Appendix to a simulator of Solidity-style smart contracts in the theorem prover Agda

1 August 2023

Fahad Alhabardi
Department of Computer Science
Swansea University
Swansea, United Kingdom
fahadalhabardi@gmail.com

Anton Setzer
Department of Computer Science
Swansea University
Swansea, United Kingdom
a.g.setzer@swansea.ac.uk

ABSTRACT

This paper serves as the appendix to a simulator of Solidity-style smart contracts in the theorem prover Agda [1]. It provides a detailed explanation of the process involved in converting code written in Solidity into Agda.

CCS CONCEPTS

• **Theory of computation** → Logic and verification; Type theory; Interactive computation; • **Security and privacy** → Logic and verification; Formal security models; • **Computing methodologies** → Distributed algorithms; Modeling methodologies.

KEYWORDS

Agda, smart contracts, Solidity, theorem prover, Ethereum, interface, simulator, blockchain

Fahad Alhabardi and Anton Setzer. 2023. Appendix to a simulator of Solidity-style smart contracts in the theorem prover Agda 1 August 2023. 4 pages. https://tinyurl.com/githubSimulatorSolidityAgda

A TRANSLATON OF SOLIDITY INTO AGDA

This section describes the translation from the Solidity language into the theorem prover Agda. One disadvantage of Solidity are problems with security. For instance, Solidity-written smart contracts are vulnerable to reentrancy attacks [2]. To reduce these risks, developers must be knowledgeable about safe coding practices and carefully verify their contracts before deployment. In our work, we achieve a higher level of security by verifying programs in Agda. An advantage of Agda is that it allows to write and verify programs in the same system in which it is written. This avoids any translation errors from one system to another. There are two approaches to translating Solidity into Agda. One approach is to use a compiler that translates Solidity code to Agda, which would be a major project which goes beyond the current paper. Another approach is to convert Solidity code into Agda manually. In our work, we are using manual translation, and we will explain this translation using examples.

In our approach we restrict ourselves to most commonly used features of Solidity. We omit floats, which are not fully implemented in Solidity language yet; rationals, which are not used commonly yet; and function types.

Arrays can be represented as constant functions that return a message representing the data type. We represent unsigned integers as nat i and signed integers x as a pairs which are represented as lists (nat b) (nat i), where b is 0 for negative and 1 for positive, and i = x if x is positive, and i = - x if x is negative. In both cases i will be range restricted, with the range given by the underlying Solidity types. Addresses are represented as range restricted unsigned integers. Unicode characters and numerations can be represented as unsigned integers with a range given by the data type. Range restricted unsigned integers are represented as natural numbers, where, when using them, a case distinction needs to be made whether the arguments are in range or not. If they are out of range, we raise an exception in accordance with Solidity ≥ 0.8. Similarly, if a message isn't a representation of an element of the data type in question (e.g. in case of signed integers the msg is not a list of length 2 with the first element being 0 or 1), we raise an exception. We represent byte arrays as list x, where x consists of elements nat y, and y is a range restricted natural number corresponding to the value of the array. We represent arrays as list x in which each array element is defined as a message representing the element of its type. We also represent contracts by the address where it is located, i.e. as an unsigned integers within a range. Furthermore, we implement strings as an array of range restricted integers in which each integer is the character's ASCII code. In addition, we implement maps as pure functions that take an element of the domain as an argument represented as a message and return the result of the map applied to it as a message. Finally, we define functions as functions of the contract, which take a message representing the list of arguments and return the result as a message (or in case of multiple returned elements a list of the results as a message).

A.1 Simple Simulator

This section will illustrate an example demonstrating the process of translating Solidity into Agda.

- **In Solidity:**
  We implement a contract called CounterExample, which includes a variable called counter of type uint16 and a function called increment function (increment()), which is used to increment the counter value by 1. The counter variable is initialised to 0 (the default in Solidity). The definition of the contract of CounterExample in Solidity is as follows:

```
pragma solidity >=0.8.2 <0.9.0;

contract CounterExample {

  uint16 counter;

  function increment() public{
    counter ++;}}
```

- **In Agda:**
  To translate the CounterExample in Agda, we define auxiliaries functions to allow us to represent our code in Agda. First, in Solidity, we declared the counter of type uint16, which has a minimum value is 0, and a maximum value is 65535. (We have tested the example as well with uint256 in Agda, we use here the smaller number for presentation purposes). In Agda, we define the Max_Uint function, which has a maximum value of 65535, and the counter is initialised to 0 as initial value (which is the default initialisation in Solidity).
  The definition of Max_Uint is as follows:

```
Max_Uint : ℕ
Max_Uint = 65535
```

  In our example testLedger, we have three fields at address 1:
  – We set the balance (amount) for testing purposes to 40 wei.
  – We have two functions fun:
    * The first is "counter", which represents a variable. This variable is initialised to nat 0.
    * The second is "increment". It will look up the current address, so that it knows which address to refer to. Then it calls the counter function and obtains the old counter value. After obtaining the result returned by the counter, the function makes an anonymous case distinction on whether the element returnd is of the form (nat n) or not, using (syntax λ{ ··· }): if the old counter is a number, it checks whether it is less the Max_Uint. If yes it updates "counter" to the constant function (const) returning suc *oldcounter* (increment by 1); In all other cases, an error is raised.
      The definition of testLedger is as follows:

```
testLedger 1 .amount              = 40
testLedger 1 .fun "counter" m     = const 0 (nat 0)
testLedger 1 .fun "increment" m =
  exec currentAddrLookupc λ addr →
  exec (callc addr "counter" (nat 0))
  λ{(nat oldcounter) →
    (if oldcounter < Max_Uint
    then exec (updatec "counter"
      (const (suc oldcounter)))
      (λ _ → return (nat (suc oldcounter)))
    else
    error (strErr "out of range error"));
  _ →
     error (strErr "counter returns not a number")}

testLedger ow .amount = 0
testLedger ow .fun ow' ow''
    = error (strErr "Undefined")
```

For other addresses, the balance (amount) will be 0 and the functions (fun) will raise an error.

The simple simulator can handle mappings, but this requires mappings to be represented as a list of pairs, which needs to be encoded and decoded using Agda – a cumbersome process which makes verification difficult.

The complex model can deal with mappings more directly by representing them as pure functions. Therefore, in this section, our solidity code does not include a mapping data type.

A.2 Complex simulator

In this section, we will provide an example that shows how to translate Solidity code into Agda code using the complex simulator.

- **In Solidity:**
  We introduce a contract named "Voting_Example" with two variables. The first is a mapping named "checkVoter", which maps addresses to Boolean values, and determines whether a voter is allowed to vote. The second is a mapping named "voteResult", which maps uint to uint values. The "voteResult" mapping stores number of voters' votes with each key representing a candidate as an unsigned integer value.
  Then, we create the addVoter() function, which takes an address of a voter as input and returns a Boolean value, and adds the voter to "checkVoter". The addVoter() function first checks if the address is already in the mapping using the "require" statement. If the address is already present, the function will raise an error. In contrast, if the address is not in the mapping, the function will set the value of checkVoter for the argument of the function to true and then return true.
  Next, we define deleteVoter(), which is similar to the addVoter() function, but it requires the mapping to return true for the voter, and if yes, sets the mapping for the voter to false.
  Finally, we develop the vote() function, which returns a Boolean value. First, the vote() function checks if the voter is eligible to vote, by checking the result of the "checkVoter" mapping. If the voter is allowed to vote, it will first set the mapping for the voter to false, then it increments the voterResult by 1. If not, it will raise an exception.
  The definition of Voting_Example contract is as follows:

```
pragma solidity >=0.8.2 <0.9.0;

contract Voting_Example {
 mapping(address => bool) public checkVoter;
 mapping(uint => uint) public voteResult;


```

```
8   function addVoter(address user) public returns (
        bool) {
9    require(!checkVoter[user],"Voter already exists"
        );
10       checkVoter[user] = true;
11       return true;}
12
13  function deleteVoter(address user) public returns
        (bool) {
14   require(checkVoter[user],"Voter does not exist")
        ;
15       checkVoter[user] = false;
16       return false;}
17
18  function vote(uint candidate) public returns (
        bool) {
19    require(checkVoter[msg.sender], "The voter is
         not allowed to vote");
20       checkVoter[msg.sender] = false;
21       voteResult[candidate] += 1;
22       return true;}}
```

- **In Agda:**
  We translate the Voting_Example contract from Solidity into the following Agda code testLedger:

```
testLedger 1 .amount = 100
testLedger 1 .purefunction "checkVoter" msg =
    checkMsgInRangePure Max_Address msg λ voter
    → theMsg (nat 0)
testLedger 1 .purefunction "voteResult" msg =
    checkMsgInRangePure Max_Uint msg λ voter
    → theMsg (nat 0)
testLedger 1 .purefunctionCost "checkVoter" msg = 1
testLedger 1 .purefunctionCost "voterResult" msg = 1
testLedger 1 .fun "addVoter" msg      =
  checkMsgInRange Max_Address msg λ user →
  exec (callPure 1 "checkVoter" (nat user))
  (λ _ → 1) λ msgResult →
  checkMsgOrErrorInRange Max_Bool msgResult
    λ {0 ->
    exec (updatec "checkVoter"
    (addVoterAux user) λ oldFun oldcost msg → 1)
    (λ _ → 1) (λ _ → return 1 (nat 1));
    (suc _) → exec (raiseException 1
    "Voter already exists")(λ _ → 1)(λ ())}

testLedger 1 .fun "deleteVoter" msg =

  checkMsgInRange Max_Address msg λ user →
  exec (callPure 1 "checkVoter" (nat user))
  (λ _ → 1) λ msgResult →
  checkMsgOrErrorInRange Max_Bool msgResult
    λ {0 → exec (raiseException 1
    "Voter does not exist")(λ _ → 1)(λ ());
    (suc _) → exec (updatec "checkVoter"
    (deleteVoterAux user) λ oldFun oldcost msg → 1)
    (λ _ → 1) (λ _ → return 1 (nat 0))}

testLedger 1 .fun "vote" msg =
  checkMsgInRange Max_Uint msg λ candidate →
  exec callAddrLookupc (λ _ → 1)
  λ addr →
    exec (callPure 1 "checkVoter" (nat addr))
    (λ _ → 1) λ msgResult →
    checkMsgOrErrorInRange Max_Bool msgResult
      λ b → voteAux addr b candidate


testLedger 0 .amount    = 100
testLedger 3 .amount    = 100
testLedger 5 .amount    = 100
testLedger ow .amount   = 0
testLedger ow .fun ow' ow"
  = error (strErr "Undefined") ⟨ ow » ow · ow' [ ow"]⟩
testLedger ow .purefunction ow' ow" = err (strErr "Undefined")
testLedger ow .purefunctionCost ow' ow" = 1
```

In the testLedger example, there are four fields located at address 1:

- The balance of the contract (amount) has been set to 100 wei for testing purposes. The same applies to contracts 0, 3, and 5.
- There are two pure functions (purefunction) available:
  - The pure function "checkVoter" initially calls the checkMsgInRangePure function to check whether the message is a number in the range of addresses. A continuation function is applied to the resulting address, if the message is a number within the designated range. In this particular case, it returns the message (nat 0) for false. If the message is outside the range, an error message is returned. An error message is also returned if the message is not a number. It is initialised with the value of 0. The definition of checkMsgInRangePure is as follows:

```
checkMsgInRangePure : (bound : ℕ) → Msg
  → (ℕ → MsgOrError) → MsgOrError
checkMsgInRangePure bound (nat n) fn =
  if n < bound
  then (fn n)
  else err (strErr "Pure function result out of range")
checkMsgInRangePure bound (msg +msg msg₁) fun =
  err (strErr "Pure function didn't return a number")
checkMsgInRangePure bound (list l) fun =
  err (strErr "Pure function didn't return a number")
```

  - The pure function "voteResult" checks as "checkVoter" that the argument is a number, and checks that it is in the range of uint. It returns for all candidates nat 0 as the number of votes.
- We have two pure function costs (purefunctionCost), which are "checkVoter" and "voteResult" that are used to calculate the pure function for each process. These costs are both initialized with a value of 1.
- We have three functions (fun):
  - The "addVoter" function will invoke the checkMsgInRange function to verify if the argument is an address

within the acceptable range of addresses. It will then call "checkVoter" applied to the address, and then check using the checkMsgOrErrorInRange function, if the result is a number within the range of Booleans, i.e. it is ≤ 1 (if not raises an exception). It then makes a case distinction on the result. If the result is 0 for false, the "addVoter" function will update the pure function "checkVoter" by using the addVoterAux function. Otherwise, the result is suc _, i.e. false, and it will raise an exception. The addVoterAux function updates the previous "checkVoter" function: if the argument is equal to the new address, it will return nat 1 for true. Otherwise, it will return the previous result of "checkVoter". The definition of addVoterAux is as follows:

```
addVoterAux : ℕ → (Msg → MsgOrError)
                → Msg → MsgOrError
addVoterAux newaddr oldCheckVoter (nat addr) =
  if    newaddr ≡ᵇ addr
  then theMsg (nat 1) - return 1 for true
  else  oldCheckVoter (nat addr)
addVoterAux ow ow' ow'' =
  err (strErr "The argument of checkVoter
               is not a number")
```

The definitions of checkMsgInRange and checkMsgOrErrorInRange functions are similar to the definitions of checkMsgInRangePure, we just give their signatures:

```
checkMsgInRange : (bound : ℕ) → Msg
  → (ℕ → SmartContract Msg)
  → SmartContract Msg

checkMsgOrErrorInRange : (bound : ℕ) → MsgOrError
  → (ℕ → SmartContract Msg)
  → SmartContract Msg
```

* The "deleteVoter" function deletes a voter. It works similarly to the "addVoter" function, but it checks whether "checkVoter" for the argument is true, and if yes, sets it to 0 for false.
* "vote" does the following: It first calls the checkMsgInRange function to check whether it is a number within the range of uint. If not it will raise an exception. Otherwise, it looks up the calling address and then evaluates the result of the ("checkVoter") applied to the calling address. Then it will use the checkMsgOrErrorInRange function to check if the result is a number in the range of Booleans. It then calls the voteAux function. The voteAux determines whether the outcome was 0 for false or suc _ for true. If it is false, it will return an error message. Otherwise, it will first set the "checkVoter" for the voter to false (to prevent multiple voting). Then it will increment the pure function ("voteResult") applied to the candidate by one.
  The full definition of voteAux is as follows:

```
voteAux : Address → ℕ → (candidate : ℕ)
          → SmartContract Msg
voteAux addr 0 candidate
  = error
  (strErr "The voter is not allowed to vote")
  ⟨ 0 » 0 · "Voter is not allowed to vote"
          [ nat 0 ]⟩
voteAux addr (suc _) candidate =
  exec (updatec "checkVoter"
  (deleteVoterAux addr) λ oldFun oldcost msg → 1)
  (λ _ → 1)
  (λ x → (incrementAux candidate))
```

For other addresses and other normal and pure functions for contract 0, it will return an error ("Undefined"). For other pure functions, purefunctionCost will set the gas cost to 1.

REFERENCES

[1] Fahad F. Alhabardi and Anton Setzer. 2023. A simulator of Solidity-style smart contracts in the theorem prover Agda. To appear in Proceedings of ICBTA 2023, http://www.icbta.net/.

[2] Noama Fatima Samreen and Manar H. Alalfi. 2020. Reentrancy Vulnerability Identification in Ethereum Smart Contracts. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, London, ON, Canada, 22–29. doi: https://doi.org/10.1109/IWBOSE50093.2020.9050260.