A simulator of Solidity-style smart contracts in the theorem prover Agda

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 extends the previous paper [6] by implementing two blockchain simulators of Solidity-style smart contracts – a simple and a complex one, using the interactive theorem prover Agda. In the previous article [6], we built a simple and complex abstract model of Solidity-style smart contracts in Agda. These models had many features, such as calling different smart contracts, supporting the ability to call different smart contracts, and providing simple and complex instructions. Because of the use of coalgebras for representing smart contracts they supported loops and conditionals, using the support of those features for coalgebraic programs in Agda. The complex model supported gas costs and pure functions, similar to the Solidity language.

In this paper, we implement and design interfaces which allow to interactively interact with users in the simple and complex models. This makes use of the fact that Agda is as well a dependently typed programming language. Therefore we can write interactive programs which are running in the same language in which we will in a future next step verify smart contracts, avoiding the translation of programs which could be a source of errors. The simple blockchain simulator we have created can call other contracts, transfer funds to specific contracts, and update contracts. The complex blockchain simulator has in addition features that can deal with more complex blockchain instructions, support gas costs, and evaluate and update pure functions.

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

ACM Reference Format:
Fahad Alhabardi and Anton Setzer. 2023. A simulator of Solidity-style smart contracts in the theorem prover Agda. In *2023 6th International Conference on Blockchain Technology and Applications (ICBTA) (ICBTA 2023), December 15–17, 2023, Xi'an, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3651655.3651656

ICBTA 2023, December 15–17, 2023, Xi'an, China

ACM ISBN 979-8-4007-0867-1/23/12
https://doi.org/10.1145/3651655.3651656

1 INTRODUCTION

Blockchains are distributed digital ledgers that operate without a centralised authority. A blockchain is a form of database that is decentralised, reliable, and difficult to utilise for fraud. Once a transaction publishes in the blockchain network, no one can change it except by changing the ledger, e.g., by a 50% attack, which is unlikely in the case of Ethereum.

Blockchain technology is the foundation for Ethereum [13] and other cryptocurrencies, enabling peer-to-peer transactions through a distributed public record. The technology supports other applications as well, such as smart contracts [40]. Thus, the technology offers countless possibilities.

Ethereum was first suggested by Vitalik Buterin [13] in 2013. Since its launch in 2015, the project has rapidly risen to become one of the most widely used blockchain platforms. Ethereum is used to build many decentralised applications (DApps).

Smart contracts in blockchain are programs that automatically run when specific criteria are met [26]. Due to the immutability of blockchains, once a smart contract has been issued, it cannot be modified in any way. Therefore, before deploying smart contracts, it is necessary to verify their correctness and security to avoid possibly triggering substantial financial losses.

Many languages can be used to write Ethereum's smart contracts, such as Solidity [15] and Vyper [41], both high-level languages. Other cryptocurrencies have their own languages [9].

Formal verification is one way to detect weaknesses and vulnerabilities at an early stage in smart contracts. It is also a promising way to provide security guarantees by mathematically verifying designs that use various mathematical and logical methods [22]. There are many approaches to formal verification, such as theorem proving [21] and modelling checking [33]. In theorem proving, it is possible to verify the correctness of different types of systems, such as smart contracts [21].

In our approach, we use the interactive theorem prover Agda [3] to execute the blockchain simulator. One interesting aspect of Agda is that it can be used as a dependently typed programming language [34]. Therefore, together with the fact that it is also a theorem prover, Agda allows the writing of verified programs, where the verification takes place in the same language in which the program is written. This prevents translation problems that might arise because the verified program differs from the implemented one.

However, what we do not do at this stage is to translate solidity programs automatically into Agda, this is done manually.

As a background for this paper we note as well that the main unit of the Ethereum currency is ether, where 1 ether = 10^{18} wei.

The remaining parts of this paper are as follows: In Sect. 2, we present related work. In Sect. 3, we briefly introduce the proof assistant Agda and the library used to build our interface. In Sect. 4, we implement and design two simulators of Solidity-style smart contracts. We end with a conclusion and future work in Sect. 5.

Git repository. This work was created and formalised using the proof assistant Agda. All of the Agda code shown in this paper was derived from the type-checked Agda code. The source code can be found in [37]. The git repository also contains a pdf file [8] that explains how to translate Solidity code manually into Agda. There we show as well how to deal with arithmetic overflow – in the main paper for brevity we use unrestricted integers.

2 RELATED WORK

In this section, we describe the work that have focused on verifying, formalizing, and analyzing smart contracts using theorem provers such as Agda [3], Coq [11, 14], and Isabelle/HOL [24]. Then, we provide research using model checking, tools, and a framework to analyze and verify smart contracts. Our earlier articles [5–7] analysed the relevant literature in more depth, this part only provides a brief update.

Some efforts verify smart contracts using the theorem prover. Ayoade et al. [10] proposed and developed a framework for rewriting Ethereum bytecode without access to the source code. Their approach enables bytecode modifications to Ethereum without a high-level language's source code. They used the Coq theorem prover to implement and verify the Ethereum virtual machine code. Zheng et al. [42] developed Lolisa, an intermediate specification language for Ethereum smart contracts in Coq. Lolisa has a major subset of Ethereum's Solidity programming language in its formal syntax and semantics. Lolisa's formal syntax uses a stronger static type system than Solidity to improve type safety. Lolisa also incorporates general-purpose programming language capabilities and a substantial fraction of Solidity syntax components. Thus, translating Solidity programs into Lolisa is possible. Lolisa is naturally generalizable and can express various programming languages. Finally, Coq interprets Lolisa's syntax and semantics. Thus, Coq can execute and verify Lolisa's smart contracts symbolically. In [5, 7], we have proved the correctness of Bitcoin script in Agda and developed a methodology to obtain a human-readable weakest precondition of Hoare logic. This helps to fill the validation gap between user needs and the formal specification of smart contracts. We have applied this methodology to two standard scripts, *Pay to Public Key Hash (P2PKH)* and *Pay to Multisig (P2MS)*. Marmsoler et al. [27] propose an executable denotational semantics of Solidity in the Isabelle/HOL proof assistant. Their formal semantics creates the groundwork for an interactive program verification environment for the Solidity program and enables checking Solidity programs by symbolic execution.

Many types of research analyse and verify smart contracts using frameworks, model checking, and tools. Mavridou et al. [29] developed FSolidM, a framework for creating more secure contracts on ETH via a graphical interface for developing finite state machines that can immediately be converted into ETH smart contracts. Nam et al. [32] presented a novel formal verification approach using an alternating-time temporal logic (ATL) model to investigate blockchain smart contracts developed through solidity. They used MCMAS [25], an effective ATL model checker that verifies multi-agent systems. They aimed at identifying subtle defects in real smart contracts. So et al. [38] presented a static analysis tool, VeriSmart, to ensure the arithmetic safety of Ethereum smart contracts. They focused on detecting arithmetic bugs, such as integer over/underflows and division-by-zeros, because smart contracts typically involve many arithmetic operations which are major sources for security vulnerabilities.

3 BACKGROUND

3.1 A brief introduction to the Agda theorem prover

Agda [12, 39] is a language that implements Martin-Löf's type theory [28], with expanding records and modules. It serves as a functional programming language and proof assistant.

In the following we provide a concise overview of Agda; more comprehensive information on the proof assistant Agda is available in our previous papers [5–7].

Agda's user interface is based on Emacs, which has been beneficial for interactively developing and verifying proofs [12]. When using Agda, programmers can write their code incrementally, allowing them to leave certain parts unfinished. With the help of Agda's type-checking tool, they can receive useful guidance on completing these sections step by step.

Agda is based on dependent types. $(x : A) \to B$ is the type of functions which takes an element $x : A$ and map it to an element of B, where B may depend on x. Agda supports hidden arguments with syntax $\{x : A\} \to B$ – in this case we can omit the application of the function to its argument, if it can be inferred uniquely by the compiler. If it cannot be inferred, one can provide the hidden argument explicitly, writing $f\ \{a\}$ for the application of f to hidden argument a. Nondependent function types are instances of dependent types with no dependency, and we write $A \to B$ for the type of functions from A to B. In Agda one writes $\forall x \to B for (x : A) \to B$ and $\forall\{x\}\to B$ for $\{x : A\}\to B$, if A can be inferred uniquely by Agda. Furthermore, _ denotes arguments which are not used, or can be inferred uniquely.

Apart from dependent function types, Agda supports inductive (data) types and record types, where the latter can be coinductive types. As an example of a data type, we define ErrorMsg, which will be used in the simple simulator later in this article. It can be used to describe a variety of error messages as follows:

```
data ErrorMsg : Set where
  strErr    : String → ErrorMsg
  numErr    : ℕ → ErrorMsg
  undefined : ErrorMsg
```

We define three different error message constructors in the ErrorMsg data type. The strErr constructor is used for error messages given by a string, numErr is used for error messages given by a natural number, and undefined is used for reporting the error message

"undefined". We can define functions by using pattern matching on these constructors, e.g.

```
errorMsg2Str : ErrorMsg → String
errorMsg2Str (strErr s)    = s
errorMsg2Str (numErr n) = show n
errorMsg2Str undefined  = "undefined"
```

The function errorMsg2Str converts an error into a readable string suitable for printing or displaying to users.

An example of a record type is as follows:

```
record StateIO : Set where
      constructor ⟨_ledger,_initialAddr,_gas⟩
      field
        ledger      : Ledger
        initialAddr : Address
        gas         : ℕ
```

It has a constructor ⟨_ledger,_initialAddr,_gas⟩ which is mix fix (_ denote the argument positions of this function). It constructs from elements of Ledger, Address, ℕ an element of StateIO. The projection of a record types to its field (also called observation) is defined using the dot notation, for instance if x : StateIO, then x .ledger : Ledger. Elements of a record type can be defined by copattern matching (see [1]): We can introduce an element a : StateIO by determining its components a .ledger, a .initialAddr, and a .gas.

Record types can refer directly or indirectly via other types to themselves. If we add the word coinductive, then an element of it can be defined using copattern matching by using full recursion referring to itself, as long as in the chain from on element to itself there is at least one observation. This allows to define coalgebras in Agda. They are infinite structures which don't break normalisation in Agda (which means every term in Agda has finite normal formal), because in order to unfold a term one needs to apply one of the observations (fields) of the record type. For more details see [1].

Agda employs different levels of types, with the smallest level being called Set for historical reasons [23, 30]. In this article, we use apart from Set the next higher type level Set_1. Set_1 encompasses all sets (via an explicit embedding), but as well Set itself and types formed from it such as Set → Set.

3.2 Interface Library

The representation of interactive programs as the IO monad [31] in dependent type theory was developed by the second author and Peter Hancock in a sequence of articles [16–20], see as well [1, Sect. 4]. All the Agda code in this section is taken from [2, Sect. 4] (with minor modifications). Interaction between a program with, for example, an operating system dealing with IO can be created as a series of commands (elements of Command) issued by the program to the operating system. For each of these commands the operating system returns a response (an element of Response). The type Response will depend on the command being issued. As shown in Figure 1, the interactive program gives a question to the world using a command, and the world answers with some response. Then the next command is issued depending on that response etc.

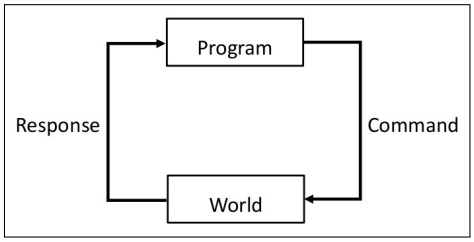

Figure 1: Interactive program. Source: [36]

Consequently, the interface for interaction consists of a set of commands Command and a set of responses Response, depending on commands. We define the type of interfaces in Agda as a record type. Since its fields include the type Set, the type IOInterface of interfaces resides in the type level Set_1 above Set. IOInterface has two fields: Command of type Set and a field Response which depending on a command returns the set of responses (Source: [2]):

```
record IOInterface : Set₁ where
  field
    Command : Set
    Response  : Command → Set
```

In our interactive program interactions consist of input and output of strings. We use the interface ConsoleCommand to deal with console interface, which has two commands getLine and putStrLn. The command getLine : ConsoleCommand has no argument, and reads a user input line. The response returned by the system is the String typed in by the user, therefore we define
ConsoleResponse getLine = String.
The command putStrLn command has one argument of type String, namely the string to be printed, so we define
putStrLn : String → ConsoleCommand.
The response is just the information that the string has been printed (we assume this command always succeeds so there is no error message), so the information is the void information given by the one element type Unit, and we define putStrLn s = Unit.

The complete definition is as follows (Source: [2]):

```
data ConsoleCommand : Set where
  putStrLn : String → ConsoleCommand
  getLine : ConsoleCommand

ConsoleResponse : ConsoleCommand → Set
ConsoleResponse (putStrLn s) = Unit
ConsoleResponse getLine      = String
```

The console interface consoleI is the interface consisting of ConsoleCommand and ConsoleResponse (Source: [2]):

```
consoleI : IOInterface
consoleI .Command = ConsoleCommand
consoleI .Response  = ConsoleResponse
```

We define the set of interactive programs generically for any IOInterface. We will abstract from it, which is written in Agda by using the lines (Source: [2])

```
module _
  (I : IOInterface ) (let C = I .Command) (let R = I .Response)
```

This line unpacks as well the interface into its two commands: the set of commands (the set C) and the response set R of the abstracted interface I.

We now define the type IO of interactive programs mutually recursively as a coinductive record IO together with the data type IO'. This definition is coinductive, since interactive program in principal are allowed to run an infinite non-terminating sequence of interactions. In accordance with Moggi's IO monad [31], interactive programs may as well terminate, returning an element of type A. We use here sized types, which allow to define elements of coalgebras in a more generic way which without sized types would be rejected in this form by Agda's termination checker, even though they are productive. See [1, Sect. 6] for a detailed explanation of sized types. As a first approximation the user might ignore all arguments referring to the type Size in the following (most elements of type Size will be inferred automatically by Agda when writing Agda code). One could view sized types as a form of gas, where a program of size n is allowed to be unfolded at most n times.

IO has a field (or observation) force, which returns an element of type IO'. It has as well for convenience a lazy constructor delay which turns an element p of IO' into an element of IO' A, namely an element, which when we apply .force to it returns p. Elements of IO' are either terminating programs return' a, returning an element of type A. Or they are of the form exec' c p, which means they execute command $c : C$, and continue, if a response $r : R\ c$ is returned, executing program $p\ r$.

The full definition is as follows (Source: [2]):

```
record IO (i : Size) (A : Set) : Set where
  coinductive
  constructor delay
  field
    force : {j : Size< i} → IO' j A

data IO' (i : Size) (A : Set) : Set where
  exec'   : (c : C) (f : R c → IO i A) → IO' i A
  return' : (a : A) → IO' i A
```

Note that elements of IO are not directly of the form (return' a) or (exec' c p) – instead we need to apply observation .force to it in order to unfold it into one these two choices. Otherwise an element of IO representing a infinite sequence of interactions would reduce to an infinite term, whereas Agda requires each correctly typed term to reduce to a finite normal form. In order to unfold an IO' once, we need to pay the price of applying once .force to it, breaking a possibly infinite reduction sequence.

We define the monad operation bind [31] for the IO monad in order to combine programs as follows (Source: [2]):

```
_>>='_ : ∀{i}{A B : Set}(m : IO' I i A) (k : A → IO I (↑ i) B) → IO' I i B
exec' c f >>=' k = exec' c λ x → f x >>= k
return' a >>=' k = (k a) .force

_>>=_ : ∀{i}{A B : Set}(m : IO I i A) (k : A → IO I i B) → IO I i B
(m >>= k) .force = m .force >>=' k
```

The program p >>= q first executes program p. If it terminates, returning $a : A$, then it continues executing $q\ a$. If that program terminates the overall program terminates as well, returning the response returned when executing $q\ a$.

4 A SIMULATOR OF SOLIDITY-STYLE SMART CONTRACTS IN AGDA

4.1 Simulator of the simple model

In the previous work [6], we developed a simple model of Solidity-type smart contracts. This model had as commands transferc for transferring money from one contract to another, callc for calling a function (in object-orientation terminology a method) of another contract, updatec for updating one of the functions of the contract, we are making the call from. In addition we had commands for looking up the address of the current contract and the balance (wei) of any contract.

These commands formed the set of commands (CCommands) of an interactive program. We defined for each command the set of possible responses CResponse returned in response to issuing this command. Since CResponse depends on CCommands, it's type is that of a function from CCommands to the type of sets Set.

Smart contracts are given as elements of a coinductive type Contract. They possibly infinitely often issue a command (element of CCommands) and depending on the response (CResponse) obtained when executing it execute the next command. They are similar to the interactive programs described in Sect. 3.2, however the commands are executed on the ledger instead of asking the user for a response via an operating system. The resulting responses are obtained from the ledger, and the ledger changes as a result of the execution. The execution forms an object-model of Ethereum, similar to the object models developed by the second author with coauthors in [1, 35]. Smart contracts are given as coinductive interactive programs, executing this sequence of commands, with the commands computed depending on the response to the previous command. In Agda they are represented as a coinductive record type SmartContract (in [6] we called it SmartContractExec, but thought this shorter name improves readability of this paper). SmartContract is defined similar to the type of interactive programs, but it is running on the ledger instead of an operating system.

Contracts (Contract) are records consisting of the balance (amount of wei) given as a natural number and a function from function names to SmartContract. A ledger (Ledger) is a function from addresses (natural numbers) to Contract.

In this section we build, based on our previous work [6], a simulator of the simple model of Solidity-style smart contracts. The simple simulator supports the above mentioned operations, i.e. calling functions from other contracts, updating functions, transferring funds, and obtaining the money balance in other smart contract. However, at this level, the simple simulator does not include an explicit cost of gas – that will be included in the complex model. Without gas, execution of smart contracts may not terminate. This is reflected by the fact that in the file `Ledger-Simple-Model.agda` in the git repository [37], two auxiliary functions used in the evaluation of smart contracts are under the pragma
{-# NON_TERMINATING #-}

Agda requires that the programs terminate in order to be consistent as a theorem prover, and uses a termination checker to check for termination. Using this pragma, we can break the termination checker. Because of these nontermination problems, the simulator, which makes use of the evaluation function, is not guaranteed to terminate (an example would be a contract calling itself with the same argument it is called). This problem will be repaired in the complex model, where we add an explicit gas limit. We could as well restrict in the simple model the number of recursive calls to a certain number, having the effect of a simple form of gas limit.

Arguments of functions and the results returned will be serialised messages, which allows the encoding of complex data structures as byte arrays. In order to abstract from the process of serialisation, we use a data type Msg representing serialised messages:

```
data Msg : Set where
  nat : ℕ        → Msg
  list : List Msg → Msg
```

The simplest elements of it are of the form nat *n*, representing the serialised number *n*. All inputs of functions will be elements of Msg and outputs will be elements of MsgOrError (which includes in addition to Msg an error element). In Solidity, numbers are restricted integers or restricted positive integers. Integers can be, for instance, represented as pairs consisting of a Boolean value representing the sign and number, whereas Booleans can be represented as the natural numbers 0 and 1. When sending or receiving elements of these data types, one needs in Agda to check that the numbers are in the range given by the data type and, if they are not, deal with it according to what is done in Solidity, which means for the current version 0.8.21, to raise an exception.

In the simple model, variables are represented as constant functions, which returns the value of the variable. In order to change a variable, we update the function representing it.

We illustrate the simulation interface by referring to the following example testLedger. It will have only one defined contract at address 1 which has a variable "counter", and a function to increment this variable by 1. In order to demonstrate other features, we have as well a function which transfers 10 wei to contract 0.

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 n) → exec (updatec "counter" (const (suc n)))
                  λ _ → return (nat (suc n));
      _ → error (strErr "counter returns not a number")}
testLedger 1 .fun "transfer" m
          = exec (transferc 10 0) λ _ → return m
testLedger ow .amount        = 0
testLedger ow .fun ow' ow" = error (strErr "Undefined")
```

In testLedger, we define a contract at address 1. We set the balance (field amount) to 40. We have 3 functions: The first function "counter" represents a variable. This variable is initialised with the value nat 0.

The second function is "increment", which will increment the variable of "counter" by 1. In order to be independent of the address, on which it is deployed, it will look up the current address (which returns 1). Then it will lookup the value of the variable by applying the function "counter" to an arbitrary value (we choose nat 0). The function might return any serialised message, but it should, if correctly used, return a serialised number. So we make a case distinction on the serialised message: if it is of the form (nat n), we update "counter" to the constant function returning (nat (suc n)). Otherwise we return an error message, since "counter" returned not a serialised number. The final function is "transfer", which will transfer 10 wei to Address 0. All other contracts are initialised to have a balance of 0, with all functions being undefined, i.e. returning an error message ("Undefined"). In the same way, all other functions (given by other strings) of contract 1 apart from the three functions mentioned before return the same error message. We use here the fact that in Agda patterns are evaluated in sequence. The first matching pattern will be used to determine the result, and any future pattern after a matching pattern will be ignored. So the line testLedger *ow* .amount= 0 applies to all arguments *ow* (for otherwise) which haven't been covered by a previous pattern, in this case all natural numbers except for 1.

Next, we develop our interface menu (mainBody) of the simple simulator Solidity-style smart contract, which has four options a user can select from to interact with the ledger, as shown in Figure 2. These options are as follows: "Option 1", execute a function of a contract; in case of our example testLedger, we can look up the value of "counter", by executing it, increment that variable by 1, or execute the transfer given; "Option 2" allows changing the calling address from which other contracts are called (the initial value used is 0); "Option 3" looks up the balance of any contract; and "Option 4" terminates the program.

```
Please choose one of the following options:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
                4- Terminate the program.
```

Figure 2: The simple blockchain simulator program interface.

The mainBody takes two arguments, *ledger* and *callAddr*. The definition of mainBody is as follows:

```
mainBody : ∀{i} → Ledger → (callAddr : Address)
           → IOConsole i Unit
mainBody ledger callAddr .force
  = WriteString'
    "Please choose one of the following options:
      1- Execute a function of a contract.
      2- Look up the balance of a contract.
      3- Change the calling address.
      4- Terminate the program."
  λ str → (GetLine >>= λ str →
  if str      == "1"
    then executeLedger ledger callAddr
```

```
  else (if str == "2"
    then executeLedgercheckamount ledger callAddr
  else (if str == "3"
    then executeLedgerChangeCallingAddress ledger callAddr
  else (if str == "4"
    then WriteString "The program will be terminated"
  else WriteStringWithCont "Please enter 1,2,3 or 4"
      λ _ → mainBody ledger callAddr))))
```

We define mainBody mutually recursively with auxiliary functions for the different options. In case of `"Option 1"` these are executeLedgerStep2 - executeLedgerStep5. Function executeLedger asks a user to enter the calling address, i.e. the contract of which we want to execute a function. Then executeLedgerStep2 will check whether the result was a number. If yes, it asks for the function name to be executed (given as a string) After that, executeLedgerStep2 will call executeLedgerStep3 to ask the user to enter the argument of the function name as a natural number (we currently only support arguments of functions which are serialised natural numbers, in a future version, we will allow arbitrary serialised messages as inputs). Then, executeLedgerStep4 will check whether the user entered indeed a number, and if yes, return the result of evaluating the function applied to the message using executeLedgerStep5 and return to the start menu. Here the result returned will be the number returned if it was a number, a message indicating it was a list, if the result was a list, and otherwise, the error message. Note that in case of an error the ledger will return to its initial state except for gas used in the failed execution being deducted.

When converting a user input to a natural number, we get an element of Maybe ℕ having elements (just n) for a successful converted natural number and nothing, if the string was not a natural number. Our code makes therefore a case distinction on whether the result of that conversion was nothing or (just n).

For example, as shown in Figure 3, we selected `"Option 1"`, and execute at address 1 function `"counter"` with argument 1. The result is nat 0 (returning the content of the variable counter).

```
Please choose one of the following options:
               1- Execute a function of a contract.
               2- Look up the balance of a contract.
               3- Change the calling address.
               4- Terminate the program.
1
Enter the calling address
1
Enter the function name (e.g. counter, increment, transfer)
counter
Enter the argument of the function as a natural number
0
The result of execution is nat 0
```

Figure 3: Execute a function of a contract (option 1).

The types of executeLedger and the auxiliary functions are as follows:

```
executeLedger : ∀{i} → Ledger → (callAddr : Address)
  → IOConsole i Unit
executeLedgerStep2 : ∀{i} → Ledger → (callAddr : Address)
  → Maybe ℕ → IOConsole i Unit
executeLedgerStep3 : ∀{i} → Ledger → (callAddr : Address) → ℕ
  → FunctionName → IOConsole i Unit
executeLedgerStep4 : ∀{i} → Ledger → (callAddr : Address) → ℕ
  → FunctionName → Maybe ℕ → IOConsole i Unit
executeLedgerStep5 : ∀{i} → MsgAndLedger
  → (callAddr : Address) → IO' consoleI i Unit
```

In case of `"Option 2"`, the program will ask for the address to look up the balance for, print out the result and return to the starting menu.

For example, as shown in Figure 4, when selecting `"Option 2"` and entering the calling Address 1, the result will be the available money, 40 wei, in Address 1, and it will return to the main interface.

```
Please choose one of the following options:
               1- Execute a function of a contract.
               2- Look up the balance of a contract.
               3- Change the calling address.
               4- Terminate the program.
2
Enter the address of the contract you want to look up the balance
1
The available money is 40 wei in address 1
```

Figure 4: Look up the balance of a contract (option 2).

The types of executeLedgercheckamount and the auxiliary function executeLedgercheckamountAux are as follows:

```
executeLedgercheckamount : ∀{i} → Ledger
  → (callAddr : Address) → IOConsole i Unit
executeLedgercheckamountAux : ∀{i} → Ledger
  → (callAddr : Address) → Maybe ℕ → IOConsole i Unit
```

For `"Option 3"`, which is defined by function executeLedgerChangeCallingAddress, the system asks for the new calling address, and once obtained executes the same code as for `"Option 1"`.

For instance, as shown in Figure 5, when selected, `"Option 3"` will ask to enter the new calling address; in our case, we enter the new calling address 1, the function `"increment"`, and the function's argument as 0. The result will be (nat 1), and the operation increments the variable `"counter"` to 1.

```
Please choose one of the following options:
               1- Execute a function of a contract.
               2- Look up the balance of a contract.
               3- Change the calling address.
               4- Terminate the program.
3
Enter the new calling address
1
Enter the calling address
1
Enter the function name (e.g. counter, increment, transfer)
increment
Enter the argument of the function as a natural number
0
The result of execution is nat 1
```

Figure 5: Change the calling address (option 3).

The types of executeLedgerChangeCallingAddress and executeLedgerChangeCallingAddressAux are as follows:

```
executeLedgerChangeCallingAddress : ∀{i} → Ledger
  → (callAddr : Address) → IOConsole i Unit
executeLedgerChangeCallingAddressAux : ∀{i} → Ledger
  → (callAddr : Address) → Maybe Address → IOConsole i Unit
```

Finally we define the main function, as follows:

```
main : ConsoleProg
main = run (mainBody testLedger 0)
```

The main function serves as the entry point when executing the Agda program. It is in charge of starting the program and executing its main logic. In this scenario, the main function applies mainBody to the testLedger and starts with calling Address set to 0. This creates an interactive program. run translates it into a native IO program. Agda's compiler MAlonzo [4] will then create an interactive program. The compiled executable will execute the interactive program as described above.

4.2 Simulator of the complex model

In the previous work [6], we developed the complex model. One main feature is that it has in addition to normal functions pure functions. Pure functions are given as functions of type
Msg → MsgOrError
Pure functions don't interact with other functions or make updates, and directly compute from its input either the result or an error. Pure functions can be updated from the contract they belong to. Standard functions get a new command updatec which allows to update a pure function by referring to its previous definition. This is useful to represent maps in Solidity, which are finite functions from input to output. We represent maps as pure functions. We can update them for one argument to a new value, by checking whether the argument is equal to the updated argument (in which case we return the updated result) or not (in which case we return the result of the previous version of this function).

Another addition of the complex model is the use of gas cost. Since we cannot control the cost of execution of functions in Agda from Agda, we require that the user states the cost for computing the various operations explicitly as part of all commands of normal functions. Note that the main purpose of the model is to verify smart contracts. Whether a contract is correct depends on making realistic choices for the gas cost.

The main change to accommodate the gas cost is in the following definition of the commands for the complex model:

```
data SmartContract (A : Set) : Set where
  return : ℕ → A → SmartContract A
  error  : ErrorMsg → DebugInfo → SmartContract A
  exec   : (c : CCommands) → (CResponse c → ℕ)
         → (CResponse c → SmartContract A)
         → SmartContract A
```

The constructor return for terminating a SmartContract has an extra argument of type ℕ which determines the cost for computing the result – the value returned could be computed by a very time consuming computation, and the argument states that cost. The constructor exec, which creates an program which executes one command and depending on the result returned executes a continuing SmartContract has an extra argument of type

```
CResponse c → ℕ
```

which determines the cost of computing the continuation. The gas cost of each instruction will be at least one (technically we increment any gas cost stated by one), and therefore termination is guaranteed. The code actually passes Agda's termination checker, overcoming the obstacle of nontermination of the simple model, which compromises consistency of the theorem prover.

In the complex model SmartContract has new commands

```
data CCommands : Set where
  callPure : Address → FunctionName → Msg → CCommands
  updatec : FunctionName → ((Msg → MsgOrError)
          → (Msg → MsgOrError)) → ((Msg → MsgOrError)
          → (Msg → ℕ) → Msg → ℕ) → CCommands
  raiseException : ℕ → String → CCommands
  -  ... in addition commands as in the simple model

CResponse : CCommands → Set
CResponse (callPure addr fname msg) = MsgOrError
CResponse (updatec fname fdef cost) = ⊤
CResponse (raiseException _ str)    = ⊥
```

callPure calls a pure function. updatec updates a pure function by referring to the previous version of it, using argument
((Msg → MsgOrError) → (Msg → MsgOrError))
Furthermore, we add an explicit error command raiseException with an explicit cost and error message.

Using the implementation of the complex model in our previous work [6], we expand the simple simulator into the complex one, adding more complex options for the user: to evaluate pure functions, and to change and check the gas limit.

In order to demonstrate our interface, we develop a simple voting example (testLedger). The current example has only one candidate. We leave it to the user to enhance this example to a more advanced one involving multiple candidates (by making the counter and vote functions depend on a candidate number).

The definition of testLedger is as follows:

```
testLedger 1 .amount = 100
testLedger 1 .purefunction "checkVoter" msg = theMsg (nat 0)
testLedger 1 .purefunction "counter" msg    = theMsg (nat 0)
testLedger 1 .purefunctionCost "checkVoter" msg = 1
testLedger 1 .fun "addVoter" msg    =
  exec (updatec "checkVoter" (addVoterAux msg) λ _ _ _ → 1)
    (λ _ → 1) λ _ → return 1 msg
testLedger 1 .fun "deleteVoter" msg =
  exec (updatec "checkVoter" (deleteVoterAux msg) λ _ _ _ → 1)
    (λ _ → 1) λ _ → return 1 msg
testLedger 1 .fun "vote" msg
  = exec callAddrLookupc (λ _ → 1)
  λ addr → exec (callPure addr "checkVoter" (nat addr))
    (λ _ → 1) λ check → voteAux addr check msg
testLedger 0 .amount  = 100
testLedger 3 .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 our example testLedger at address 1, we have three fields, as follows:

- Amount (amount) (i.e. balance of the contract in wei) is 100;
- We define two pure functions (purefunction) as follows:
  - "checkVoter", which determines for its argument whether the argument (which is assumed to be (nat *n*) for an address *n*) represents a voter who is allowed to vote; Booleans are represented as (nat 0) for false and (nat (suc *n*)) for true. "checkVoter" initially always returns (nat 0) for false, i.e. nobody is allowed to vote.
  - "counter" is a variable counting the number of votes for the only candidate. It is initialised with the value of 0.
- We define pure function cost (purefunctionCost) to calculate the pure function for each process.
- Three functions (fun) are added as follows:
  - "addVoter" updates the pure function"checkVoter" to allow the address represented by its argument to vote. It makes use of the following function, which determines the new value "checkVoter" by checking whether the argument was updated or not, and if not referring to the old version of "checkVoter":

```
addVoterAux : Msg → (Msg → MsgOrError)
                  → Msg → MsgOrError
addVoterAux (nat 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 " You cannot add voter ")
```

  - "deleteVoter" deletes a voter. The implementation is almost the same as "addVoter", except for that it sets "checkVoter" to 0 for false, in case the argument is the voter to be deleted.
  - "vote" does the following:
    It looks up the calling address first, then evaluates the ("checkVoter") function applied to it to check whether the caller is allowed to vote or not. Then it invokes voteAux to make a case distinction on whether the result was true, false, or a message not representing a number. If the voter is permitted to vote, the counter (pure function ("counter")) is increased by one, and the voter is deleted from "checkVoter" to prevent double voting. Otherwise, an error will be returned. The full definition of voteAux is as follows:

```
voteAux : Address → MsgOrError → (candidate : Msg)
              → SmartContract Msg
voteAux addr (theMsg (nat zero)) candidate
  = error
  (strErr "The voter is not allowed to vote")
  ⟨ 0 » 0 · "Voter is not allowed to vote" [ nat 0 ]⟩
voteAux addr (theMsg (nat (suc n))) candidate
  = exec (updatec "checkVoter"
    (deleteVoterAux (nat addr)) λ _ _ _ → 1)(λ _ → 1)
    (λ x → (incrementAux1 (theMsg candidate)))
voteAux addr (theMsg ow) candidate
  = error (strErr "The message is not a number")
  ⟨ 0 » 0 · "Voter is not allowed to vote" [ nat 0 ]⟩
voteAux addr (err x) candidate
  = error (strErr " Undefined ")
  ⟨ 0 » 0 · "The message is undefined" [ nat 0 ]⟩
```

For other addresses, and for other normal and pure functions for contract 0, we will return an error ("Undefined") that includes the debug information, such as the last call address, current address, last function name call, and the last function argument.

We define our main menu of the complex simulation interface mainBody, as shown in Figure 6. We have created three additional options ("Option 4", "Option 5", and "Option 6") to complement the existing ones in the simple simulator. These new options aid in verifying the voting example and show the gas consumption at each stage. Below are explanations for all seven options:

- "Option 1", "Option 2", and "Option 3", which are functions similarly to the simple simulator. However, these options have been redefined to incorporate gas cost and pure function.
- "Option 4" may be utilized to update the gas limit used when calling smart contracts.
- "Option 5" may be used to verify the amount of gas left before or after each operation.
- "Option 6", which we use to evaluate pure functions. In Solidity, pure functions do not call other functions. When called externally, these functions do not incur any gas cost. However, gas costs will be required if they are called from an internal function.
- "Option 7", which terminates the simulator.

```
Please choose one of the following:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
                4- Update the gas limit.
                5- Check the gas limit.
                6- Evaluate a pure function.
                7- Terminate the program.
```

Figure 6: The complex blockchain simulator program interface.

The state of the system will be given by an element *stIO* : StateIO defined below. The mainBody function depends on that state variable *stIO*. The definition of the complex simulator (mainBody) is as follows:

```
mainBody : ∀{i} → StateIO → IOConsole i Unit
mainBody stIO .force
  = WriteString' ("Please choose one of the following:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
```

```
                4- Update the gas limit.
                5- Check the gas limit.
                6- Evaluate a pure function.
                7- Terminate the program.") λ _ →
  GetLine >>= λ str →
  if    str == "1" then executeLedger stIO
  else (if str == "2" then executeLedger-CheckBalance stIO
  else (if str == "3" then
          executeLedger-ChangeCallingAddress stIO
  else (if str == "4" then executeLedger-updateGas stIO
  else (if str == "5" then executeLedger-checkGas stIO
  else (if str == "6" then executeLedger-purefunction stIO
  else (if str == "7" then
          WriteString "The program will be terminated"
  else WriteStringWithCont
      "Please enter a number 1 - 7"
      λ _ → mainBody stIO ))))))
```

We develop the StateIO, a record type that defines the current state of computation. It comprises three fields:

- ledger is the current ledger on which the calculation will be executed.
- initialAddr is the initial address used to initialise the calculation; in our case, we initialised it to 0, but it can be changed by using "Option 3".
- gas is the quantity of gas left for use in the calculation.

The constructor for StateIO requires three parameters, which are the values that are to be used for each of the three fields. The definition of StateIO is as follows:

```
record StateIO : Set where
      constructor ⟨_ledger,_initialAddr,_gas⟩
      field
        ledger      : Ledger
        initialAddr : Address
        gas         : ℕ
```

As an example, the line of code below establishes the element of StateIO that has the ledger as our voting example (testLedger), 0 as initial address, and a gas amount of 20 wei:
⟨ testLedger ledger, 0 initialAddr, 20 gas⟩
As we mentioned before, "Option 1", "Option 2", and "Option 3" have comparable functions and structures as the simple simulator, with the inclusion of gas cost. For instance, as shown in Figure 7, when selecting "Option 3", entering a new calling address 1 instead of the previous address 0, it will start to execute the contract function "Option 1" by entering the "addVoter" function and the argument of the function 1. The result will be that the initial address is 1, the call address is 1, the argument of the function name is (nat 1), the remaining gas is 16 wei, and the value returned is (theMsg (nat 1)).

In addition, we have created the executeLedger-updateGas function along with its corresponding auxiliary function (executeLedgerStep-updateGasAux) mutually recursively. These

```
Please choose one of the following:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
                4- Update the gas limit.
                5- Check the gas limit.
                6- Evaluate a pure function.
                7- Terminate the program.
3
Enter a new calling address as a natural number
1
Enter the called address as a natural number
1
Enter the function name (e.g. addVoter, deleteVoter, vote)
addVoter
Enter the argument of the function name as a natural number
1
 The result is as follows:

 The initial address is 1
 The called address is 1
 The argument of the function name is (nat 1)
 The remaining gas is 16 wei ,  The function returned (theMsg 1)
```

Figure 7: Change the calling address in the complex blockchain simulator (option 3).

functions allow for the implementation of "Option 4", which enables updating the gas limit. Upon execution of executeLedgerStep-updateGasAux, the user will be prompted to input a new value for the gas amount. If the input is successful, executeLedgerStep-updateGasAux will be called, and the function will return both the new and old gas limit values. For example, as shown in Figure 8, when selecting "Option 4", then entering the new gas limit 30. The result is that the gas limit has been updated successfully, the new gas limit is 30 wei, and the old value is 20 wei.

```
Please choose one of the following:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
                4- Update the gas limit.
                5- Check the gas limit.
                6- Evaluate a pure function.
                7- Terminate the program.
4
Enter the new gas amount as a natural number
30
The gas amount has been updated successfully.
 The new gas amount is  30 wei and the old gas amount is 20 wei
```

Figure 8: Update the gas limit in the complex blockchain simulator (option 4).

The type of executeLedger-updateGas and its auxiliary function (executeLedgerStep-updateGasAux) are as follows:

```
executeLedger-updateGas          : ∀{i} → StateIO
                                 → IOConsole i Unit
executeLedgerStep-updateGasAux : ∀{i} → StateIO → Maybe ℕ
                                 → IOConsole i Unit
```

For "Option 5", we have developed a mutually recursive function called executeLedger-checkGas. This function ensures that the gas limit is verified after updating to the new value, as illustrated in Figure 9.
The definition of executeLedger-checkGas is as follows:

```
Please choose one of the following:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
                4- Update the gas limit.
                5- Check the gas limit.
                6- Evaluate a pure function.
                7- Terminate the program.
5
 The gas limit is 30 wei
```

Figure 9: Check the gas limit in the complex blockchain simulator (option 5).

executeLedger-checkGas : ∀{*i*} → StateIO
→ IOConsole *i* Unit

Moreover, we develop mutually recursively executeLedger-purefunction together with its auxiliary functions (executeLedger-purefunction, executeLedger-purefunction0, executeLedger-purefunction1, executeLedger-purefunStep1-2, executeLedger-purefunStep1-3, and executeLedger-purefunStep1-4), in order to implement "Option 6". As an example, after using "Option 1" to add 1 as a voter, we proceed to select "Option 6", by entering calling address 1, called address 1, and the pure function "checkVoter" along with its argument 1. The result is that the initial address is 1, the called address is 1, and the pure function returns theMsg (nat 1), signifying that it is true, as shown in Figure 10.

```
Please choose one of the following:
                1- Execute a function of a contract.
                2- Look up the balance of a contract.
                3- Change the calling address.
                4- Update the gas limit.
                5- Check the gas limit.
                6- Evaluate a pure function.
                7- Terminate the program.
6
Enter the Calling Address as a natural number
1
Enter the Called Address as a natural number
1
Enter the function name (e.g. checkVoter, counter)
checkVoter
Enter the argument of the function name as a natural number
1
The information you get is below:

 The initial address is 1
 The called address is 1
 The pure function returns (theMsg 1)
 The pure function cost returns 1
```

Figure 10: Evaluate a pure function in the complex simulator at (option 6).

The types of executeLedger-purefunction and its auxiliary functions are as follows:

executeLedger-purefunction : ∀{*i*} → StateIO → IOConsole *i* Unit
executeLedger-purefunction0 : ∀{*i*} → StateIO
→ Maybe Address → IOConsole *i* Unit
executeLedger-purefunction1 : ∀{*i*} → StateIO → IOConsole *i* Unit
executeLedger-purefunStep1-2 : ∀{*i*} → StateIO
→ Maybe Address → IOConsole *i* Unit
executeLedger-purefunStep1-3 : ∀{*i*} → StateIO
→ (*calledAddr* : Address) → Maybe FunctionName
→ IOConsole *i* Unit
executeLedger-purefunStep1-4 : ∀{*i*} → StateIO
→ (*calledAddr* : Address) → FunctionName
→ Maybe ℕ → IOConsole *i* Unit

Finally, we define the main function to run the program:

main : ConsoleProg
main = run (mainBody (⟨ testLedger ledger, 0 initialAddr, 20 gas⟩))

The main function has one single argument, and it will run the mainBody, which includes an argument with a tuple of three values - the ledger, the initial address, and the gas limit. The mainBody function will use our ledger (testLedger), start from the initial address 0, and have the gas limit of 20 wei.

In the git repository [37], we demonstrate our complex simulator by an example. The example shows that, if we change first the calling address to address 1, and then try to vote, the vote is rejected, because voter 1 has not been added yet. If we then add voter 1, and vote (with calling address 1), the vote succeeds, and the counter is incremented by 1. If contract 1 votes again, it is rejected, and we see that the number of votes stays at 1.

5 CONCLUSION AND FUTURE WORK

This paper presents two blockchain simulators of Solidity-style smart contracts in the theorem prover Agda. The first is the simple simulator, which has simple instructions for transferring money to specific addresses and executing and updating smart 'contracts. The second is the complex simulator, which has more features and complex instructions, supports gas costs, uses a pure function similar to the Solidity language, and displays better error messages than the simple simulator. The simulator is written in the interactive theorem prover Agda, in the same language in which we plan to carry out the verification. Therefore, there is no explicit translation needed from the simulated program to the verified program, avoiding translation errors.

In future work, we will verify contracts which can be run in the simple and complex blockchain simulators by using weakest preconditions [5, 7]. In our previous articles [5, 7]. we already implemented, proved, and verified Bitcoin's smart contracts in the Agda proof assistant using weakest preconditions. We plan to expand the work in those previous articles in order to verify Solidity-style smart contracts.

REFERENCES

[1] Andreas Abel, Stephan Adelsberger, and Anton Setzer. 2017. Interactive programming in Agda – Objects and graphical user interfaces. *Journal of Functional Programming* 27 (Jan 2017), e8. https://doi.org/10.1017/S0956796816000319 https://doi.org/10.1017/S0956796816000319.

[2] Andreas Abel, Stephan Adelsberger, and Anton Setzer. 2017. Interactive programming in Agda – Objects and graphical user interfaces. *Journal of Functional Programming* 27 (2017), e8. doi: https://doi.org/10.1017/S0956796816000319.

[3] Agda Community. 2005. Welcome to Agda's documentation! Retrieved 19 June 2023, Availabe from https://agda.readthedocs.io/en/v2.6.2/.

[4] Agda Wiki. 2011. MAlonzo. Retrieved 19 June 2023, Availabe from http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Docs.MAlonzo.

[5] Fahad Alhabardi, Bogdan Lazar, and Anton Setzer. 2022. Verifying Correctness of Smart Contracts with Conditionals. In *2022 IEEE 1st Global Emerging Technology Blockchain Forum: Blockchain & Beyond (iGETblockchain)*. IEEE, Irvine, CA, USA, 1–6. doi: https://doi.org/10.1109/iGETblockchain56591.2022.10087054.

[6] Fahad Alhabardi and Anton Setzer. 2023. A model of Solidity style smart contracts in the theorem prover Agda. https://fahad1985lab.github.io/papers/A_model_of_Solidity-style_smart_contracts_in_the_theorem_prover_Agda.pdf To appear in: To appear in: The IEEE International Conference on Artificial Intelligence, Blockchain, and Internet of Things, (AIBThings), August 2023. Availabe from https://fahad1985lab.github.io/papers/A_model_of_Solidity-style_smart_contracts_in_the_theorem_prover_Agda.pdf.

[7] Fahad F. Alhabardi, Arnold Beckmann, Bogdan Lazar, and Anton Setzer. 2022. Verification of Bitcoin Script in Agda Using Weakest Preconditions for Access Control. In *27th International Conference on Types for Proofs and Programs (TYPES 2021) (LIPIcs, Vol. 239)*. Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:25. doi: https://doi.org/10.4230/LIPIcs.TYPES.2021.1.

[8] Fahad F. Alhabardi and Anton Setzer. 2023. Appendix to "A simulator of Solidity-style smart contracts in the theorem prover Agda". , 4 pages. Available from https://github.com/fahad1985lab/A_simulator_of_Solidity-style_smart_contracts_in_the_theorem_prover_Agda and https://csetzer.github.io/articles/Appendix_to_a_simulator_of_Solidity-style_smart_contracts_in_the_theorem_prover_Agda.pdf.

[9] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. 2020. Verification of smart contracts: A survey. *Pervasive and Mobile Computing* 67 (2020), 101227. doi: http://dx.doi.org/10.1016/j.pmcj.2020.101227.

[10] Gbadebo Ayoade, Erick Bauman, Latifur Khan, and Kevin Hamlen. 2019. Smart Contract Defense through Bytecode Rewriting. In *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, Atlanta, GA, USA, 384–389. doi: https://doi.org/10.1109/Blockchain.2019.00059.

[11] Yves Bertot and Pierre Castéran. 2004. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer, Berlin, Heidelberg. doi: https://doi.org/10.1007/978-3-662-07964-5.

[12] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda – A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics*. Springer, Berlin, Heidelberg, 73–7. doi: https://doi.org/10.1007/978-3-642-03359-9_6.

[13] Vitalik Buterin. 2014. Ethereum: A next-generation smart contract and decentralized application platform. Retrieved 19 June 2023, Availabe from https://ethereum.org/en/whitepaper.

[14] Coq Community. 2023. The Coq Proof Assistants. Retrieved 15 June 2023, Availabe from https://coq.inria.fr/.

[15] Ethereum Community. 2016. Solidity documentation. Retrieved 19 June 2023, Availabe from https://docs.soliditylang.org/en/v0.8.16/.

[16] Peter Hancock and Anton Setzer. 1999. The IO monad in dependent type theory. 13 pages. Electronic proceedings of the workshop on dependent types in programming, Göteborg, 27-28 March 1999.

[17] Peter Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Computer Science Logic*, Peter Clote and Helmut Schwichtenberg (Eds.). Lecture Notes in Computer Science, Vol. 1862. Springer, Berlin / Heidelberg, 317–331. http://dx.doi.org/10.1007/3-540-44622-2_21

[18] Peter Hancock and Anton Setzer. 2000. Specifying interactions with dependent types. Electronic proceedings of the Workshop on subtyping and dependent types in programming, Ponte de Lima, Portugal. 13 pp. Retrieved 19 June 2023, Availabe from http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.

[19] Peter Hancock and Anton Setzer. 2004. Interactive Programs and Weakly Final Coalgebras (Extended Version). In *Dependently typed programming (Dagstuhl Seminar Proceedings, 04381)*, T. Altenkirch, M. Hofmann, and J. Hughes (Eds.). Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, 1 – 30. http://drops.dagstuhl.de/opus/volltexte/2005/176/

[20] Peter Hancock and Anton Setzer. 2005. Interactive programs and weakly final coalgebras in dependent type theory. In *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*, L. Crosilla and P. Schuster (Eds.). Clarendon Press, Oxford, 115 – 136. http://dx.doi.org/10.1093/acprof:oso/9780198566519.003.0007

[21] John Harrison, Josef Urban, and Freek Wiedijk. 2014. History of Interactive Theorem Proving.. In *Computational Logic (Handbook of the History of Logic, Vol. 9)*. North-Holland, Elsevier, Amsterdam, The Netherlands, 135–214. doi: https://doi.org/10.1016/B978-0-444-51624-4.50004-6.

[22] Katharina Hofer-Schmitz and Branka Stojanović. 2020. Towards formal verification of IoT protocols: A Review. *Computer Networks* 174 (2020), 107233. doi: http://dx.doi.org/10.1016/j.comnet.2020.107233.

[23] Antonius J. C. Hurkens. 1995. A simplification of Girard's paradox. In *Typed Lambda Calculi and Applications*. Springer, Berlin, Heidelberg, 266–278.

[24] Isabelle Community. 2023. Isabelle documentation. Retrieved 19 June 2023, Availabe from https://isabelle.in.tum.de/documentation.html.

[25] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. 2018. MCMAS: an open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer* 19, 1 (2018), 9–30. doi: https://doi.org/10.1007/s10009-015-0378-x.

[26] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. doi: http://dx.doi.org/10.1145/2976749.2978309.

[27] Diego Marmsoler and Achim D. Brucker. 2021. A Denotational Semantics of Solidity in Isabelle/HOL. In *Software Engineering and Formal Methods*. Springer, Cham, 403–422. doi: https://doi.org/10.1007/978-3-030-92124-8_23.

[28] Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*. Vol. 80. Elsevier, Amsterdam, The Netherlands, 73–118. doi: http://dx.doi.org/10.1016/S0049-237X(08)71945-1.

[29] Anastasia Mavridou and Aron Laszka. 2018. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *Financial Cryptography and Data Security*. Springer, Berlin, Heidelberg, 523–540. doi: https://doi.org/10.1007/978-3-662-58387-6_28.

[30] Albert R. Meyer and Mark B. Reinhold. 1986. "Type" is Not a Type. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida) *(POPL '86)*. Association for Computing Machinery, New York, NY, USA, 287–295. doi: https://doi.org/10.1145/512644.512671.

[31] Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55 – 92. https://doi.org/10.1016/0890-5401(91)90052-4

[32] Wonhong Nam and Hyunyoung Kil. 2022. Formal Verification of Blockchain Smart Contracts via ATL Model Checking. *IEEE Access* 10 (2022), 8151–8162. doi: https://doi.org/10.1109/ACCESS.2022.3143145.

[33] Zeinab Nehaï, Pierre-Yves Piriou, and Frédéric Daumas. 2018. Model-Checking of Smart Contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, Halifax, NS, Canada, 980–987. doi: http://dx.doi.org/10.1109/Cybermatics_2018.2018.00185.

[34] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Springer, Berlin, Heidelberg, 230–266. doi: https://doi.org/10.1007/978-3-642-04652-0_5.

[35] Anton Setzer. 2006. Object-oriented programming in dependent type theory. In *Conference Proceedings of TFP 2006*. Intellect Books, Bristol, 1 – 16. https://csetzer.github.io/index.html

[36] Anton Setzer. 2009. Interactive Programs in Agda. Swansea University. Retrieved 19 June 2023, Availabe from https://csetzer.github.io/slides/agdaimplementorsmeeting/agdaImplementorsMeetingGoeteborg2009/goeteborg2009AgdaIntensiveMeeting.pdf.

[37] Anton Setzer and Fahad Alhabardi. 2023. A simulator of Solidity-style smart contracts in the theorem prover Agda. Available from https://github.com/fahad1985lab/A_simulator_of_Solidity-style_smart_contracts_in_the_theorem_prover_Agda.

[38] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1678–1694. doi: https://doi.org/10.1109/SP40000.2020.00032.

[39] Aaron Stump. 2016. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, New York, US. doi: https://doi.org/10.1145/2841316.

[40] Nick Szabo. 1996. Smart Contracts: Building Blocks for Digital Markets. *EXTROPY: The Journal of Transhumanist Thought,(16)* 18, 2 (1996), 28. Availabe from https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[41] Vyper Team. 2017. Vyper documentation. Retrieved 19 June 2023, Availabe from https://vyper.readthedocs.io/en/stable/.

[42] Zheng Yang and Hang Lei. 2020. Lolisa: Formal Syntax and Semantics for a Subset of the Solidity Programming Language in Mathematical Tool Coq. *Mathematical Problems in Engineering* 2020 (2020), 1–15. doi: https://doi.org/10.1155/2020/6191537.