2022 IEEE 1st Global Emerging Technology Blockchain Forum: Blockchain & Beyond (iGETblockchain) | 978-1-6654-5198-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/iGETblockchain56591.2022.10087054

Verifying Correctness of Smart Contracts with Conditionals

1st Fahad Alhabardi
Department of Computer Science
Swansea University
Swansea, United Kingdom
fahadalhabardi@gmail.com

2nd Bogdan Lazar
School of Management
University of Bath
Bath, United Kingdom
lazarbogdan90@yahoo.com

3rd Anton Setzer
Department of Computer Science
Swansea University
Swansea, United Kingdom
a.g.setzer@swansea.ac.uk

***Abstract*—In this paper, we specify and verify the correctness of programs written in Bitcoin's smart contract SCRIPT in the interactive theorem prover Agda. As in the previous article [1], we use weakest preconditions of Hoare logic to specify the security property of access control, and show how to develop human-readable specifications. In this article, we include conditionals into the language. For the operational semantics we use an additional stack, the ifstack, to deal with nested conditionals. This avoids the addition of extra jump instructions, which are usually used for the operational semantics of conditionals in Forth-style stack languages. The ifstack preserves the original nesting of conditionals, and we determine an ifthenselse-theorem which allows to derive verification conditions of conditionals by referring to conditions for the if- and else-case.**

***Index Terms*—Bitcoin, Bitcoin script, smart contracts, security, cryptocurrency, blockchain control flow, Agda, operational semantics, Hoare logic, weakest precondition, access control**

I. Introduction

Blockchain is a decentralized, distributed ledger containing blocks of records linked by hashes. It operates through immutable peer-to-peer technology in a trustless environment. Besides creating a cryptocurrency, the blockchain allows the development of several use-cases due to the creation of smart contracts.

Satoshi Nakamoto [2] introduced Bitcoin in 2008 as a cryptocurrency that provides a private anonymous payment mechanism in a peer-to-peer network. Several cryptocurrencies have been introduced since [3], with Ethereum extending smart contracts to a Turing complete language.

Smart contracts [4] extend simple transactions of cryptocurrencies by allowing the addition of programs. Smart contracts can be defined as programs automatically executed when certain conditions are fulfilled on the blockchain. Various smart contract languages have been developed. For example, Bitcoin smart contracts are written in SCRIPT [5], a low-level language. In Ethereum Solidity [6] and Vyper [7] are two high level smart contract languages, which compile to the low level language of the Ethereum Virtual Machine. The smart contract language of Cardano is Plutus [8], a high-level language based on the functional language Haskell.

Smart contracts face several challenges, particularly security [9], because once a smart contracts has been published on the blockchain network, its code and published transactions referring to it are immutable. This means developers must ensure code security. If any errors are discovered, there is no direct way to correct the code, and cyber criminals may take advantage of these errors. As a result, ensuring that the smart contract is functioning properly before deploying it to the network is critical, as errors can be extremely costly. There are two formal ways to verify smart contract's correctness and security [10]: (1) using mathematical methods like theorem proving or (2) conducting test cases.

In this paper, we use Agda [11], an interactive theorem prover, which is both a functional programming language and a proof assistant. As a result, Agda enables us to build programs and reason about them within the same system. This decreases the risk of errors when converting programs from a programming language to a theorem prover and enables smart contracts to be executed directly in Agda.

Bitcoin Script is based on a Forth-style stack machine. Our operational semantics differs from the conventional method of dealing with such languages, which consists of first replacing conditionals in Forth-style programs by conditional and nonconditional jumps. Instead we use an ifstack to operate directly on the script. Therefore the structure of the script is preserved, and we can define theorems which from verification conditions for the if-case and else-case of a conditional derive the verification condition for the conditional.

The remaining part of this paper is structured as follows: Sect. II introduces related work. In Sect. III, we give an overview of the proof assistant Agda. Then, we introduce Bitcoin Script and define its operational semantics in Sect. IV. We present our definition of the Hoare logic in Sect. V. We then introduce in Sect. VI an ifthenelse-theorem and apply it to the verification of a conditional consisting of two P2PKH scripts. We finish with a conclusion in Sect. VII.

Git repository. This work has been formalized and full proofs have been carried out in the proof assistant Agda. The code can be found at [12].

II. Related Work

In this section, we will discuss the smart contract verification studies that are relevant to our approach. We begin by reviewing papers that discuss the verification of smart contracts using theorem provers. Then, we demonstrate various techniques used to verify smart contracts, such as symbolic execution and model checking. A more extended review of the literature can be found in our previous paper [1].

Using theorem provers to verify smart contracts. Several authors have discussed approaches related to verifying the correctness of smart contracts. Hirai [13] developed a formal EVM model in the Lem programming language [14] and used Isabelle/Higher-Order Logic (HOL) [15] as a theorem prover to verify EVM bytecode. He utilized this approach to prove Ethereum smart contracts' safety. Zheng et al. [16] developed FSPVM-E, a formal symbolic process virtual machine that verifies smart contracts' dependability, security, and function. FSPVM-E comprises a broad, extendable, and reusable formal memory framework; an extensible programming language called Lolisa which uses generalized algebraic data types; and a formally verified interpreter of Lolisa called FEther. The self-correctness of the components described before is certified through Coq [17]. FSPVM-E supports ERC20 and can symbolically run Ethereum-based smart contracts, scan their vulnerabilities, and validate their dependability and security using Hoare logic in Coq. Annenkov et al. [18] incorporated functional languages in Coq by employing meta-programming. After that, they developed the language's meta-theory with deep embedding and reasoning about concrete programs with shallow embedding. Then, they developed a fundamental smart contract language in Coq and validated a crowdfunding contract's characteristics. Lamela et al. [19] developed the domain-specific language Marlowe for financial contracts. This language was developed on the Cardano blockchain. Marlowe was utilized to ensure that any smart contracts created in this language would conserve funds. This means that except for an error, the money that comes in plus the contract money before the transaction should be equal to the money that comes out plus the contract after the transaction. Using the Isabelle theorem prover, the Marlowe system has been formally proven, along with features such as money conservation. Sun et al. [20] presented formal verification approaches for five types of smart contract security issues in Ethereum, namely integer overflow, the function specification issue, the invariant issue, the authority control issue, and the behavior of the specific function. They also verified the Binance Coin (BNB) contract. They used the Coq proof assistant to verify and formalise their proofs.

Using symbolic execution to verify and analyze smart contracts. There are many efforts to analyze and expose security vulnerabilities in smart contracts using symbolic execution. Tikhomirov et al. [21] introduced SmartCheck, a static analysis tool that can be expanded and used to discover Solidity contract vulnerabilities. SmartCheck takes Solidity source code, turns it into an intermediate form based on Extensible Markup Language, and compares this form to XPath patterns. This tool can find certain security holes like the Denial of Service (DoS). SmartCheck was written with Java. Beukema [22] attempted to establish a formal Bitcoin specification. Bitcoin's interface functions and the expected outputs were specified in his research. The majority of these functions outline how the Bitcoin network protocol should work. He used mCRL2, a programming language for specification. In his contribution, he verified some properties like double-spending.

Using model checking to verify smart contracts. Grishchenko et al. [23] presented a full small-step semantics of EVM bytecode and formalised a substantial part of it in the F*. This gave them executable code that they were able to check against the official Ethereum test suite. Also, they formally defined some critical security features for smart contracts. See our article [1] for a review of several other article which use model checking for verification of smart contracts.

III. INTRODUCTION TO THE PROOF ASSISTANT AGDA

Agda [11], is a dependently typed functional programming language that expands the Martin-Löf constructive type theory [24]. The most recent version of Agda is Agda 2, the version designed and introduced by Ulf Norell in his doctoral thesis in 2007 [25], and then further developed by a collection of people known as the Agda development team [11]. The Integrated Development Environment (IDE) for editing Agda programs is based on Emacs, mostly used for interactive editing and verifying proofs [26]. Agda's features include inductive and inductive-recursive data types, coinduction, pattern matching with dependent patterns, mixfix operators, copattern and a module system [27]. Agda has complete support for Unicode which supports mathematical and multifix symbols in order to be able to write Agda code that is close to standard mathematical notations. Moreover, identifiers, keywords, and a type-directed development environment is provided through Agda [28]. Agda has coverage and termination checkers [27], and these concepts are required for Agda to be a consistent proof assistant. In Agda, program code can be written gradually, meaning some parts of the program can remain unfinished, and programmers are able to get helpful information from Agda on filling the parts of the code left open step by step, supported by the type checking tool. The type checker is able to detect incorrect proofs by detecting type errors. In addition, it displays the current goals with type information and the environment information associated with those goals. The coverage checker of Agda checks that the initial code of a defined function includes all possible existing cases in a particular program. Agda has many type levels, the lowest one is called Set for historical reasons. Record types serve as the newer approach to define coinductive types. Agda defines inductive data types as dependent versions of algebraic types using the keyword "data" as common in functional programming, with strict positivity of the constructors required. Agda is similar to Coq [17], a language that extends the Calculus of Constructions, but is impredicative, whereas Agda is predicative. Coq is especially good for writing formal specifications and proofs. However, there are some key distinctions between Agda and Coq that might suggest the wider applicability of Agda. For example, Agda supports inductive-recursive types, while Coq does not [1]. Agda also has a more flexible pattern matching system than Coq and supports copattern matching.

As an example, we define the inductive type of InstructionAll in Agda as follows:

```
data InstructionAll : Set where
  opEqual opAdd opSub : InstructionAll
  opVerify opCheckSig   : InstructionAll
```

The definition above includes a new type called InstructionAll with 16 constructors, opEqual, opAdd, opSub . . . of which we only show the first 5. The elements of (InstructionAll) are used in order to develop Bitcoin programs in Agda.

It is possible to write a function in Agda that returns a type:

```
BitcoinScript : Set
BitcoinScript = List InstructionAll
```

BitcoinScript defines the type of a Set as a list of instructions of type InstructionAll.

IV. Operational Semantics for Bitcoin Script

In this section, we introduce Bitcoin script, and define its operational semantics.

A. *Bitcoin Script - the language of Bitcoin for Smart Contracts*

The Bitcoin smart contract language SCRIPT is stack-based, similar to Forth [5]. SCRIPT is not Turing-complete and execution of a Bitcoin scripts either fails or terminates. Bitcoin script does not include loops, jumps or similar control structures [29]. In Bitcoin 0 denotes false and nonzero values denote true. Every transaction input and output in Bitcoin is associated with a script. A transaction input is valid w.r.t. a corresponding transaction output, if the execution of the output script starting with an empty stack terminates, and then the execution of the input script on the resulting stack terminates with a non-empty stack having a non-zero top element. SCRIPT is a sequence of instructions called Op_Codes which are encoded as hexadecimal numbers.

In the previous paper [1], we introduced and explained the meaning of selected non-conditional instructions. We illustrate the execution of a non-conditional Bitcoin script by the following simple example: <2> <3> OP_ADD <5> OP_EQUAL The stack evolves as follows:

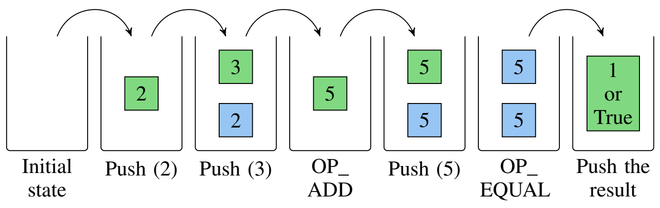

Fig. 1. Simple example of local instructions

In this example we start with an empty stack. After pushing 2, 3 on the stack (instructions <2> <3>), OP_ADD adds the two top elements together. After pushing 5 on the stack OP_EQUAL checks whether the two top elements are equal, and returns in this case 1 for true.

Control flow operations are executed as follows:

- If the value at the top of the stack is non-zero, after an OP_IF the set of consecutive opcodes until the next matching OP_ELSE or OP_ENDIF will be executed; in case this is an OP_ELSE, all the following instructions until the next matching OP_ENDIF will be ignored.
- In case the top element is 0, all instructions until the next matching OP_ELSE or OP_ENDIF will be ignored; in case this is an OP_ELSE, all the following instructions until the next matching OP_ENDIF will be executed.
- In case of nested if then else, the complete conditional from OP_IF to OP_ENDIF is either executed or ignored depending on whether it occurred within an if-case or else-case to be executed.
- OP_NOTIF behaves in the same as OP_IF but executing the if-case in case of top element 0 and the else-case in case of top element not 0.

Consider the following example:
OP_IF <Alice's PubKey> OP_CHECKSIG
OP_ELSE <Bob's PubKey> OP_CHECKSIG OP_ENDIF
Assume the stack contains $[\ 1,\ \mathrm{sig}\ ]$. Then the if-case will be executed, pushing Alice's public key on the stack. The script succeeds, if sig is a signature for the transaction using Alice's private key. If the stack contained $[\ 0,\ \mathrm{sig}\ ]$, the same would be done using Bob's public key.

There are multiple standard scripts [30] used in Bitcoin, for instance, the Pay-to-Public-Key-Hash (P2PKH) and Multi-signature (P2MS) scripts.

B. *Operational Semantics*

This subsection defines the operational semantics of Bitcoin SCRIPT in detail. The semantics is implemented in Agda. It needs to be checked (validated) carefully to ensure that there are no translation errors.

We include control flow statements of Bitcoin SCRIPT, which allows to formalise more complex smart contracts, but have non-local behavior. In our previous article [1], we provided the operational semantics of local instructions, such as OP_DUP, OP_ADD. We included as well instructions with a more complex behavior, such as a multi-signature instruction and a time delay instruction. We showed how to derive and verify these scripts using weakest preconditions for Hoare triples. All opcodes may fail if the stack has insufficient elements to complete the operation. The operational semantics in our previous article [1] was given w.r.t. a state, consisting of a standard stack (Stack), which is given as a list of natural numbers, a message (Msg) corresponding to the transaction that has to be signed (we defined Msg as a data type in Agda), and the current time as represented as an element of Time. The resulting definition is

$$\mathsf{StackState} := \mathsf{Time} \times \mathsf{Msg} \times \mathsf{Stack}$$

Time is referred for instance by the instruction OP_CHECKLOGTIMEVERIFY, and Msg is referred by the instructions which check correctness of signatures.

In order to deal with conditionals, we extend the state of the previous article by adding an additional stack (IfStack) to deal with possibly nested conditionals. Therefore the state which allows to deal with control flow statements is as follows:

$$\mathsf{State} := \mathsf{Time} \times \mathsf{Msg} \times \mathsf{Stack} \ \times \mathsf{IfStack}$$

Here IfStack is a list of elements from the following list: (ifCase, elseCase, ifSkip, elseSkip, ifIgnore).

- An empty IfStack means that we are currently not within any conditional,
- a top element ifCase means that we are in the if-case of a conditional to be executed,
- top element elseCase means that we are in the else-case to be executed,
- ifSkip means that we are in the if-case of a conditional not to be executed where the else-case is to be executed,
- elseSkip means that we are in the else-case of a conditional not to be executed,
- ifIgnore means that we are in the if-case of a conditional, where the whole conditional is to be ignored because it is nested within an if or else-case of a conditional to be ignored.
- There is no need for an elseIgnore, since we can reuse elseSkip for it.

If the IfStack is created using the above semantics starting with the empty stack, we see that ifCase, elseCase, ifSkip can only occur above an empty ifstack, or ifstack with top element in $\{\mathsf{ifCase}, \mathsf{elseCase}\}$, and ifIgnore can only occur above an ifstack with top element in $\{\mathsf{ifIgnore}, \mathsf{ifSkip}, \mathsf{elseSkip}\}$. We add to the IfStack the consistency condition that this condition is fulfilled. In the actual Agda code we have instead of a consistent ifstack, two components, an ifstack, and condition requiring the ifstack to be consistent. The consistency condition avoids having to prove, when verifying Bitcoin scripts, verification conditions for ifstacks which never occur.

The type for all opcodes is given as an element of the Agda data type InstructionAll. Accordingly, the operational semantics of an instruction op : InstructionAll is represented as

$$[\![\ op\ ]\!]\mathsf{s} : \mathsf{InstructionAll} \rightarrow \mathsf{State} \rightarrow \mathsf{Maybe\ State}$$

We will give the definition of $[\![\ op\ ]\!]\mathsf{s}$ for conditional instructions. The definition of $[\![\ \mathsf{opIf}\ ]\!]\mathsf{s}$ is as follows:

- If the top element of IfStack is ifSkip, elseSkip, or ifIgnore, then the conditional starting with the IF_CASE needs to be ignored. This is achieved by pushing an additional ifIgnore onto the IfStack.
- Otherwise, if the stack is empty, the execution will fail.
- Otherwise, the IfStack is empty, or the top element of it is ifCase or elseCase. Then if the top element of the stack is
  - 0 then ifSkip will be pushed onto IfStack, since the if-case is to be ignored and the else-case to be executed,
  - is not 0 then ifCase will be pushed on the IfStack, since the if-case is to be executed.

Now we define $[\![\ \mathsf{opElse}\ ]\!]\mathsf{s}$:

- If the IfStack is empty, then there is no OP_IF matching the OP_ELSE, and therefore the execution fails.
- Otherwise, if the top element of IfStack is:
  - elseSkip or elseCase then there was already an OP_ELSE matching the previous OP_IF, and the current OP_ELSE is unmatched, therefore execution of the script fails;
  - ifSkip then the top element will be replaced with elseCase.
  - ifCase or ifIgnore then the top element will be replaced with elseSkip.

Finally, we define $[\![\ \mathsf{opEndIf}\ ]\!]\mathsf{s}$:

- If the IfStack is empty then the OP_ENDIF is unmatched, so the operation fails.
- Otherwise the OP_ENDIF terminates the current conditional, and we pop the top element from the IfStack.

For all local instructions,

- if the IfStack is empty or its top element is ifCase or elseCase then the instruction is executed (as defined in our previous paper [1]) on all components excluding the IfStack, while the IfStack remains unchanged;
- otherwise the State remains unchanged.

V. Hoare Logic

As explained in our previous paper [1], one can specify the security of Bitcoin script by weakest preconditions, where the post condition is the accept condition. Both pre- and post conditions are predicates on State. Since scripts might fail, the operational semantics of a program applied to a state might fail, and is an element of a Maybe Type, which adds an error element nothing to the elements of the underlying type. We therefore lift the post condition ψ to a predicate ψ^{+} on Maybe State, where ψ^{+} is false for the error element nothing, and otherwise returns ψ applied to the underlying element of State.

Hoare triples consists of a precondition, a program, and a postcondition. This triple asserts that if a precondition is met before the execution of a program, then the postcondition will be true after the program has been executed. Hoare triples can be defined as follows:

$$<\varphi> p <\psi> := \forall s \in \mathsf{State}.\varphi(s) \rightarrow (\psi^{\,+})\ ([\![\ \mathsf{p}\ ]\!]\ s)$$

As discussed in detail in [1] weakest precondition formalizes access control to the resource guarded by a Bitcoin script. It expresses that the precondition is not only sufficient but also necessary in order for the postcondition to hold once the program has been executed. It can be defined as follows:

$$<\varphi>^{\leftrightarrow}\ p <\psi> := \forall s \in \mathsf{State}.\varphi(s) \leftrightarrow (\psi^{\,+})\ ([\![\ \mathsf{p}\ ]\!]\ s)$$

In order to unlock a locking script, it is necessary to provide an unlocking script, which computes a state which fulfills the weakest precondition for the locking script w.r.t. the accept condition, therefore this weakest precondition specifies access control for the unlocking script. A more detailed discussion of using weakest preconditions for access control can be found in [1].

VI. Verification of Conditionals

In our previous paper [1] we developed techniques for determining and, proving weakest preconditions for scripts not involving conditionals. Conditionals, as discussed in this paper, allow to define more complex scripts which allow the unlocking of scripts depending on different scenarios. In order to verify scripts using conditionals we develop ifthenelse-theorems which form the weakest preconditions for the ifProg and the $\mathrm{elseProg}$ of a conditional derive the weakest preconditions for the conditional clause.

In our setting, when writing a script as
OP_IF ifProg OP_ELSE $\mathrm{elseProg}$ OP_ENDIF
we don't require the OP_ELSE and OP_ENDIF to match the OP_IF - there could be some other OP_ELSE or OP_ENDIF occurring in ifProg or $\mathrm{elseProg}$ matching the OP_IF. The script might still be correct because of the occurrence of another OP_IF. The reason for not requiring parsed programs is that it allows us to keep the data structure for scripts as a simple list of instructions and mirrors as well the real situation where there is no requirement that scripts submitted to Bitcoin are parsed correctly. This is different from normal program verification, where one has control over programs and requires them to be parsed correctly. Instead of requiring correctly parsed scripts we will add additional conditions in the ifthenelse-theorem to make sure that if the condition of the OP_IF is true, the $\mathrm{elseProg}$ has no effect, and if it is false, the ifProg has no effect. This will be in addition to the two expected conditions, one for the ifProg in case the top element of the stack is true and one for the $\mathrm{elseProg}$ in case the top element of the stack is false. The condition for $\mathrm{elseProg}$ requires as well some extra cases: when working backwards from the post condition to obtain the weakest precondition, we need to deal with the situation that before the OP_ENDIF the top element of the ifstack could have been any element except (because of the consistency condition) ifIgnore. So we need to have conditions for all these elements of $\mathrm{elseProg}$ even though, while working further backwards we have reached the OP_ELSE, it follows that the element must have been elseCase or elseSkip.

We first define some notations used and then introduce the main ifthenelse-theorem.

Definition 1:

(a) Let for a predicate ϕ on StackState the predicate $\mathrm{lift}(\phi)$ on IfStack be its lifting ignoring the ifstack component.
(b) Let ∧p and ∨p be the conjunction and disjunction of two predicates on State.
(c) Let ϕ be a predicate on StackState. Then $\mathrm{truePr}(\phi)$ is the predicate on State expressing that the stack has top element > 0 (i.e. not false), and ϕ holds for the remaining stack, the message to be signed, and the time.
Let $\mathrm{falsePr}(\phi)$ be the same predicate, but assuming the top element is $= 0$ (i.e. false).

Theorem 2 (Main ifthenelse-theorem): Let ϕ_{true}, ϕ_{false}, ψ be predicates on StackState and ifProg, $\mathrm{elseProg}$ two Bitcoin scripts. Let i : IfStack, which is either empty or has top element in s{ifCase, elseCase}.
Assume the following conditions:

(1) <$\mathrm{lift}(\phi_{\mathrm{true}})$ ∧p ifStack =cons(ifCase, i)>$^{\leftrightarrow}$
ifProg <$\mathrm{lift}(\psi)$ ∧p ifStack =cons(ifCase, i)>
(2) ifStack =$\mathrm{lift}(\phi_{\mathrm{false}})$ ∧p ifStack =cons(ifSkip, i)
ifProg <$\mathrm{lift}(\phi_{\mathrm{false}})$ ∧p ifStack =cons(ifSkip, i)>
(3) $\forall x \in$ {ifCase, elseCase}.
<$\mathrm{lift}(\phi_{\mathrm{false}})$ ∧p ifStack =cons(x, i)>$^{\leftrightarrow}$
elseProg <$\mathrm{lift}(\psi)$ ∧p ifStack =cons(x, i)>
(4) $\forall x \in$ {ifSkip, elseSkip}.
<$\mathrm{lift}(\psi)$ ∧p ifStack =cons(x, i)>$^{\leftrightarrow}$
elseProg <$\mathrm{lift}(\psi)$ ∧p ifStack =cons(x, i)>

Then we get
<($\mathrm{truePr}(\phi_{\mathrm{true}})$ ∨p $\mathrm{falsePr}(\phi_{\mathrm{false}})$) ∧p ifStack =$i$>$^{\leftrightarrow}$
[opIf] ++ ifProg ++ [opElse] ++ elseProg ++ [opEndIf]
<$\mathrm{lift}(\psi)$ ∧p ifStack =i>

In order to prove the conditions (2) and (4) for scripts where the ifProg or $\mathrm{elseProg}$ have no occurrence of conditional instructions, we use the following theorem:

Theorem 3: Let ϕ be a predicate on StackState, $x \in$ {ifSkip, elseSkip, ifIgnore, }, i : IfStack, and p be a Bitcoin script not containg conditional instructions. Then we have
<$\mathrm{lift}(\phi)$ ∧p ifStack =cons(x, i)>$^{\leftrightarrow}$
p <$\mathrm{lift}(\phi)$ ∧p ifStack =cons(x, i)>

Using these two theorems we can prove as an example the weakest precondition for a simple conditional:

- Let $\mathrm{P2PKHscript}(pbkh)$ be the P2PKH Bitcoin script as defined in [1] which checks that the stack has size at least two, the top element of the stack is pkh hashing to $pbkh$ and the next element is a signature sig for the message corresponding to pbk.
- Let $\mathrm{P2PKHc}(pbkh)$ be the weakest precondition for $\mathrm{P2PKHscript}(pbkh)$, which expresses that the stack is indeed as described before.
- Let accept be the accept condition on StackState, stating that the stack has size at least 1, and top element which is > 0 (i.e. not false).
- Let
$\mathrm{P2PKHCondScr}$:= OP_IF $\mathrm{P2PKHscript}(pbkh_1)$
OP_ELSE $\mathrm{P2PKHscript}(pbkh_2)$
OP_ENDIF
be a conditional P2PKH script, which operates like a P2PKH script but allowing two different public key hashes $pbkh_1$ and $pbkh_2$ and requiring an extra element on the stack which considered as a Boolean decides which of the two public key hashes is to be used.

The theorem expresses that the weakest precondition for the accept condition for p is that the top element of the stack is > 0 and the remaining stack fulfills the weakest precondition for P2PKH w.r.t. $pbkh_1$ or the top element is 0 and we have the weakest precondition for P2PKH w.r.t. $pbkh_2$, and the ifstack is empty:

Theorem 4:

$$<\!(\mathrm{truePr}(\mathrm{P2PKHc}(pbkh_1))\ \vee\mathsf{p}\ \mathrm{falsePr}(\mathrm{P2PKHc}(pbkh_2))) \\ \wedge\mathsf{p}\ \mathrm{ifStack} = [\,]\!>^{\leftrightarrow} \\ \mathrm{P2PKHCondScr}\ \ <\!\mathrm{lift}(\mathrm{accept})\ \wedge\mathsf{p}\ \mathrm{ifStack} = [\,]\!>$$

The proof is by Theorem 2, where the proof conditions (1) and (3) follow by the verification conditions for the P2PKH script lifted to having an ifstack, and conditions (2) and (4) follow by Theorem 3.

VII. Conclusion and Future Work

In this paper, we used the Agda proof assistant in order to verify Bitcoin scripts. The paper deals with non-local instructions such as OP_IF, OP_ELSE, and OP_ENDIF. We formalise these non-local instructions' operational semantics to re-create the process of smart contract validation. We extended the state from our previous article [1] by adding an additional ifstack, and defined the operational semantics of conditionals. We developed an ifthenelse-theorem and used it to verify an example script.

In future work, we will apply those non-local instructions to more complex scripts such as the Pay-to-Public-Key-Hash (P2PKH) and Pay to Multisig (P2MS) scripts. In [1] we demonstrated two approaches for obtaining weakest preconditions for scripts not containing conditionals: (1) a step-by-step method going backwards through a script, instruction by instruction, possibly combining some of them into one step, or (2) a symbolic execution of the code translated into a nested case distinction. Using the ifthenelse-theorem of this paper, we can then derive from weakest preconditions for each of the pathways (obtained by these 2 techniques) a weakest precondition for the full script.

References

[1] F. F. Alhabardi, A. Beckmann, B. Lazar, and A. 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)*, ser. LIPIcs, vol. 239. Dagstuhl, Germany: Leibniz-Zentrum für Informatik, 2022, pp. 1:1–1:25, doi: https://doi.org/10.4230/LIPIcs.TYPES.2021.1.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, Availabe from https://www.debr.io/article/21260.pdf.

[3] D. Vujičić, D. Jagodić, and S. Ranđić, "Blockchain technology, Bitcoin, and Ethereum: A brief overview," in *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, 2018, pp. 1–6, doi: http://dx.doi.org/10.1109/INFOTEH.2018.8345547.

[4] N. Szabo, "Smart Contracts: Building Blocks for Digital Markets," *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, no. 2, p. 28, 1996.

[5] A. M. Antonopoulos, *Mastering Bitcoin: Programming the open blockchain*. " Second ed. O'Reilly Media, Inc.", 2017.

[6] Ethereum Community, "Solidity documentation," Retrieved 15 August 2022, Availabe from https://docs.soliditylang.org/en/v0.8.16/.

[7] Vyper Team, "Vyper documentation," Retrieved 15 August 2022, Availabe from https://vyper.readthedocs.io/en/stable/.

[8] IOHK, "Plutus: Programming Languages," Retrieved 15 August 2022, Availabe from https://testnets.cardano.org/en/programming-languages/plutus/overview/.

[9] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in *Principles of Security and Trust*. Berlin, Heidelberg: Springer, 2017, pp. 164–186, doi: https://doi.org/10.1007/978-3-662-54455-6_8.

[10] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "Verification of smart contracts: A survey," *Pervasive and Mobile Computing*, vol. 67, pp. 1–19, 2020, doi: http://dx.doi.org/10.1016/j.pmcj.2020.101227.

[11] Agda Community, "Welcome to Agda's documentation!" Retrieved 19 July 2022, Availabe from https://agda.readthedocs.io/en/v2.6.2/.

[12] A. Setzer, F. Alhabardi, and B. Lazar, "Verifying correctness of smart contracts with conditionals," 2022, Available from https://github.com/fahad1985lab/Verifying--Correctness--of-Smart--Contracts--with--Conditionals.

[13] H. Yoichi, "Defining the Ethereum Virtual Machine for Interactive Theorem Provers," in *Financial Cryptography and Data Security*. Cham: Springer, 2017, pp. 520–535, doi: https://doi.org/10.1007/978-3-319-70278-0_33.

[14] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: Reusable engineering of real-world semantics," *ACM SIGPLAN Notices*, vol. 49, no. 9, p. 175–188, Aug 2014, doi: https://doi.org/10.1145/2692915.2628143.

[15] Isabelle Community, "Isabelle documentation," Retrieved 19 July 2022, Availabe from https://isabelle.in.tum.de/documentation.html.

[16] Yang, Zheng and Lei, Hang and Qian, Weizhong, "A Hybrid Formal Verification System in Coq for Ensuring the Reliability and Security of Ethereum-Based Service Smart Contracts," *IEEE Access*, vol. 8, pp. 21 411–21 436, 2020, doi: https://doi.org/10.1109/ACCESS.2020.2969437.

[17] Coq Community, "The Coq Proof Assistants," Retrieved 15 July 2022, Availabe from https://coq.inria.fr/.

[18] D. Annenkov, J. B. Nielsen, and B. Spitters, "ConCert: A Smart Contract Certification Framework in Coq," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 215–228, doi: https://doi.org/10.1145/3372885.3373829.

[19] P. Lamela Seijas, A. Nemish, D. Smith, and S. Thompson, "Marlowe: Implementing and Analysing Financial Contracts on Blockchain," in *Financial Cryptography and Data Security*. Cham: Springer, 2020, pp. 496–511, doi: https://doi.org/10.1007/978-3-030-54455-3_35.

[20] T. Sun and W. Yu, "A Formal Verification Framework for Security Issues of Blockchain Smart Contracts," *Electronics*, vol. 9, no. 2, 2020, doi: https://doi.org/10.3390/electronics9020255.

[21] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static Analysis of Ethereum Smart Contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 9–16, doi: https://doi.org/10.1145/3194113.3194115.

[22] Beukema, W.J.B, "Formalising the bitcoin protocol," in *21th Twente Student Conference on IT (2014)*, 2014, Availabe from https://fmt.ewi.utwente.nl/media/120.pdf.

[23] I. Grishchenko, M. Maffei, and C. Schneidewind, "A Semantic Framework for the Security Analysis of Ethereum Smart Contracts," in *Principles of Security and Trust*. Cham: Springer, 2018, pp. 243–269, doi: https://doi.org/10.1007/978-3-319-89722-6_10.

[24] P. Martin-Löf, "An Intuitionistic Theory of Types: Predicative Part," in *Logic Colloquium '73*. Elsevier, 1975, vol. 80, pp. 73–118, doi: http://dx.doi.org/10.1016/S0049-237X(08)71945-1.

[25] U. Norell, "Towards a practical programming language based on dependent type theory," PhD thesis, Department of Computer Science and Engineering, Chalmers, Göteborg, Sweden, September 2007, Availabe from http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf.

[26] A. Bove, L. S. Barbosa, A. Pardo, and J. S. Pinto, *Language Engineering and Rigorous Software Development*. Springer Science & Business Media, 2009, vol. 5520, ISBN: 978-3-642-03152-6.

[27] N. A. Danielsson and U. Norell, "Parsing Mixfix Operators," in *Implementation and Application of Functional Languages*. Berlin, Heidelberg: Springer, 2011, pp. 80–99, doi: https://doi.org/10.1007/978-3-642-24452-0_5.

[28] T. Altenkirch, J. Chapman, and T. Uustalu, "Relative monads formalised," *Journal of Formalized Reasoning*, vol. 7, no. 1, p. 1–43, Jan. 2014, doi: http://dx.doi.org/10.6092/issn.1972-5787/4389.

[29] H. Brakmić, *Bitcoin Script*. Berkeley, CA: Apress, 2019, pp. 201–224, doi: http://dx.doi.org/10.1007/978-1-4842-5522-3_7.

[30] S. Bistarelli, I. Mercanti, and F. Santini, "An Analysis of Non-standard Bitcoin Transactions," in *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, 2018, pp. 93–96, doi: http://dx.doi.org/10.1109/CVCBT.2018.00016.