall the above project gets us familiar with the language processors for arithmetic expressions. We had a chance to observe and experience how the virtual machine functions and analyses the code given.

We implemented source program interpreter and target program interpreter with all of its accompanying functions. All of the above let us show the capstone of the project which stated that the interpreter applied to a source program yields the same result as run does when applied on the compiler, applied on a source program. We also discovered the mystery of magritte function, and have implemented the decompiler using magritte target interpreter.

The above project also summarizes our knowledge from the whole semester. We were able to use the advanced tactics of proving such as introducing the induction, strengthening the hypothesis, using injection, destruction of our hypothesis and much more. After the whole course we can confidently work in the world of proofs and Coq Proof Assistant.