A formalization of symbolic execution in Coq

Tania-Adelina Bulz

Summer 2020

Abstract

This report aims to illustrate the connection between symbolic and concrete execution for multithreaded shared variable programs by providing a formalization of the two executions as transitional systems. We show how to relate these transition systems in order to express formal definitions of correctness and completeness. The language used for the formalization of execution is Coq, a formal proof management system used as an interactive theorem prover. Towards the end we use Coq's tactics which implement backward reasoning to look at a piece of the completeness proof. This serves as a basis for future formal proofs of correctness and completeness of symbolic execution .

1 Introduction

The two main topics discussed in this report are symbolic execution versus concrete execution in analyzing programming languages and the usage of Coq as a proof assistant in formalizing execution. Symbolic execution is a technique used for software analysis, applied mostly to sequential languages as the nondeterminism in concurrent languages gives rise to a state space explosion [5]. In the article on sympaths [5], the authors are proposing a formal model of symbolic and concrete execution for multithreaded shared variable programs. The formalization of the two executions in Coq closely follows this formal model in its attempt to create a proof basis for formal justification in terms of correctness and completeness [5].
Coq is used for the formalization as an interative theorem prover. The main interest of this project lies in the mathematical techniques to reason about properties held by programming languages that can be found in Coq [2] . When it comes to language verification, two of the most popular frameworks used nowadays are K and Coq. As opposed to conventional program verifiers which are created for specific programming languages, frameworks like K and Coq do not have language semantics in the tool itself. As a result, they can be applied to any language, including a generic one based on a programming principle as it is the case for this project. However, as there are not many user-friendly syntax notation, there is more work to be done before reaching the step of specifying and proving properties of the given language, like correctness and completeness. First and foremost the syntax of the language need to be formally defined followed by a formal definition of the language semantics. With both the syntax and the semantics, the behaviour of different programs can be analysed. Based on this properties of the language are expressed and proven correct. As [5], the article this project is based on, focuses on symbolic execution, a big part of this report is focused on defining the formal semantics of the language. The report will go into the details of each of these three steps (defining syntax, semantics and proving properties) and suggest ways to further develop the formalization. The complete Coq program can be found at

$$https://github.com/taniaadb/sympath-coq$$

2 Syntax

Coq's logic is based on the formalism of Calculus of Inductive Constructions. It represents functional programs using the style of the ML language and proofs in higher-order logic [4]. The syntax of the language used in the formalization is defined using inductive data-types. We use the language presented in [5] in order to illustrate symbolic and concrete execution in Coq. This is a basic programming language PL with a given set of types and program variables. For simplicity, we use the type Nat instead of Int for our arithmetic expressions and assume that all variables are global and can only hold natural numbers, as opposed to both numbers and booleans.
The syntax of arithmetic expressions uses inductive type declaration starting with the key-word Inductive. In the definition of aexpr we give a constructor that specifies the type of arguments it requires for each arithmetic expression from the language. We work with the following aexpr: natural numbers, multiplication and addition. Other expressions can easily be added by specifying their constructors in the inductive definition. The variables require some extra processing. As the values stored in the variables are needed when defining the semantics of PL, total maps from strings to natural numbers will be used, as defined in the Maps.v file from the github repository. This implementation is taken from the Software Foundation series (Logical Foundations [2]). Thus, a state represents the current values for all variables. 0 will be used as the default value for undeclared variables.

```
Inductive aexpr : Type :=
  | ANum (n : nat)
  | APlus (a1 a2 : aexpr)
  | AVar (s : string)
  | AMult (a1 a2 : aexpr).
```

Listing 1: Inductive type - arithmetic expressions

Boolean expressions are similarly defined, accepting and, not, equal and less than expressions.
In order to make it easier to write programs and define semantics, some notations and coercions are introduced. The notations are declared in notation scopes as to clarify which interpretation of the symbol they are referring to.

```
Notation "x + y" := (APlus x y) (at level 50, left associativity) : expr_scope.
Notation "x * y" := (AMult x y) (at level 40, left associativity) : expr_scope.
```

Listing 2: Notation - arithmetic expressions

Further, the arithmetic and boolean expressions are used to define statements. Accepted statements in PL are sequential compositions, assignments, conditionals and while-statements. A while-loop with an optional bound is

added in order to avoid an infinite number of paths. This is taken from the Maude implementaton in [5]. Coq, as opposed to Maude, accepts non-terminating executions as long as they are declared as inductive Propositions so the bounded while statement is not necessary, but illustrates how the set of accepted statements can be extended.
A threadPool is defined in order to introduce concurrency in PL. A threadPool can consist of a single thread represented by a thread id (natural number) or a pair of several threads. It is now possible to define a statement that is similar to Example 1 introduced in [5] to illustrate symbolic execution:

$$\textbf{proc}\{x := 0; x := y + 1; s\}\ \textbf{proc}\{\textbf{if}(x = 0)\{s1\}\textbf{else}\{s2\}\}$$

```
Definition example_article : threadPool :=
   (TPar
      << id 1 | X ::= 0 ;;
                X ::= Y + 1 >>
      << id 2 | If X = 0 THEN
                   Y ::= 0
                ELSE
                   Y ::= 1 >>).
```

Listing 3: Example 1 - sympath

In the Coq implementation the procedure is packed in a thread and the thread in a thread-id. As Coq needs concrete statements the s from procedure 1 is not mentioned and s_1 and s_2 are specified as two different assignments. In the rest of the implementation, the accent is put on statements and threadpools as the programs are not very advanced. However, if PL is to be used for more advanced calculations it is possible to create a program which consists of a list of procedures. The implementation of Lists from Coq's base library is used in the inductive definition of a program. A procedure is defined as a statement wrapped in a Proc constructor. As mentioned in [1], this encoding of PL programs and statements into terms corresponds to constructing parse trees of programs. This process is manual and there are no features in Coq for parsing programs like associativity in Maude. Therefore, the defined syntax carries no semantic information at this point.
One last element of the syntax is connected to the symbolic evaluation. As arithmetic and boolean expressions will not be evaluated to concrete values, a symbolic language to evaluate our expressions is needed. The symbolic expressions are defined closely related to the expressions in PL. LNat represents symbolic variables. These are defined as x, y or z followed by a natural number. A program variable X can now be represented as a logical variable x(0). Additionally, arithmetic expressions can be represented as symExprArit and boolean expressions as symExprBool. As the notation for the symbolic language is similar as the notation for PL, an additional scope is defined to differentiate between the two. An example of a symbolic arithmetic expression can be found bellow:

```
Check (1 + 2 + x(0))%symexpr.
```

Listing 4: Example symExprArit

3 Semantics

3.1 Concrete Evaluation

Now that the syntax of PL is in place, the second step in order to be able to discuss about properties is to define the semantics of the language. PL is not deterministic because of the mutithreaded setting. As programs in PL can evaluate to more than one value, functions are not sufficient to denote the construct of the language. Relations, as opposed to functions in Coq. are more general [1] and offer the possibility of looking into the internal steps of a computation. Another downside of functions in Coq is that they have to terminate in order to be accepted. As the WHILE statement are not always terminating, the functional approach runs into difficulties. This can be solved by adding a parameter telling the evaluation function how long to run, similar to the bounded while-loop from the Maude implementation associated with [5]. This together with concurrency hint that a relational approach is to be preferred.
In order to capture the semantics of arithmetic expressions, a binary transition relation between a pair of arithmetic expressions and a total map is created. The total map models the association between strings (program variables) and constants (values of the variables). Thus, this represents the valuation function from [5] and the binary transition relation represents the transition system for the concrete execution of PL programs. The concrete states from the same article are similar to the pairing between expressions and the total map, but require an extra level given by an additional relation between a pair consisting of a thread-pool and the same total map used for evaluating statements.
The transition relation, is an inductively defined proposition (Prop) in Coq. The sort Prop represents expressions that denote logical proposition. What is done in the cstep is to show that a pair of two concrete states on the statement level belongs to the cstep relation. The valuation function from [5] is represented by the total-map "state" and the "statement" is previously defined in the syntax file. Notation is added in order to make the relation more readable. The following excerpt is taken from the full cstep definition in the github repository.

```
Inductive cstep : (state * statement) -> (state * statement) -> Prop :=
    | CS_Ass : forall st i a,
        (|| st, i ::= a ||) -->c (|| (i !-> (aeval st a) ; st), SKIP ||)
    | CS_SeqStep : forall st s1 s1' s2 st',
        (|| st, s1 ||) -->c (|| st', s1' ||) ->
        (|| st, s1 ;; s2 ||)-->c (|| st', s1' ;; s2 ||)
```

```
    | CS_SeqFinish : forall st s2,
        (|| st, SKIP ;; s2 ||) -->c (|| st, s2 ||)
    | CS_IfTrue : forall st s1 s2 b,
        (beval st b) = true ->
        (|| st, If b THEN s1 ELSE s2 ||) -->c (|| st, s1 ||)
    | CS_IfFalse : forall st s1 s2 b,
        (beval st b) = false ->
        (|| st, If b THEN s1 ELSE s2 ||) -->c (|| st, s2 ||)
where "'(||' st ',' t '||)' '-->c' '(||' st' ',' t' '||)'" := (cstep (st,t) (st',t')).
```

Listing 5: Small-step evaluation for statements

For example, CS_Ass states that a concrete state on the statement level ($||st, i ::= a||$), where st is the valuation and the statement is an assignmnent(SAss) that assigns the program variable i the arithmetic expression a, is related via the cstep relation to another concrete state on the statement level ($||(i\ !\rightarrow$ (aeval st a); $st), SKIP||$), where the previous valuation is extended with a mapping from i to the evaluation of the arithmetic expression and the statement $SKIP$. The $SKIP$ statement is used to signal the end of the evaluation for a given statement. An alternative would be to add an extra layer and reduce $SKIP$ to *idle* as suggested in [5]. Another example from cstep is CS_IfFalse. This rule states that for every valuation st, statement s1 and s2 and boolean expression b, if b is evaluated to false, the the concrete state (statement level) ($||st$,If b THEN s1 ELSE s2$||$) is related via cstep with the concrete state (statement level) ($||st, s2||$). In other words, the evaluator steps in the "ELSE" branch of the conditional.
In addition to cstep which gives semantics to statements, two more relations are needed that define how the arithmetic and boolean expressions are evaluated. These relations are, as seen from cstep, aeval and beval. They are both recursive definitions, illustrated by the keyword Fixpoint in Coq. These relations take in a valuation and an arithmetic or boolean expression and recursively built its value following its syntax definition. In case of a program variable, its concrete value is obtained from the valuation st.
The evaluator is formulated in a small-step style. This style offers a finer-grained way of reasoning about program behaviours [3], as opposed to big-step style when an expression is evaluated to its final value. The small-step style is particularly useful for concurrency as it gives the possibility of switching between the threads in the thread-pool. In order to give full semantics to PL, the evaluation relation for the threadpool, tpstep, is introduced. This reduces the threads until statements level and uses cstep on the statements. The non-deterministic aspect of the language is captured by the two rules bellow from the tpstep transitional relation, that show that at each step, the evaluator can either focus on the first thread or the second thread from a thread pair.

```
Inductive tpstep : (state * threadPool) -> (state * threadPool) -> Prop :=
    | TS_T1 : forall st t1 t1' t2 st',
        (| st , t1 |) -->tc (| st' , t1' |) ->
        (| st, TPar t1 t2 |) -->tc (| st', TPar t1' t2 |)
    | TS_T2 : forall st t1 t2 t2' st',
        (| st, t2 |) -->tc (| st', t2' |) ->
        (| st, TPar t1  t2 |) -->tc (| st', TPar t1 t2' |)
where "'(|' st ',' t '|)' '-->tc' '(|' st' ',' t' '|)'" := (tpstep (st, t) (st', t')).
```

Listing 6: Small-step evaluation for threads

Note that both cstep and tpstep are single-step reduction relations, which formalize the individual steps of a transition machine for executing programs [3]. As the aim is to reduce the programs to completion ($SKIP$), the multi-step reduction relation *multi* is introduced. Multi is defined generically as it is used several times throughout the project. Given a binary relation R, the *multi-step closure of R* is defined as the reflexive and transitive closure of R. Thus, in the multi-step closure of cstep, multi cstep, with notation $' \rightarrow c^{*\prime}$, if $t \rightarrow c^{*}t_1$ and $t_1 \rightarrow c^{*}t_2$, then $t \rightarrow c^{*}t_2$, where t, t_1, t_2 are concrete states.
Now that all the semantic elements are in place, it is possible to look at how example 1 from [5] is evaluated in this framework. We record the expected result in as a Coq example and prove that our assertion is correct.

```
Example tpstep_article :
  exists st st',
    (| st, example_article |) -->tc* (| st', << id 1 | SKIP >>|).
```

Listing 7: Evaluation for example from article

This states that there are 2 valuations, st and st', such that multi tpstep relates the concrete state formed by the valuation st and the thread-pool defined in example_article and the concrete state formed by the valuation st' and the completion element for threads, $SKIP$ packed in the thread with the lowest thread identifier. In order to prove that the assertion is correct, the order of application of each rule from multi tpstep, tpstep, multi cstep and cstep in order to achieve completion is given. Arbitrary valuations are introduced for st and st' with the help of ex_intro. eapply on ex_intro and multi_step are used as opposed to the apply tactic in order to take advantage of Coq's automation features. Apply requires explicitly intermediate valuations while eapply waits until the end of the proof to match the intermediate states backwards. This proof can be automated even more by giving the constructors of each of the relations as hints to Coq. With these hints, eauto tactic manages to do much of the proof without pointers towards the order of rule application.

```
Proof.
  eapply ex_intro. eapply ex_intro.
  unfold example_article.
  eapply multi_step. apply TS_ST1. apply CS_SeqStep. apply CS_Ass. simpl.
  eapply multi_step. apply TS_ST1. apply CS_SeqFinish.
  ...
  Qed.
```

Listing 8: Proof of example

3.2 Symbolic Evaluation

As the transition system for concrete evaluation is in place, it can be used to introduce symbolic execution of PL. The semantics of symbolic execution is built up parallelly to the semantics of concrete execution, following the same relations between states. The valuation is replaced by symbolic substitution, a function from program variables to symbolic arithmetic expressions (symExprArit). This is formalized in the Coq implementation by the total map sym_state. As this is a total map, the empty substitution from [5] is replaced by a mapping that sends all program variables to SymNat(0), the symbolic number 0. The recursive function sym_aeval defines how an arithmetic expression is evaluated to a symbolic arithmetic expressions. As a program variable is an arithmetic expression, sym_aeval takes a symbolic substitution as an argument and retrieves the symbolic value of the variable from it. The rest of the function follows the inductive definition of symExprArit. sym_beval is defined in a similar way, keeping a parallel to the bexpr function.
In order to describe symbolic execution in [5], the notion of symbolic path is introduced. A symbolic path is a sequence of events $\iota\langle e, V_1, V_2\rangle$, where ι is a thread identifier, e a predicate and $V_1, V_2 \subseteq VarSet$. The predicate e captures path conditions and the sets V_1 and V_2 register the read and write effects of a statement [5]. In the Coq implementation, the variables are not gathered in a set, but in a list. The base implementation of lists from the Coq library is used for that purpose. However, as the formalization does not cover partial order reduction, the variable lists are not put into use in the same way as in the Maude implementation associated with the article [5]. In Coq, the recursive functions vars_arit and vars_bool gather the variables from arithmetic and boolean expressions respectively.
Symbolic paths in Coq are defined as lists of events, with the base implementation of lists from the Coq library. A symbolic path is then a list of events (as opposed to sequence). The empty path is represented by the empty list. With the implementation of the symbolic path in place, it is time to define the transition system for symbolic execution. This is done through a transition relation between symbolic states, tp_sym_step which goes into the threadpool and an additional transition relation on the statement level, sym_step. These are similar to tp_step and cstep in the concrete semantics, the main difference is that they relate symbolic states as opposed to concrete states.

```
Inductive sym_step : tid -> (sym_state  * symPath * statement) ->
                           (sym_state * symPath * statement) -> Prop :=
    | S_Ass : forall st i a l sp,
        (|| st, sp, i ::= a ||) --[l]-->s
    (|| (i |-> sym_aeval st a ; st), sp ++ [l<SymBool true, [i], vars_arit a>], SKIP ||)

    | S_SeqStep : forall st s1 s1' s2 l sp sp' st',
        (|| st, sp, s1 ||) --[l]-->s (|| st', sp', s1' ||) ->
        (|| st, sp, s1 ;; s2 ||) --[l]-->s
        (|| st', sp', s1' ;; s2 ||)

    | S_SeqFinish : forall st s2 l sp,
        (|| st, sp, SKIP ;; s2 ||) --[l]-->s
        (|| st, sp, s2 ||)

    | S_IfTrue : forall st b s1 s2 l sp,
        (|| st, sp, If b THEN s1 ELSE s2 ||) --[l]-->s
        (|| st, sp ++ [l<sym_beval st b, [], vars_bool b>], s1 ||)

    | S_IfFalse : forall st b s1 s2 l sp,
        (|| st, sp, If b THEN s1 ELSE s2 ||) --[l]-->s
        (|| st, sp ++  [l<SymNot (sym_beval st b), [], vars_bool b>], s2 ||)
where " '(||' st ',' sp ',' t '||)' '--[' l ']-->s' '(||' st' ',' sp' ',' t' '||)'" :=
        (sym_step (l) (st,sp,t) (st',sp',t')) .
```

Listing 9: Small-step symbolic evaluation for statements

The "sym_state" given as an argument represents the symbolic substitution so the transition system describes how two different symbolic states are related to each other. The relation is written is small-step style so $' - [l] \rightarrow s'$ describes an atomic execution step associated with a specific thread l. As the threadpool is being handled in the tp_sym_state relation, currying is used to handle the arguments of the sym_step relation. The thread id is taken from tp_sym_state and by applying tid to the relation, a binary relation between 2 symbolic states within a thread is obtained.
The S_Ass rule describes the symbolic execution of an assignment. S_Ass states that a symbolic state on the statement level $(||st, sp, i ::= a||)$, where st is a symbolic substitution, sp is a symbolic path and the statement is an assignment that assigns the program variable i the arithmetic expression a, is related via the sym_step relation to another symbolic state on the statement level $(||(i \ ! \rightarrow (\text{sym\_aeval st a}); st), sp \ ++ \ [l \ <\text{SymBool true}, [i],\text{vars\_arit } a >], SKIP||)$. The previous symbolic substitution is extended with a mapping from i to the symbolic evaluation of the arithmetic expression defined earlier. The previous sympath is also updated according to the rules described in [5]. sp is extended with another event with the corresponding thread identifier, indicating that an assignment has the symbolic variant of True as path condition,

there is a write to the program variable i ($[i]$) and there are reads from the list of variables in the arithmetic expression a (vars_arit a).$SKIP$ is used as a termination marker for symbolic evaluation as well.
The symbolic evaluation of conditionals is slightly different from the concrete evaluation as it is not known whether a certain boolean expression b is true or false. Therefore, the evaluation is branching for every conditional, creating two symbolic states which correspond to the two different outcomes of the evaluation of b. This is represented in the Coq implementation by adding non-determinism to the sym_step relation. Every symbolic state on statement level consisting of a symbolic substitution st, sympath sp and a conditional statement, ($||st, sp$, If b THEN s1 ELSE s2 $||$) , is related to both ($||st, sp$ $++$ $[l$ <sym_beval st b, [], vars_bool b>], $s1||$) and ($||st, sp$ $++$ $[l$ <SymNot (sym_beval st b), [], vars_bool b>], $s2||$). The two different outcomes of the evaluation of b are captured by the path condition in the event that extends the symbolic path, sym_beval st b and SymNot (sym_beval st b). The while and bounded while statements are being treated in a similar way to the conditionals.
The SCHEDULE that introduces branching and CONTEXT rule that lifts transitions from [5] are being covered in the tp_sym_step. The following excerpt is taken from the full tp_sym_step definition in the github repository. These 2 rules show how the relation accesses the earlier defined sym_step if the evaluation has come far enough in the threads to reach a statement.

```
Inductive tp_sym_step : (sym_state * symPath * threadPool ) ->
                        (sym_state * symPath * threadPool ) -> Prop :=
    | S_ST1 : forall st s1 s1' st' l t2 sp sp',
        (|| st, sp, s1 ||) --[l]-->s (|| st', sp', s1' ||) ->
        (| st, sp, (TPar << l | s1 >> t2) |)  -->ts
        (| st', sp', (TPar << l | s1' >> t2) |)

    | S_ST2 : forall st s2 s2' st' t1 l sp sp',
        (|| st, sp, s2 ||) --[l]-->s (|| st', sp', s2' ||) ->
        (| st, sp, (TPar t1 << l | s2 >>) |)  -->ts
        (| st', sp', (TPar t1 << l | s2' >>) |)
where "'(|' st ',' sp ',' t '|)' '-->ts' '(|' st' ',' sp' ',' t' '|)'" :=
        (tp_sym_step (st, sp, t) (st', sp', t')).
```

Listing 10: Small-step symbolic evaluation for threads

It is now possible to observe how an example is evaluated with the help of the multi-step closure of sym_step and tp_sym_step. The example is the one previously discussed in this report and [5], with an expected result of the evaluation. The assertion is proven true with the same strategy as for concrete evaluation, by giving the order in which each of the rules in our relations are being applied.

```
Example tpsym_article_true_brances :
  (| (X |-> x(0); Y |-> y(0)), [], example_article |) -->ts*
  (| (X |-> 0 + 1 ; Y |-> 0 ; X |-> 0 ; X |-> x(0) ; Y |-> y(0)),
               [id 1<SymBool true, [X], []>;
                id 2<(0 = 0), [], [X]>;
                id 2<SymBool true, [Y], []>;
                id 1<SymBool true, [X], [Y]>], << id 1 | SKIP >>|).
```

Listing 11: Small-step symbolic evaluation for threads

4 Language Properties

After formalizing the syntactic structures and semantics, it is time to specify and reason about the PL language properties properties. The properties in focus in [5] are correctness and completeness of symbolic execution. The property that is expressed in the Coq formalization is completeness. [5] defines the notion of completeness as follows:

Theorem 3 (Completeness). *For any concrete computation $cs_0, ..., cs_n$, there exists a symbolic execution $ss_0, ..., ss_n$ such that for $0 \leq i \leq n$ the states ss_i and cs_i have the same thread pool, and $\epsilon_i = \epsilon_0 \circ \sigma_i$, where ϵ_i denotes the valuation of cs_i and σ_i denotes the substitution recorded by ss_i.*

In order to be able to specify this property in Coq, it is necessary to have a look at function composition. A new total map is described that illustrates the connection between a valuation and a symbolic substitution. subst_state is a total map from symbolic variables to arithmetic expressions. In order to obtain the expression a symbolic variable is mapped to, a recursive function, subst_symExprArit which follows the definition of symExprArit is used. The relation concretization is following the connection between a valuation and a symbolic substitution with the help of a subst_state throughout the execution of a threadpool. The Con_empty rule states that an empty valuation is always related to an empty symbolic substitution. The Con_Update rule describes how a related pair of a valuation st and a symbolic substitution sym_st can be updated to an extended related pair $(X! \rightarrow n; st)$ and $(X| \rightarrow sym\_X; sym\_st)$. An example that shows how the concretization works is found bellow. The

```
Example conc_2:
  concretization (x 0 ||-> 1) (X |-> x 0 + 1) (X !-> 2).
Proof. apply Con_update. apply Con_empty. simpl. reflexivity. Qed.
```

Listing 12: Example concretization

valuation $(X! \rightarrow 3)$ and the symbolic substitution $(X| \rightarrow x0 + 1)$ are related via the composition $(x0|| \rightarrow 1)$ because if the value of $x0$ from the subst_state $(x0|| \rightarrow 1)$ is replaced in the symbolic substitution, the program variable X is

mapped to the same value as in the valuation $(X! \to 3)$.
Now that concretization is defined it is possible to express the completeness property for the first relation defined within concrete execution (cstep):

```
Theorem complete:
    forall st st' stm l f st_s sp,
      (|| st , stm ||) -->c (|| st' , SKIP||)
      -> concretization f st_s st ->
    exists st_s' sp',
      (|| st_s, sp, stm ||) --[l]-->s (|| st_s', sp', SKIP ||)
      /\ concretization f st_s' st'.
```

Listing 13: Completeness theorem

A complete proof together with assumptions of this property can be found in the full Coq implementation in the github repository. The proof starts with the intro tactic to introduce the variables from the $forall$ context and the premises of each implication. Then comes the inversion tactic on one of the premises: $(||st, stm||) \longrightarrow (||st', SKIP||)$. Inversion deduces what equalities are necessary in order for the premise to be true. In other words, it keeps track of what needs to happen in order for a concrete state to be executed to completion. As PL accepts different types of statements, a subgoal for each type of statement that can achieve completion is now created.
The most complicated situation is when the statement is an assignment as it is here where the valuation and symbolic execution are expanded. The main premises for this case are
$(||st, i ::= a||) \longrightarrow c(||i! \to aevalsta; st, SKIP||)$ and
$concretization\ f\ st\_s\ st$ and the goal is
$exists\ (st\_s' : sym\_state)(sp' : symPath)$,
$(||st_s, sp, i ::= a||) - [l] \to s(||st_s', sp', SKIP||)\ \wedge\ concretization\ f\ st\_s'\ (i! \to aeval\ st\ a; st)$
st_s' as the expanded version of st_s, $(i| \to sym\_aevalst\_sa; st\_s)$ and sp' as the expanded symbolic path are introduced. The goal can now be split in the two subgoals that make up the and statement. The first goal is easy to prove as it is one of the rules in the constructor of sym_step. The second goal involving the concretization is a bit more complicated and requires an extra assumption:
$concretization\ f\ st\_s\ st \to$
$aeval\ st(subst\_symExprArit\ f\ (sym\_aeval\ st\_s\ a)) = aeval\ st\ a.$
If that is the case, the Con_update rule from concretization can be used to prove the subgoal and this covers the Assignment case.
The other subgoals are similar to each other. As there are no more assignments, the st_s' is introduced as st_s which is already part of the premises and in some cases an updated symbolic path is added. Then the subgoals are easily proved by using the rules in the sym_step definition. For the function compositions, the valuations and symbolic substitutions do not change so the goals are part of the assumptions.

This completeness proof for cstep can be used as a base for a completeness proof for the transition system designed for the threadpool (tp_step) and further for the multi-step closures of both relations.

5 Conclusion

The previous section has shown the first steps of formalizing the completeness proof for symbolic evaluation, as presented in [5]. The next step is to move to the correctness proof. Correctness states that *For every symbolic execution* $ss_0, ..., ss_n$ *such that* $\epsilon([\phi_n]) = true$*, where* ϵ_0 *is a valuation and* ϕ_n *the symbolic path of* ss_n*, there exists a concrete computation* $cs_0, ..., cs_n$*, such that, for* $0 \leq i \leq n$*, the states* ss_i *and* cs_i*, have the same thread pool, and* $\epsilon_i = \epsilon_0 \circ \sigma_i$*, where* ϵ_i *denotes the valuation of* cs_i *and* σ_i *denotes the substitution recorded by* ss_i [5]. An attempt to define this property as a theorem in Coq can look as follows:

```
Theorem correct:
    forall st_s st_s' stm sp l st f,
      (||st_s, [], stm||) --[l]-->s (|| st_s', sp, SKIP ||)
      /\ concretization f st_s st
      /\ (*the conditions in the symPath are true*)
      forallb (andb true)
         (beval_cond st (subst_sym_cond f (sym_cond sp))) = true ->
    exists st',
      (|| st, stm ||) -->c (|| st', SKIP ||)
      /\ concretization f st_s' st'.
```

Listing 14: Correctness theorem

The assumption that $\epsilon([\phi_n]) = true$ is illustrated in Coq by $forallb\ (andb\ true)\ (beval\_cond\ st\ (subst\_sym\_cond\ f\ (sym\_cond\ sp))) = true$ and the connection between the valuation and symbolic substitution $\epsilon_i = \epsilon_0 \circ \sigma_i$ is illustrated by the concretization relation. The previous proofs on completeness can be used to prove this theorem as well. Further on, as some elements connected to partial order reduction can already be found in the transition system, the correctness and completeness of POR can be defined and proven.
This report presents a study on a formalization of concurrent languages in the interactive theorem prover Coq. It first offers a possible syntax for the generic language PL, then it goes into representing the semantics as transition systems. The connection between concrete and symbolic evaluation is illustrated by inductive relations constructed in parallel. In the last section, a start on a completeness proof with respect to concrete semantics is attempted. Keeping that in mind, other possible extensions of this work are presented. Furthermore, as the language being formalized is a generic one, this work can be used as a basis for proving other properties regarding concurrent languages.

References

[1] Musab A. and Moore B. *K vs. Coq as Language Verification Frameworks.* URL: https://runtimeverification.com/blog/k-vs-coq-as-language-verification-frameworks-part-1-of-3/.

[2] Pierce B.C. et al. *Logical Foundations.* URL: https://softwarefoundations.cis.upenn.edu/current/lf-current/index.html.

[3] Pierce B.C. et al. *Programming Language Foundations.* URL: https://softwarefoundations.cis.upenn.edu/current/plf-current/index.html.

[4] Paulin-Mohring C., Woltzenlogel Paleo B., and Delahaye D. "Introduction to the Calculus of Inductive Constructions." In: *All about Proofs, Proofs for All, 55,* College Publications, Studies in Logic (Mathematical logic and foundations) (2015).

[5] de Boer F.S. et al. *SymPaths: Symbolic Execution Meets Partial Order Reduction.* 2020.