Termination-checked Solidity-style smart contracts in Agda in the presence of Turing completeness

Fahad F. Alhabardi[1] and Anton Setzer[2]

[1,2]Swansea University, Dept. of Computer Science
[1]fahadalhabardi@gmail.com https://fahad1985lab.github.io/
[2]a.g.setzer@swansea.ac.uk http://www.cs.swan.ac.uk/~csetzer/

Abstract

This paper is a further step in extending the verification of Bitcoin Script using weakest precondition semantics in our articles [6, 1, 5] to Solidity-style smart contracts. The first step is to develop a model, which is substantially more complex than that of Bitcoin Script because smart contracts in Solidity are object-oriented. This paper extends the simple model of Solidity-style smart contracts in Agda in our article [2] to a complex model. The main addition in the complex model is that it deals with the termination problem by adding a cost per instruction (gas cost) as implemented in Ethereum, therefore execution of smart contracts passes the termination checker of Agda.

One main application of blockchain are smart contracts. Smart contracts can be defined as programs that automatically run when specific predetermined criteria are met [16, 13].

Smart contracts face several challenges, particularly in terms of security [8]. All smart contract transactions and codes are immutable once published on the blockchain network. The only way to amend the clauses of an ongoing smart contract or to withdraw it is by using functions already provided by the original contract. Thus, the developers must ensure and verify the security of the code before publishing it on the blockchain in order to avoid any errors. Errors in smart contract programs have resulted in massive financial losses [14, 15].

One formal way to specify the validity of imperative programs is Hoare logic [11]: one defines pre- and postconditions as the required conditions on the state of a program before and after execution. Hoare logic works well for guaranteeing the safety of programs, i.e. that programs work correctly when executed according to requirements. A very stringent technique can identify errors early in the development phase [12]. However, it doesn't work very well for showing that a program is secure in the presence of malicious inputs. Our solution is to use weakest preconditions of Hoare logic instead. Weakest preconditions express that the conditions are not only sufficient but also necessary for the program to end up in a state fulfilling the postcondition. An example is that certain data needs to be present in order to obtain cryptocurrency coins.

In this paper, we extend the simple model of Solidity-style smart contracts in our previous paper [2] (see as well the simulator [3]) to a complex model. In the complex model, we add gas cost. We use the gas cost to guarantee termination – each instruction costs at least one unit of gas, and once all gas allocated has been consumed, the program terminates with an out of gas error. Using this idea we succeed it showing that our implementation of the execution mechanism of programs passes Agda's termination checker.

We work directly on Solidity code rather than on its compiled Ethereum Virtual Machine (EVM) code. Therefore we cannot use gas costs associated with EVM instructions, and instead add to each high level Solidity instruction a parameter which estimates the gas costs for its execution. Therefore verification depends on good estimates for these parameters.

As in our previous simple model, we have ordinary functions (corresponding to methods in the terminology of object-orientation). We encode the arguments and return values of functions

as elements of a message type, which allows as well to encode multiple arguments as single ones. In our settings, functions have only one argument and one return element of this message type. Ordinary functions are given by a coalgebraic definition, which consist of a possibly unbounded sequence of basic operations such as making a transfer, looking up the balance of an account, or making recursive calls to other functions. In addition to ordinary functions, we add view functions (functions which can be modified by ordinary functions but don't call other functions). Variables are represented as view functions. They are especially useful for representing variables which have the type of a mapping, which frequently occur in Solidity code. View functions are represented as simple functions in Agda, and, therefore, are elements of a data type different from that of ordinary functions. Ordinary functions have instructions for updating view functions, but are not able to update ordinary functions. Therefore we keep view functions and normal functions as separate entities. (In Solidity view functions are defined as ordinary functions, but with a restriction on their code). The gas cost of ordinary functions is given by the cost of the basic instructions involved during their execution. For view functions we need in addition functions which estimate the cost for their execution.

We start by defining the data type of contracts (Contract), which includes four fields: the balance of a contract (amount), its functions (fun), its view functions (viewFunction), and the estimated gas cost for executing a view function (viewFunctionCost). The definition of Contract is as follows:

```
record Contract : Set where
  field
    amount : Amount
    fun : FunctionName → (Msg → SmartContractExec Msg)
    viewFunction : FunctionName → Msg → MsgOrError
    viewFunctionCost : FunctionName → Msg → ℕ
```

Ethereum uses a simple model of mapping addresses to their state as opposed to the UTXO model (see e.g. [9], or our article [15]) used e.g. in Bitcoin which tracks the state to previous unspent transaction outputs. We call such a mapping for brevity a ledger. Strictly speaking it is the state of a ledger – a full ledger would include its history. The execution of a smart contract function in Ethereum only depends on the current state of the ledger without its history, and function calls are executed as one atomic operation which includes all its recursive calls and updates. Therefore the correctness of a smart contract in this setting relates only to the current state of the ledger. We define therefore a ledger as a function which maps addresses to contracts: Ledger = Address→ Contract

As in the simple model, we have an execution stack, which records currently open recursive calls. The elements of the execution stack (ExecStackEl) include the following fields: the address that made the last call (lastCallAddress), the address that was called (calledAddress), continuation which determines the next execution step to be executed depending on the message returned after the call to the function has been completed, funcNameexecStackE which is the last function called and the argument of the last function call (msgexecStackEl).

```
record ExecStackEl : Set where
  field lastCallAddress calledAddress : Address
        continuation : (Msg → SmartContractExec Msg)
        costCont              : Msg → ℕ
        funcNameexecStackEl : FunctionName
        msgexecStackEl        : Msg
```

The execution stack is a list of ExecStackEl. The state of the execution (StateExecFun) include the following fields: the ledger, the execution stack (executionStack), the initial address that initiated the current sequence (initialAddr), the last called made (lastCallAddr), the address which is called (calledAddr), the current code to be executed (nextstep), the gas left (gasLeft), and two extra fields that we use with debug information: funcNameexecStackE and msgexecStackEl.

```
record StateExecFun : Set where
field ledger          : Ledger
      executionStack  : ExecutionStack
      initialAddr lastCallAddr calledAddr : Address
      nextstep        : SmartContractExec Msg
      gasLeft         : ℕ
      funNameevalState : FunctionName
      msgevalState : Msg
```

In order to state the verification conditions in Hoare logic, we define the state of the system as given by the ledger and the address making the call. Pre- and post-conditions will be defined as predicates on this state. In order to accommodate with intermediate steps in the program execution, the program will be given by the code to be executed, the execution stack and the called address. In order to have a robust definition which works as well in the simple model where programs are not guaranteed to terminate, we define a relation expressing that during execution, the program starting in a start state terminates successfully in an end state. Then, we show that the precondition is a weakest precondition for the program to end in the postcondition state. A simple example is that in order for the amount in one contract to reach a certain value, a second contract (which triggered a transfer) must have had a sufficient balance. In a follow-up paper, we will show how to formally prove this in Agda, which reveals unexpected subtleties in the precise formulation of its precondition.

Related Work. For a detailed literature review see our article [4]. Some additional work to mention is the formalisation KEVM [10] of the EVM in the K framework, which directly formalises the low level Ethereum virtual machine. Our approach works instead directly on Solidity in order to support the derivation of human readable weakest preconditions. Annenkov et. al. [7] developed a framework ConCert for extracting smart contracts from Coq, and a testing framework that allows to detect specific high level exploits. In our work we define instead a direct semantics for Solidity style contracts based on weakest preconditions. There is extensive work such as [9] from researchers, many of whom are associated with IHOK, which studies and extends the unspent transaction model (UTXO). We have studied the UTXO model used in Bitcoin in [15]. In this paper, we use the model used in Ethereum, which instead directly maps addresses to balances. Ethereum uses transaction nonces instead of UTXOs in order to prevent replay attacks.

References

[1] Fahad Alhabardi, Bogdan Lazar, and Anton Setzer. Verifying correctness of smart contracts with conditionals. In *2022 IEEE 1st Global Emerging Technology Blockchain Forum: Blockchain & Beyond (iGETblockchain)*, pages 1–6, 2022. doi:10.1109/iGETblockchain56591.2022.10087054.

[2] Fahad Alhabardi and Anton Setzer. A simple model of smart contracts in Agda, January 2023. In Abstracts for Types 2023. URL: https://types2023.webs.upv.es/TYPES2023.pdf.

[3] Fahad Alhabardi and Anton Setzer. A simulator of Solidity-style smart contracts in the theorem prover Agda, August 2023. To appear in Proceedings of ICBTA 2023, Xi'an, China. http://

www.icbta.net/. International Conference Proceedings Series by ACM (ISBN: 979-8-4007-0867-1), ACM Digital library. URL: https://csetzer.github.io/articles/alhabardiSetzerICBTA2023.pdf.

[4] Fahad Alhabardi and Anton Setzer. A model of solidity-style smart contracts in the theorem prover agda. In *2023 IEEE International Conference on Artificial Intelligence, Blockchain, and Internet of Things (AIBThings)*, pages 1–10, Mount Pleasant, MI, USA, 2023. IEEE. doi:10.1109/AIBThings58340.2023.10292478.

[5] Fahad Alhabardi, Anton Setzer, Arnold Beckmann, and Bogdan Lazar. Verification techniques for smart contracts in Agda, June 2022. In Abstracts for Types 2022. URL: https://types22.inria.fr/programme/.

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

[7] Annenkov, Danil and Milo, Mikkel and Nielsen, Jakob Botsch and Spitters, Bas. Extracting smart contracts tested and verified in Coq. In *CPP 2021: Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 105–121, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437992.3439934.

[8] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer. doi:10.1007/978-3-662-54455-6_8.

[9] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The Extended UTXO Model. In *Financial Cryptography and Data Security*, pages 525–539, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-54455-3_37.

[10] Hildenbrandt, Everett and Saxena, Manasvi and Rodrigues, Nishant and Zhu, Xiaoran and Daian, Philip and Guth, Dwight and Moore, Brandon and Park, Daejun and Zhang, Yi and Stefanescu, Andrei and Rosu, Grigore. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018. doi:10.1109/CSF.2018.00022.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 585, October 1969. doi:10.1145/363235.363259.

[12] Lamport, Leslie and Schneider, Fred B. The "Hoare Logic" of CSP, and All That. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, April 1984. doi:10.1145/2993.357247.

[13] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978309.

[14] Zeinab Nehaï, Pierre-Yves Piriou, and Frédéric Daumas. 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)*, pages 980–987, 2018. doi:10.1109/Cybermatics_2018.2018.00185.

[15] Anton Setzer. Modelling Bitcoin in Agda. *CoRR*, abs/1804.06398, 2018. URL: http://arxiv.org/abs/1804.06398, arXiv:1804.06398.

[16] Nick Szabo. Smart contracts: Building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought*, 18(2), 1996. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.