Formal Methods for Security and Privacy (SS20) Prof. Matteo Maffei

Project 2 – F*

Due | *24.07.2020* TU Wien

Setting up Your System for the Project

In order to complete this project, you will have to install the tool F*. Please download F* (version 0.96) from

`https://github.com/FStarLang/FStar/releases`

and install it on your machine.

Although you can use any editior to write F* code, we recommend to use `emacs` with the fstar mode. Please follow the installation guide on

`https://github.com/FStarLang/fstar-mode.el#setup`

General Remarks

- Before you start with the project read the submission instructions carefully. Since there will often be hints at the end of the exercises, please read the *whole* exercise before starting it.
- Be neat and hand in readable (possibly annotated) F* files since we will have to manually check the faithfulness of your lemmas and definitions. A comment in F* can be written as `(* this is a comment *)`
- Please use the file names specified in the exercise descriptions and the submission instructions.
- The exercises of this project are divided into *tasks* and *questions*:
  - Tasks refer to the practical part of the project and shall be inserted in the F* file `Project2.fst`. This file already contains definitions and lemmas that guide you through the exercises of the project.
  - Questions refer to the theoretical part of the project and should should be answered in the file `project2.pdf`. You will find questions in most of the following exercises. Please make sure to answer all these questions and insert explanations where required in the `project2.pdf` file. We emphasize that it is inherently necessary to answer the questions and fill explanations in order to receive full points.

In this project, we formalize the small-step semantics of the simplified EVM bytecode fragment that was presented in the lecture in F*. You can find the full (formal) paper definition in the file `semantics.pdf`.

Exercise 2.1: Small-step semantics(4 points)

The file Project2.fst contains definitions of the datatypes required for specifying the small-step semantics for (simplified) EVM bytecode. In addition, it contains an (incomplete) definition of the function step that models the (functional) small-step relation. These definitions closely follow the formal description given in semantics.pdf. Please read semantics.pdf carefully and match it to the formalization provided in Project2.fst. We tried to make the connection between the two formalizations clear by inserting explanatory comments in Project2.fst. If something stays unclear, please use the discussion board in TUWEL for posting your questions.

Normally, we would only consider contract executions that start on a call stack with exactly one element representing the execution of the initial (external) blockchain transaction. However, characterizing valid call stacks as those that result from the execution of such initial call stacks would make proofs quite tedious. Instead, we state some syntactic invariants that a call stack should satisfy in order to represent a valid smart contract execution. To this end, we define a function wellformed : callstack → Tot bool that decides whether a call stack statisfies these syntactic conditions. More precisely, we require that well-formed call stacks are non-empty and that every element of a call stack, but the top element represents a *call state*. A call state is an execution state that (could have) performed a call, as it has a sufficient number of arguments and the opcode that was executed in this state is a CALL instruction. Well-formedness allows us to assume that we always have at least one element on a call stack and that when returning from an (internal) transaction, we can assume the sufficient amount of elements on the local stack that can be accessed.

Consequently we define the step function only on those call stacks that satisfy well-formedness which releases us from defining it e.g. on the empty call stack. The step function returns a value of type step_outcome. These values carry despite the resulting call stack also the indication of whether the execution of a further step was possible.

We shortly discuss the general structure of the step function:

```
val step: tenv →cs: callstack {wellformed cs} →Tot step_outcome
let step te cs =
  match cs with
  | Ter ts [] →Stop (Ter ts [])
  | Ter ts (s :: ps) →Next (Exec ((apply_returneffects ts s)::ps))
  | Exec (s :: ps) →
   let (((gas, pc, mem, stack), (actor, input, code), gs)) = s in
     if gas < 1 then Next (Ter ExcState ps)
     else
       match (getOpcode code pc, stack) with
       | (ADD, a::b::stack') →
       ...
       | _ →Next (Ter ExcState ps)
```

The step function receives as input the transaction environment te and a call stack cs. It proceeds by pattern matching on the structure of the call stack:

- If the call stack consists of a single terminal state (so a halting or an exception state) on an otherwise empty call stack, no further steps can be performed as there was no caller where the internal transaction that is currently executed can return to. This is indicated by the step outcome Stop

- If the top element of the call stack is a terminal state and there is at least one execution state on the remaining call stack (so there exists a caller state), then the execution goes on with the execution the caller state after applying some effects of the the terminated internal transaction (such as writing the return value in the memory of the caller state) on the caller state. This functionality is outsourced to the function apply_returneffects. The resulting call stack is labelled with Next, indicating that a proper execution step was performed
- Finally, if the top element of the call stack is an execution state, then the current instruction is executed. To this end, it is first checked whether a sufficient amount of gas (more than 0) is available (line 8). If this is not the case, the exception state is entered directly. Otherwise, the current opcode is extracted using using the function getOpcode with the code from the execution environment and the current pc (line 10). The function matches on this opcode as well as on the structure of the stack. In this way it can be checked in one go whether a sufficient amount of arguments for each opcode is on the stack. This allows for treating the positive cases of the instruction execution in separation while all other cases (where the number of arguments are not sufficient) are catched by the wildcard case in line 13.

For illustrating, how the semantics of the instructions can be defined, we left some examples (such as ADD, PUSH and JUMP).

Tasks

Fill in the semantics of the remaining instructions. To this end, you should replace all occurrences of `magic()` by a correctly typed term that models the semantics specified in `semantics.pdf`.

Notes

In addition to the simple step function, we define a version step_simp of it that strips of the execution outcome and use this function to define nsteps that executes a given call stack under a transaction environment for a predefined number of steps.

We used this function to write some test cases that you can use for checking your semantics and that can be found in `Tests.fst`. `Tests.fst` loads the file `Project2.fst`, so make sure that both files are located in the same directory and that `Project2.fst` executes successfully in order to execute the file `Tests.fst`. (Of course you can also just copy the tests to your `Project2.fst` file). You can do some sanity checks about your semantics by checking whether the lemmas test1 to test4 can be proven.

Exercise 2.2: Termination (4 points)

In Ethereum, the execution of each transaction is guaranteed to terminate. This should also be reflected in our semantics, so we define a function steps that iterates the step function till reaching a final state (indicated by the step outcome Stop). Your task is to prove that this function is total (so that it terminates on all potential arguments).

Tasks

Specify a function getDecArgList that given a well-formed call stack outputs a list of natural numbers that represents the 'size' of the call stack. More precisely, it should hold that this list decreases (when interpreted lexicographically) with each step of the execution.

Questions

Explain why the measure that you implemented decreases with each step. To this end, group the different cases of the step function according to the elements of the output list of getDecArgList that decrease in these cases.

Notes

Defining lexicographic orderings in F⋆ is a bit tedious. For this reason, we already defined a function getLexFromList that transforms the lists that result from getDecArgList to lexicographically interpreted ones. We also already introduced the corresponding decreases clause for the function steps. Once you correctly defined getDecArgList to produce decreasing lists, the definition of steps should type-check without further modification.

Hints

Recall that the the smart contract execution in Ethereum is guaranteed to terminate thanks to the gas mechanism. Think carefully about to which extent the gas decreases in each step. If there are cases where it does not decrease, think about what could be the argument for termination instead.

Exercise 2.3: Final states (4 points)

In the definition of the step function, the execution halts in case that a terminal state on the empty call stack is reached. We make this characterization of final call stacks explicit by defining a the function isFinal: callstack → Tot bool that decides whether a call stack is final.

We now consider two properties of final call stacks:

1. No matter how many steps are performed on a final call stack, the call stack does not change anymore. This property is formalized in the Lemma nsteps_stop. Prove this Lemma by replacing `admit()` in the function body by a term that type-checks.
2. Complementary to the previous property, it also holds that if a call stack does not change within one step then it must be final. This property shall be formulated in the Lemma progress.

Tasks

1. Prove the Lemma nsteps_stop by replacing `admit()` in the function body by a term that type-checks.
2. Formulate the Lemma progress and prove it correct.

Questions

1. Perform paper proofs of the two Lemmas. You do not need to spell out all the details, but you should at least give a justifying argument for those steps automatically performed by F⋆.

2. Why does the Lemma nsteps_stop uses propositional equality (==) instead of the usual equality (=)?

Exercise 2.4: Uniqueness of call stacks (4 points)

Finally, we plug the results of the last two exercises together. The overall goal of this exercise is to prove that all call stacks along one execution are unique as long as there is progress. Formally this result is stated in the Lemma callstacks_unique.

For proving this result, we exploit that we have already found a function (namely getDecArgList) that assigns a unique list to each call stack. To be able to use this result explicitly, we formulate the Lemma order_decreases that states that whenever a call stack cs' is reachable from a well-formed call stack cs within at least one (proper) step, then the measure assigned by getDecArgList decreases.

Tasks

Prove the lemmas order_decreases and callstacks_unique.

Questions

1. Perform paper proofs of the two Lemmas. You do not need to spell out all the details, but you should at least give a justifying argument for those steps automatically performed by F*.

2. Why do we prove that getDecArgList decreases with each step while we are only interested in the fact that it assigns different values to different call stacks? Could you directly prove the following Lemma?

```
val order_ineq: (n: nat) → (te: tenv) → (cs: callstack{wellformed cs}) → (cs': call stack
    )
→ Lemma (requires (nsteps n te cs == cs' ∧ n > 0 ∧ ¬(isFinal cs) ))
          (ensures (getDecArgList cs' ≠ getDecArgList cs))
```

If yes, insert the proof in `Project2.fst`, else explain why the direct proof fails.

Hints

You might need to use previously defined Lemmas.

Exercise 2.5: Exception Propagation (4 points)

As shortly discussed in the lecture, we simplified the treatment of gas in the provided semantics. On the one hand side, the execution of each instruction uniformly costs one unit of gas. On the other hand side, when performing a CALL instruction, all gas available is given to the execution of the callee. One implication of this simplification is that our resulting semantics shows a form of exception propagation. More precisely this means that whenever an exception occurs (so an exception state is entered), then also the caller state will (immediately) fail. This is as when an exception state is entered, all gas is consumed by the callee. As the caller did not 'save' any gas for further executions when calling, after a failed call, the caller will be left with 0 gas which will result in an exception at the next instruction that the caller will try to execute after the call. We will prove that a call stack whose top element is an exception state, will reach a final exception state within $2 * n$ steps, where n is the size of the remaining call stack. This statement is formulated in the Lemma exception_prop

Tasks

Prove the Lemma exception_prop.

Questions

1. Perform a paper proof of the Lemma. You do not need to spell out all the details, but you should at least give a justifying argument for those steps automatically performed by F*.
2. Could you also prove this Lemma for another number of steps? Justify your answer!

Submission Instructions

Please submit your solution by *24.07.2020* on TUWEL.

Your submission must be a single archive `your-group-number.zip` containing the following files:

- `report.pdf` – A `.pdf` file containing the answers to the questions. This file should be generated using the `.tex` template provided on TUWEL.
- `Project2.fst`– the F* file extended with the solutions required in the tasks.