Project: Program Analyses Formalized in Coq

2018/04/05

In this project, you will formalize a small language, its semantics, and some basic static analyses.

At several points, we offer several options, that may make the project longer to complete but would also make the results more interesting. You are expected to describe and explain the choices that you have made in a short document, or in very clear comments. Not all parts are mandatory, but we consider that doing correctly the definitions of the two semantics, and of at least one analysis (complete with proofs) would be a good first objective.

We recommend to draft the definitions on paper or with OCaml before starting to work on the Coq development. This document does not give the full list of the intermediate results to prove, and organizing proofs so as to share results is an important task, to progress better in the project.

Language. We study a basic imperative language defined by the grammar below:

$\mathtt{v}$	$\in$	$\{\mathtt{v}_0, \mathtt{v}_1, \ldots\}$	variables, indexed by natural integers
$\oplus$	$\in$	$\{+, -, *, \leq, <, \mathbf{xor}, \mathbf{or}, \mathbf{and}\}$	binary operators
$\mathtt{e}$	$::=$	b	boolean constant
	$\mid$	z	mathematical integer constant
	$\mid$	$\mathtt{v}$	variable
	$\mid$	$\mathtt{e} \oplus \mathtt{e}$	binary opterator application
$\mathtt{s}$	$::=$	$\boxdot$	empty program
	$\mid$	$\mathtt{v} := \mathtt{e}; \mathtt{s}$	assignment
	$\mid$	$\mathtt{v} := \mathbf{irand}(); \mathtt{s}$	random integer value
	$\mid$	$\mathbf{if}(\mathtt{v})\{\mathtt{s}\}\mathbf{else}\{\mathtt{s}\}; \mathtt{s}$	condition statement
	$\mid$	$\mathbf{while}(\mathtt{v})\{\mathtt{s}\}; \mathtt{s}$	loop statement

We remark that the syntax of each statement (except the nil statement $\boxdot$) contains a statement to execute next. A variable may be uninitialized, or store an integer or boolean value. Any attempt to read a variable that is not initialized will result in an initialization error. Also, trying to apply an operator to arguments which are not of the right type, or to use as a condition a variable that stores an integer value will result in a typing error.

1. Abstract syntax. Complete the definition of the types to represent expressions and statements (you may start from the definitions that are given in the template). Define the types for variables (a variable is defined by a natural integer) and values (we recommend to use an inductive data-type with two constructors).

Semantics. A memory state should define a partial function from variables to values, where values are integers (in $\mathbb{Z}$) or booleans.

2. Variables, values and memory states. Define the type for memory states. We suggest two possible definitions:

- as lists of optional values, where the i-th element describes the i-th variable (`None` if the variable is undefined);
- as association lists where each element is a pair i, v, where v is the value of variable i, and which are sorted based on the indexes of the variables;
- a lazy choice is functions from variables to optional values, but values of that type are not always computable, so it may not seem a great choice.

An execution state is characterized by a program (the code that remains to be executed) and a memory state. Define this type.

Hint 1: As we will use partial functions from variables to other things than values, it makes sense to define a polymorphic type for these association tables instead.

Hint 2: Whatever type you use, you will need functions to search for the value of a variable, modify the value of a variable, remove a variable, etc, and a few correctness theorems.

Hint 3: Using Coq functions is tempting, doable for the semantics, and easier at this stage, but it will not be great for code reuse later in the project (as one of the association types will be needed to actually define an analysis).

3. Evaluation of expression. Define the semantics of expressions using a function (`Fixpoint`). Note: the hard (and interesting part) is to describe errors.

- Option 1: Return an optional value (`None`) means the expression crashed either due to a type or initialization error (this may make it impossible to prove separately an analysis that does only focus on initialization);
- Option 2: Distinguish typing and initialization errors, using a custom inductive type.

4. Small step semantics and trace. Define the small step semantics of the language, using an inductive predicate where each constructor defines a possible kind of transitions. The transition to an error state should appear clearly (Hint: to share code, it makes sense to have a polymorphic type defining "an error or a correct behavior", where a "correct behavior" may be a value or an output state). This inductive predicate should be of type `state` $\rightarrow$ `state_or_error` $\rightarrow$ `Prop`.

Based on the small step semantics, defined the semantics of finite traces.

5. Relational semantics. Define a relational semantics for the language that also has type `state` $\rightarrow$ `memory_state_or_error` $\rightarrow$ `Prop`, that describes the complete execution of a program.

Prove that this semantics is equivalent to the previous semantics in the following sense: there is an execution from state s to memory m if and only if there is an execution trace from s to $(\boxdot, m)$. In the same time you can also prove that both semantics describe the same error behaviors.

What does the first semantics express that the second one does not talk about ?

Initialization. This part is only relevant if you have made the choice to distinguish initialization and typing errors in Question 3. Otherwise, you may skip it and move directly to typing.

We have seen that program may crash when they try to read a variable that is undefined, and propose to define a simple analysis to detect such errors statically.

6. Initialization analysis. Propose a simple conservative static analysis that computes variable initialization information, and either returns a value that speicifies which variables are initialized at the end of the execution of a program, or bails out when it cannot prove that the program will encounter no initiailzation error. The analysis should take the form of a `Fixpoint` definition. Prove the analysis sound.

Hint 1: The analysis is expected to be conservative, but of course the analysis that always bails out will not be accepted `:-)`.

Hint 2: The analysis results could take the form of a partial function from variables to booleans, and where `true` means that the corresponding variable is initialized for sure.

Hint 3: For the soundness proof, you may use either a small step semantics, or the relational one; both choices are valid, but you should plan ahead which semantics you prefer to use.

Typing. We now focus on typing errors.

7. Typing analysis. Propose a conservative analysis based on types. To define it, describe an inductive type representing value types (i.e., booleans and integers), and a function (involving on a `Fixpoint` definition) that inputs a program, and determines whether the program can be typed or not. There are several options:

- you may do typing and initialization analysis in the same time or not (actually, if the semantics does not dinguish typing and initialization errors you can probably not prove the two analyses separately);
- you may do a flow sensitive analysis, which computes a type for each variable, and at each program point (this analysis will accept a program that stores a boolean value in a variable, and then reuses it to store an integer value *provided it does it consistently*);
- you may opt for a flow insensitive analysis, which does not accept a program where a variable may be used sometimes as an integer or as a boolean.

Hint 1: You will need to use typing environments, which are functions from variables to types, and the proof of correctness of the analysis will involve a relation that says that a memory is consistent with a typing environment (it looks like an abstraction relation, and should state that variables for which we have a typing information should be initialized and store a value of the proper type). Formalizing this relation early will help in getting the analysis right.

Hint 2: Thinking on the proof early in the design of the analysis should minimize the chances of having to redo it all because the proof breaks late in the process.

8. Proving the typing analysis. Formalize the soundness of the typing analysis. Prove that the typing analysis is sound in that sense.

Hint 1: The proof is likely split into two parts:

1. prove that a well typed program with respect to a typing environment and starting in a state that is consistent with the typing environment can do at least one step of execution (unless it is finished already —the statement $\boxdot$);
2. prove that if a program is well typed with respect to a typing environment and starts in a state that is consistent with that typing environment, then after this program makes one step, the consistency with respect to typing environments is preserved.

Hint 2: Carrying out the proof in the case of expressions first should help in envisioning the rest of the proof work.

Constants analysis. We now consider a constants analysis similar to the one in the course.

9. Constants analysis. Design and prove correct a constants analysis:

1. Define an inductive data type describing the elements of lattice of constants over values that are either boolean or integer (very easy);
2. Define abstract states as partial functions from variables to this lattice, and formalize the abstraction relation;
3. Construct the abstract interpreter as a function (using a `Fixpoint` definition;
4. Prove soundness, with respect to the relational semantics.

Hint: One of the main difficulties is the treatment of loops, due to the termination issue. A solution is to iterate over loops a number of times that is fixed by the height of the lattice (which is a function of the number of variables in a program). A better solution is to iterate until either that fixed integer is reached or the result is stable. In all cases, some work is required to write a function that Coq accepts, and to prove that this function over-approximates all the output states.