Modelling Smart Contracts of Bitcoin in Agda

Anton Setzer[1]*and Bogdan Gabriel Lazar[1]†

Dept. of Computer Science, Swansea University, Swansea, UK

Abstract

This work is based on the first author's model of the transaction structure of Bitcoin using inductive-recursive data types in the theorem prover Agda. We extend this model by adding smart contracts written in the byte code language Script of Bitcoin. The goal is to use it for verifying the correctness of smart contracts. The use of non-local conditional instructions is solved without the use of jump addresses.

Cryptocurrencies are currently widely discussed. Many cryptocurrencies support smart contracts, a feature with big potential. Smart contracts are programs stored on the blockchain which are automatically executed when conditions are met, resulting in changes of data storage and monetary transactions. They are increasingly used for non-monetary applications. There is no legal framework at present to prevent the malicious execution of a smart contract. This contrasts with ordinary contracts where a legal framework exists, which void certain malicious uses of clauses written in a contract. Due to the absence of legal protection, smart contracts require a much higher level of correctness. Together with the fact that smart contracts are small and therefore easy to manage, and because mistakes might have substantial monetary consequences, smart contracts are a prime target for machine checked correctness proofs.

Bitcoin's language for smart contracts is the byte code language Script.[1] It is a Forth-like stack machine (Sect. 6 of [Ant17]). The instructions manipulate the stack, the only memory available. They may as well result in abortion of the program. In order to check signatures, Script can refer to a message extracted from the current transaction. Smart contracts might be executed after a certain amount of time and can therefore refer to the current time. Script does not have jumps; therefore, programs will always terminate. It has control flow statements, consisting of operations OP_IF/OP_NOTIF, OP_ELSE, OP_ENDIF. Depending on the top element of the stack after an OP_(NOT)IF the if-case or else-case (if it exists) is executed. This avoids miscalculation with forward jumps, a widespread problem in assembly languages.[2]

In [Set18, Set19] (see as well the PhD thesis [dS20]) the first author introduced a model of the transaction dag of Bitcoin, based on an inductive-recursive data structure. The transactions were defined inductively while recursively computing the set of unspent transaction outputs. Transactions consist of a list of inputs and a list of outputs. An input consists of a previous unspent transaction output (utxo), a public key, and a signature. An output consists of an amount, and an address. An input is correct if the public key hashes to the address. Furthermore, the signature needs to sign the relevant part of the message using the private key corresponding to the public key. In addition, there are global conditions for transactions such as that the sum of outputs needs to be less than the sum of inputs.

*a.g.setzer@swansea.ac.uk, http://www.cs.swan.ac.uk/~csetzer/

†lazarbogdan90@yahoo.com

[1]The cryptocurrency Ethereum has a high-level language Solidity for smart contracts, which on the blockchain is first compiled into the byte code of the Ethereum Virtual machine EVM. The EVM shares great similarities with Script. An attacker can directly target the EVM, therefore verification of smart contracts in Ethereum needs to address the byte code level. In this talk we will focus on Script, but techniques used there can be adapted to verify programs of the EVM.

[2]In contrast the EVM has arbitrary jumps, has reference to external memory, and replaces the conditionals of Script by conditional jumps.

In order to extend this model to include Script, the public key and signature in the input are replaced by an input script, called scriptSig. The address of the output is replaced by an output script, called scriptPubKey. To verify an input, we execute, starting with the empty stack, first the scriptSig followed by the scriptPubKey. The input is correct, if the execution is not aborted, at the end the stack is nonempty, and the top element of the stack is not false. To prevent an open OP_IF operation from the scriptSig to affect scriptPubKey, we will add our own separating instruction in between to make sure that the two parts are executed separately[3].

In order to carry out this verification we introduce an interpreter for Script in Agda. There are several groups of instructions. The first group of instructions are stack manipulating instructions. They assume a certain number of elements on the stack and replace them by different elements. For instance, OP_ADD assumes two elements on the stack, and replaces them by their sum. If there are not enough elements on the stack or certain conditions are not fulfilled the instructions abort (e.g., in case of OP_VERIFY if the top element is not true). Considering a possible failure all these instructions translate into functions of type

$$\text{Stack} \to \text{Maybe Stack}$$

The second group of instructions are those referring to the message representing the part of the instruction to be signed, or the time. They translate into functions

$$\text{Time} \to \text{Msg} \to \text{Stack} \to \text{Maybe Stack}$$

For dealing with conditionals, we need a second stack which gives information about the nesting of conditionals, and whether the current if- or else-case is to be executed. The elements are ifCase and elseCase, corresponding to the situation where we are in the if- or else-case of a conditional to be executed; ifSkip and elseSkip, where we are in an if- or else-case not to be executed; and ifIgnore, corresponding to the situation where we have a complete if-then-else which occurs inside an if- or else-case which is to be skipped. Note that we didn't introduce any jump instructions which would make verification more difficult. The operations correspond to functions of type

$$(\text{Stack} \times \text{IfStack}) \to \text{Maybe } (\text{Stack} \times \text{IfStack})$$

After lifting these different functions (using a higher order function these liftings can be done easily in a generic way), we obtain operations of type

$$\text{Time} \to \text{Msg} \to (\text{Stack} \times \text{IfStack}) \to \text{Maybe } (\text{Stack} \times \text{IfStack})$$

This lifting needs to take care of the fact that in case of ifSkip, elseSkip, ifIgnore on top of the IfStack, all non-conditional instructions need to be ignored.

As it stands the model is currently a prototype, since only a part of the language for smart contracts has been added. Once it is complete it needs to be thoroughly tested relative to Bitcoin Core. Ideally one would rewrite Bitcoin core in Agda, based e.g., on Haskoin Core [hac21]. The next step would be to use this approach to verify smart contracts, where the challenge is to specify what it means for a smart contract to be correct, which involves temporal features. Solving this challenge would allow to give more abstract specifications of smart contracts expressing directly its correctness rather than just looking for possible attacks or showing that a contract is bisimilar to an abstract smart contract assumed to be contract – these are the techniques commonly used for specifying the correctness of smart contracts. This would also allow to verify more generic forms of smart contracts, something which we don't assume is possible using the normally used automated theorem techniques. Another challenge is to expand the use of smart contracts to a language like the EVM or to directly add object-based programming to the language, which could make use of our work on integrating object-based programming into Agda. [AAS17, Set07].

[3] After suspecting a problem when developing the specification, we learned that there was indeed originally a problem in Bitcoin which was fixed in 2010, see Sect. 6 of [Ant17].

References

[AAS17] Andreas Abel, Stephan Adelsberger, and Anton Setzer. Interactive programming in Agda – Objects and graphical user interfaces. *Journal of Functional Programming*, 27, Jan 2017. https://doi.org/10.1017/S0956796816000319.

[Ant17] Andreas M Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly, 2nd edition, 2017.

[dS20] Guilherme Horte Alvares da Silva. *A simplified version of Bitcoin, implemented in Agda*. PhD thesis, Escola de Matemática Aplicada, FGV, Brasil, 2020. Available from http://bibliotecadigital.fgv.br/dspace/bitstream/handle/10438/29110/thesis.pdf?sequence=1&isAllowed=y.

[hac21] hackage.haskell.org. haskoin-core: Bitcoin & Bitcoin Cash library for Haskell. Vers 0.20.3, 17 May 2021. https://hackage.haskell.org/package/haskoin-core.

[Set07] Anton Setzer. Object-oriented programming in dependent type theory. In Henrik Nilsson, editor, *Trends in Functional Programming Volume 7*, pages 91 – 108, Bristol and Chicago, 2007. Intellect.

[Set18] Anton Setzer. Modelling Bitcoin in Agda. *arXiv*, arXiv:1804.06398:27, 17 April 2018. https://arxiv.org/abs/1804.06398.

[Set19] Anton Setzer. A model of the blockchain using induction-recursion. In Mark Bezem and Niels van der Weide, editors, *TYPES 2019. 25th International Conferences on Types of Proofs and Programs. Abstracts*, pages 102 – 103, Oslo, Norway, 2019. Centre for Advanced Study (CAS), The Norwegian Academy of Science and Letters. Available from https://eutypes.cs.ru.nl/eutypes_pmwiki/uploads/Main/books-of-abstracts-TYPES2019.pdf.