Verification of Bitcoin’s Smart Contracts in Agda using Weakest Preconditions for Access Control

Fahad F. Alhabardi ♦
Swansea University, Dept. of Computer Science

Arnold Beckmann ♦♣
Swansea University, Dept. of Computer Science

Bogdan Lazar ♦
University of Bath

Anton Setzer ♦♣
Swansea University, Dept. of Computer Science

Abstract

This paper addresses the verification of Bitcoin smart contracts using the interactive theorem prover Agda. It focuses on two standard smart contracts that govern the distribution of Bitcoins, Pay to Public Key Hash (P2PKH) and Pay to Multisig (P2MS). Both are written in Bitcoin’s low-level language script, and provide the security property of access control to the distribution of Bitcoins.

The paper introduces an operational semantics of the script commands used in P2PKH and P2MS, and formalises it in the Agda proof assistant using Hoare triples. It advocates weakest preconditions in the context of Hoare triples as the appropriate notion for verifying access control. Two methodologies for obtaining human-readable weakest preconditions are discussed in order to close the validation gap between user requirements and formal specification of smart contracts: (1) a step-by-step approach, which works backwards instruction by instruction through a script, sometimes stepping over several instructions in one go; (2) symbolic execution of the code and translation into a nested case distinction, which allows to read off weakest preconditions as the disjunction of accepting paths. A syntax for equational reasoning with Hoare Triples is defined in order to formalise those approaches in Agda.

Keywords and phrases Blockchain; Cryptocurrency; Bitcoin; Ethereum; Agda; Coq; Verification; Hoare logic; Script; P2PKH; Multi signature; Security; Weakest precondition; Correctness; Proof; Smart Contract

1 Introduction

Bitcoin has been introduced in 2008 by Satoshi Nakamoto [26] as a cryptocurrency that provides a private anonymous payment mechanism in a peer-to-peer network [17]. Bitcoin also introduced blockchain as the underlying technology realising cryptocurrencies. Several cryptocurrencies have been introduced since then.

With blockchain also smart contracts have been introduced [33, 23]. Smart contracts can be defined as programs that are executed automatically when certain conditions are fulfilled. They were introduced as distributed applications first on Ethereum blockchain. Smart contracts can be written using a high-level language such as Solidity or Vyper on Ethereum [18], however they are executed in low-level languages like Script [10, 5] in Bitcoin or EVM in Ethereum [11].

As smart contracts can control real world values and are immutable once deployed on the blockchain network, a method to demonstrate their security and correctness is needed [23, 35]. According to [2, 25, 15], there are two ways to verify their accuracy: (1) using mathematical methods like formal verification, which utilize theorem proving, and (2) testing. Theorem proving provides an extremely flexible verification method that can be applied to various
types of systems including smart contracts. It can be done in interactive, automated, or hybrid mode.

We are using the interactive theorem prover Agda for the verification of smart contracts. Using Agda allows us to utilise functional programming and dependent type theory, thus defining smart contracts and reasoning about them in the same system. Another advantage of Agda is to have proofs that are check-able by hand. Other frameworks, such as Coq [28], use automatic proof search tools which do not automatically provide proof certificates which could in principle be checked by hand. The latter is a property which is desired in case of smart contracts.

The paper introduces an operational semantics of the script commands used in Pay to Public Key Hash (P2PKH) and Pay to Multisig (P2MS), two standard smart contracts that govern the distribution of Bitcoins. We formalise their operational semantics in Agda using Hoare triples.

Our verification focuses on the security property of access control. Access control is the restriction to access for a resource, which in our use case is access to cryptocurrencies like Bitcoin. We advocate that, in the context of Hoare triples, weakest preconditions are the appropriate notion to model access control: A (general) precondition expresses that when it is satisfied, access is granted, but there may be other ways to gain access without satisfying the precondition. The weakest precondition expresses that access is granted if and only if the condition is satisfied.

Weakest precondition can always be defined in a direct way, which we will explain in detail in Sect. 4.1. However, those are usually meaningless to humans who want to convince themselves that the smart contract is behaving as expected. Thus, the challenge is to obtain simple, human-readable descriptions of the weakest precondition of a smart contract. This would allow to close the validation gap between user requirements and formal specification of smart contracts.

In this paper, we discuss two methods for obtaining readable weakest preconditions: The first is obtained by working through the program backwards instruction by instruction. In some cases it is easier to group several instructions together and deal with them differently, as we will demonstrate with an example in Sect. 6.3. The second method evaluates the program in a symbolic way, and translates it into a nested case distinction. The case distinctions are made on variables (of type nat or stack) or on expressions formed from variables by applying basic functions to them such as hashing or checking for signature. From the resulting decision tree, the weakest precondition can be read off as the disjunction of the conjunctions of the conditions that occur along branches that lead to a successful outcome.

For both methods, it is necessary to prove that the established weakest precondition is indeed the weakest precondition for the program under consideration. For the first method, this follows by step wise operation. The second uses a proof that the original program is equivalent to the transformed program from which the weakest precondition has been established, or a direct proof which follows the case distinctions used in the symbolic evaluation.

We demonstrate the feasibility of our approaches by carrying them out in Agda for concrete smart contract, including P2PKH and P2MS.

Our approach also provides opportunities for further application: The usage of the weakest precondition with explicit proofs can be seen as a method of building verified smart contracts that are correct by construction. Instead of constructing a program and then evaluating it, the designer can start with the intended weakest precondition and postcondition, add some intermediate conditions, and then develop the program between those conditions. Such an
approach would extend the SPARK Ada framework [1] of using Hoare logic without the weakest precondition to check programs.

The remainder of this paper is organized as follows: In Sect. 2, we introduce related work on verification of smart contracts. Sect. 3 introduces Bitcoin script and defines its operational semantics. In Sect. 4, we introduce Hoare logic and weakest preconditions for security; we also define a syntax for equational reasoning with Hoare Triples to formalise those approaches in Agda. Sect. 5 introduces our first, step by step method of developing human-readable weakest preconditions and proving correctness of the Pay to Public Key Hash script (P2PKH). In Sect. 6, we introduce our second method based on symbolic execution and apply it to various examples. We conclude in Sect. 7.

Notations and git repository. The formulas can be presented as full Agda code, but often the formulas can also be presented in mathematical style. In order to switch between Agda code and mathematical code easy, we use the functional style for application (i.e. writing $f(a, b, c)$ instead of $f abc$) and $x : A$ instead of $x P A$. $s :: l$ denotes the cons operation on lists. The original Agda definitions are also available [32]. Display style Agda code presented in this paper has been automatically extracted from the Agda code.

2 Related Work

In this section we describe research addressing the verification of smart contracts.

A number of authors have addressed the verification of Ethereum smart contracts. Hirai [19] uses the Isabelle/HOL theorem prover to validate Ethereum Virtual Machine (EVM) bytecode by developing a formal model for EVM using the Lem language. They use this model to prove invariants and safety properties of Ethereum smart contracts. Amani et al. [3] extend Hirai’s EVM formalisation in Isabelle/HOL by a sound program logic at the level of bytecode. To this end, they structure bytecode sequences into blocks of straight-line code and create a program logic to reason about these. Ribeiro et al. [30] develop an imperative language and respective semantics system for a relevant subset of Solidity in the context of Ethereum, together with their formalisation in Isabelle/HOL, extending existing work. Their formalisation of semantics is based on Hoare logic and the weakest-precondition calculus. Their main contributions are proofs of soundness and relative completeness, as well as applications of their machinery to verify some smart contracts including modelling of smart contract vulnerabilities. Bhargavan et al. [7] provide formalisations of EVM bytecode in F*, a functional programming language designed for program verification. They define a smart contract verification architecture that can compile Solidity contracts, and decompile EVM bytecode, into F* using their shallow embedding, to express and analyse smart contracts.

Verification of Bitcoin’s smart contracts has also been addressed. Klomp et al. [22] propose a symbolic verification theory and tool to analyse and validate Bitcoin scripts, with a particular focus on characterising the conditions under which an output script, which controls the successful transfer of Bitcoins, will succeed. Bartoletti et al. [6] present BitML, a high-level domain-specific language for designing smart contracts in Bitcoin. They provide a compiler to convert smart contracts into Bitcoin transactions, and prove the correctness of their compiler wrt. a symbolic model for BitML and a computational model for Bitcoin which they define as well. Setzer [31] develops models of the Bitcoin blockchain in the interactive theorem prover Agda. This work focuses on the formalisation of basic primitives in Agda as a basis for future work on verifying the protocols of cryptocurrencies and developing verified smart contracts.
A number of papers discuss tools for analysing and verifying smart contracts that utilise model checking. Kalra et al. [21] develop a framework called ZEUS whose aim is to support automatic formal verification of smart contracts using abstract interpretation and symbolic model checking. ZEUS starts from a high-level smart contract, and employs user assistance for capturing correctness and fairness requirements. The contract and policy specification are then transformed into an intermediate language with well defined execution semantics. ZEUS then performs static analysis on the intermediate level and uses external SMT solvers to evaluate any verification properties discovered. A main focus of the work is on efficiency to reduce the state explosion problem inherent in any model checking approach. Park et al. [27] propose a formal verification tool for EVM bytecode based on KEVM, a complete formal semantics of EVM bytecode developed in the K-framework. To address performance challenges, they define EVM-specific abstractions and lemmas, which they then utilise to verify a number of concrete smart contracts. Mavridou et al. [24] introduce the VeriSolid framework to support the verification of Ethereum smart contracts. VeriSolid is based on earlier work (FSolidM) which allows to graphically specify Ethereum smart contracts as transitions systems, and to generate Solidity code from those specification. It uses model checking to verify smart contract models. Luu et al. [23] provide operational semantics of a subset of Ethereum bytecode called EtherLite, which forms the bases of their symbolic execution tool Oyente for analysing Ethereum smart contracts. Based on their tool they discovered a number of weaknesses in deployed smart contracts, including the DAO bug. Filliâtre et al. [16] introduced the Why3 system, which allows writing imperative programs in WhyML, an ML dialect used for programming and specification. The system can add pre-, post- and intermediate conditions to it but does not make use of weakest precondition. Why3 can generate verification conditions for Hoare triple, which are checked using variously automated and interactive theorem provers. Why3 is used in SPARK Ada to verify its verification conditions.

Finally, Agda features in several papers discussing verification of blockchains. Chakravarty et al. [13, 12] extend existing UTXO accounting rules to custom assets and asset bundles of various form. Their approach avoids any form of global state for registering assets. They provide a simple domain-specific language for forging policy scripts thus forgo the need for general-purpose smart contracts. They formalise and reason about their models in Agda. Chapman et al. [14] formalise System $F_{\omega\mu}$, which is polymorphic $\lambda$-calculus with higher-kindned and arbitrary recursive types, in Agda. System $F_{\omega\mu}$ correspondence to Plutus Core, which is the core of the smart contract language Plutus that is featured by the Cardano blockchain.

3 Operational Semantics for Bitcoin Script

We give a brief introduction of Bitcoin Script in Sect. 3.1, before defining its operational semantics in Sect. 3.2.

3.1 Introduction to Bitcoin Script

The scripting language for Bitcoin is stack-based [8], inspired by the programming language Forth [29]. Script has its own set of commands which are called opcodes. Each opcode is encoded as a byte. A full list of instructions with their meaning can be found in [9], which is the defacto specification of Script.

All opcodes considered in this paper have been formalized in Agda. A number of opcodes are used to push data onto the stack; we abstract away from details and write $\langle\text{number}\rangle$.
for the sequence of instructions that will push number onto the stack—in Agda we write \opPush\ sig instead of \<sig>\. Values on the stack are also interpreted as truth values, in which case any value >0 will be interpreted as true, and any other value as false.

We introduce a number of opcodes that are relevant to this paper:

- \opDUP\ duplicates the top element of the stack.
- \opHASH\ takes the top item of the stack and replaces it with its hash.
- \opEQUAL\ checks whether the top two elements in the stack are equal or not.
- \opVERIFY\ invalidates the transaction if the top stack value is false. The top item on the stack will be removed.
- \opCHECKSIG\ hashes the entire transaction, and checks whether the top two items on the stack form a correct pair of a signature and a public key for this hash.
- \opCHECKLOCKTIMEVERIFY\ fails if the time on the stack is greater than the current time.
- \opMULTISIG\ is the multisig instruction, which will be discussed in detail in Sect. 6.2.

Scripts can also contain control flow statements such as \opIF\. The verification of scripts involving control statements is more involved and will be considered in a follow-up paper.

In Bitcoin we consider the interplay between a locking script \scriptPubKey\ and an unlocking script \scriptSig\. First the unlocking script is executed. If it terminates successfully, the stack returned is taken as starting point for the locking script. If the locking script terminates successfully and returns true on the stack, then the part of the transaction dealing with these particular two scripts succeeds.

The main example in this paper is the pay-to-public-key-hash (P2PKH) script consisting of the following locking and unlocking scripts:

\begin{verbatim}
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUAL OP_VERIFY OP_CHECKSIG
scriptSig: <sig> <pubKey>
\end{verbatim}

The standard unlocking script \scriptSig\ pushes a signature \sig\ and a public key \pubKey\ onto the stack. The locking script \scriptPubKey\ checks whether \pubKey\ provided by the unlocking script hashes to the provided \pubKeyHash, and whether the signature is a signature for the message signed by the public key. Full details will be discussed in Sect. 5.

### 3.2 Operational Semantics

Opcodes like \opDUP\ operate on the stack; we define \Stack\ for the type of stacks in Agda. Opcodes like \opCHECKSIG\ take in addition the whole transaction as input. To abstract from the exact format of transactions and similar objects, we define a message type \Msg\ in Agda which is formed from numbers by using binary products and lists, and which will allow to represent transactions. Other opcodes like \opCHECKLOCKTIMEVERIFY\ refer to the current time, for which we define a type \Time\ in Agda. Therefore, the operational semantics of opcodes depends on \Time \times \Msg \times \Stack\ which we define in Agda as the record type \StackState\.

\footnote{We are using the terminology locking script and unlocking script from Chapt. 5 \cite{4}.}

\footnote{In the original version of Bitcoin both scripts were concatenated and executed. However, because Bitcoin script has non-local instructions (e.g. the conditionals OP_IF, OP_ELSE, OP_ENDIF) when concatenating the two scripts any non-local opcode occurring in the locking script (for instance as part of data) could be interpreted when running as the counterpart of a non-local opcode in the locking script and therefore result in an unintended execution of the unlocking script. By having a break point in between the two with only the stack passed on this is prevented.}
Verification of Bitcoin’s Smart Contracts in Agda

The type of all opcodes is given as `InstructionBasic`. Opcodes can also fail, for instance if there are not enough elements on the stack as required by the operation. Hence, the operational semantics of an instruction `p : InstructionBasic` is given as

\[ \text{[ p ]s : StackState} \rightarrow \text{Maybe StackState}. \]

Using set theoretic notation to display Agda’s algebraic data type, we can define `Maybe X` as `{ nothing } \cup \{ just \ x \mid x : X \}`. Here, `nothing` denotes undefined, and `just x` denotes the defined element `x`. `Maybe` forms a monad, with `return := just : A \rightarrow \text{Maybe A}` and the bind operation `(p \Rightarrow q : \text{Maybe B})` for `p : \text{Maybe A}` and `q : A \rightarrow \text{Maybe B}` defined by `(nothing \Rightarrow q) = nothing` and `(just a \Rightarrow q) = q \ a`.

The message and time never changes, so \[\text{[ p ]s}\] will, if it succeeds, only change the stack part of it. As an example, we can define the semantics of `opEqual` as follows:

\[
\text{[ opEqual ]s = liftStackToStackStateTransformer'} \ \text{executeStackEquality}
\]

Here `executeStackEquality` is of type `Stack \rightarrow \text{Maybe Stack}`, which is lifted to the required type by `liftStackToStackStateTransformer'`. `executeStackEquality` fails and returns `nothing` if the stack has height \(\leq 1\), and otherwise compares the two top numbers on the stack, replacing them by `1` in case they are equal, and by `0` otherwise.

The component `Time of stack` will be used to define the semantics of `opCheckLockTimeVerify`. `opCheckLockTimeVerify` fails if the current time is less than the top element on the stack. `Msg` will be used to define the semantics of `opCheckSig`.

Bitcoin script has instructions with very complex behaviour, an example is the instruction `OP_MULTISIG` introduced in Sect. 6.2. Some instructions depend on cryptographic functions for hashing and checking signatures. We abstract away from their concrete definition and take them as parameters of the modules of the Agda code.

General scripts are formalised in Agda as lists of `InstructionBasic`. Let `p` be a script. We define \[\text{[ p ]} : \text{StackState} \rightarrow \text{Maybe StackState}\] by monadic composition, that is

\[
\text{[ [ ] ]} := \text{just},
\]

for an operation `op`, script `q` and `s : StackState` define

\[
\text{[ op :: q ]s} := \text{[ op ]s \ \text{\Rightarrow [ q ]}}.
\]

It follows that \(\forall s : \text{Stack}.\ [ p \ \text{\:+: q }] s = [ p ] s \ \text{\Rightarrow [ q ]}\).

We lift as well \(\text{[ p ]}\) to \(s : \text{Maybe StackState}\) by defining \(\text{[ p ]'} s := s \ \text{\Rightarrow [ p ]}\).

4 Hoare Logic, Weakest Preconditions for Security, and Equational Reasoning

The standard way of giving a semantics for imperative programs is Hoare logic [20]. Assume a precondition \(\varphi\) and postcondition \(\psi\) are given, both of type `StackState \rightarrow \text{Set}`. Using a postfix operator \(\Rightarrow\), we lift \(s\) to \((\psi^+) : \text{Maybe StackState} \rightarrow \text{Set}\), defining \((\psi^+)\ nothing = \bot\) and \((\psi^+) \circ \text{just} = \psi\).

A Hoare triple, consisting of a precondition, a program, and a post-condition, expresses that if the precondition is satisfied before execution of the program, then the post-condition holds after executing it. We formalise Hoare triples in Agda as follows:

\[
\langle \varphi > p < \psi > := \forall s \in \text{StackState}. \varphi(s) \rightarrow (\psi^+) ([ p ] s)
\]

\[^3\] This is no problem in this paper, since the weakest preconditions only depend on the results returned by these functions, such as a check whether the part of the transaction to be signed is signed by a signature corresponding to a given public key. Weak cryptographic functions make it easier to find a corresponding locking script fulfilling the weakest precondition.
4.1 Weakest Precondition for Security

Hoare logic based on pre- and postconditions works well for safety critical systems, where the set of inputs can be controlled. It allows to deal with the interaction between different procedures of a program. An example of a commercial system for writing safety critical systems using Hoare logic is SPARK 2014 [1]. When dealing with security aspects, in particular access control, it is necessary to guard against malicious entries to a program. In this case, weakest preconditions is the more appropriate notion. It requires that the precondition not only is sufficient, but as well necessary for the postcondition to hold after executing the program:

\[
\varphi \iff p < \psi := \forall s \in \text{StackState}. \varphi(s) \iff (\psi \uparrow) (\llbracket p \rrbracket s)
\]

In Bitcoin we consider a locking script \( \text{scriptPubKey} \) and an unlocking script \( \text{scriptSig} \), see Section 3.1. The scripts are not concatenated and executed, but rather combined by executing them separately with the result of the first unlocking script serving as input to the execution of the subsequent locking script.\(^4\)

Let us fix an unlocking script \( \text{unlock} \) and a locking script \( \text{lock} \). Let \( \text{init} \) be the initial state consisting of an empty stack, and let \( \text{acceptState} \) be the accepting condition that the stack is non empty with top element representing true, i.e. \( >0 \). The combination of \( \text{unlock} \) and \( \text{lock} \) is accepted iff running \( \text{unlock} \) on \( \text{init} \) succeeds and running \( \text{lock} \) on the resulting stack results in a state that satisfies the accepting condition, i.e. iff \( \text{acceptState} \uparrow \) (\( \llbracket \text{lock} \rrbracket \) \( \llbracket \text{unlock} \rrbracket \) \( \text{init} \)).

Let \( \varphi \) be the weakest precondition of \( \text{lock} \), i.e. \( < \varphi > \text{iff lock} < \text{acceptState} > \). Then the acceptance condition is equivalent to \( < \varphi > \text{iff } \) running \( \text{unlock} \) on the initial state \( \text{init} \) produces a state fulfilling \( \varphi \). Hence, by determining the weakest precondition for the locking script w.r.t. the accepting condition we have obtained a characterisation of the set of unlocking scripts which unlock the locking script. Note that we do not define inductively all successful unlocking scripts, since they could be arbitrary complex programs, but instead characterise them by the output they produce.

4.2 Equational Reasoning with Hoare Triples

To support the verification of Bitcoin scripts with Hoare triples and weakest preconditions in Agda, we have developed a library in Agda for equational reasoning with Hoare triples. The library is inspired by what is described in Wadler et al. [34]. If we define 

\[
\varphi \iff p \psi := \forall s : \text{StackState}. \varphi(s) \iff (\psi) (\llbracket p \rrbracket s)
\]

We demonstrate our syntax by an example, assuming programs \( \text{prog1}, \text{prog2}, \text{prog3} \), and proofs

\[
\text{proof1} : \text{< precondition >iff prog1 < intermediateCond1 >}
\]
\[
\text{proof2} : \text{< intermediateCond1 >iff prog2 < intermediateCond2 >}
\]

\(^4\) Note the order: first the unlocking script is executed which pushes data on the stack. Then the locking script is executed, which checks whether the data provided by the locking script fulfill the conditions required.
8 Verification of Bitcoin’s Smart Contracts in Agda

proof3 : intermediateCond2 <=>p intermediateCond3
proof4 : < intermediateCond3 >iff prog3 < postcondition >

Then the proof for the Hoare triple for \( \text{prog1} \Leftrightarrow (\text{prog2} \Leftrightarrow \text{prog3}) \) is given as follows:

\[
\text{theorem} : < \text{precondition} >iff \text{prog1} \Leftrightarrow (\text{prog2} \Leftrightarrow \text{prog3}) < \text{postcondition} >
\]
\[
\text{theorem} = \text{precondition} \llp \text{prog1} \lhrp \text{proof1} \rrp \llp \text{prog2} \lhrp \text{proof2} \rrp \llp \text{prog3} \lhrp \text{proof3} \rrp \llp \text{postcondition} \llp \text{proof4} \rrp
\]

We will make use of this proof pattern in the following sections.

5 Proof of Correctness of the P2PKH script using the Step by Step Approach

This section explains the usage of our approach by providing an example of how to prove the correctness of the Pay to Public key Hash script (P2PKH) using step by step to obtain the weakest precondition. The Pay to Public Key Hash (P2PKH) is the most used type of script in Bitcoin transactions. The locking script, which depends on a public key hash, is defined as follows:5

\[
\text{scriptP2PKH}^b : (\text{pbkh} : \mathbb{N}) \rightarrow \text{BitcoinScriptBasic}
\]
\[
\text{scriptP2PKH}^b \text{pbkh} = \text{opDup} :: \text{opHash} :: (\text{opPush} \text{pbkh}) :: \text{opEqual} :: \text{opVerify} :: [ \text{opCheckSig} ]
\]

In this section we develop a readable weakest precondition of the P2PKH script and prove its correctness by working backwards instruction by instruction. Let \( \text{acceptState} \) be the accepting state where the stack is non-empty with top element \( >0 \). We define intermediate conditions \( \text{accept}_1, \text{accept}_2, \) etc, the weakest condition \( w\text{PreCondP2PKH} \), and proofs \( \text{correct-1}, \text{correct-2}, \) etc of corresponding Hoare triples w.r.t. the instructions of the Bitcoin script, working backwards starting from the last instruction \( \text{opCheckSig} \):

\[
\text{correct-1} : < \text{accept}_1 >iff([\text{opCheckSig}]) < \text{acceptState} >
\]
\[
\text{correct-2} : < \text{accept}_2 >iff([\text{opVerify}]) < \text{accept}_1 >
\]
\[
\text{correct-3} : < \text{accept}_3 >iff([\text{opEqual}]) < \text{accept}_2 >
\]
\[
\text{correct-4} : (\text{pbk} : \mathbb{N}) \rightarrow < \text{accept}_4 \text{pbk} >iff([\text{opPush pbk}]) < \text{accept}_3 >
\]
\[
\text{correct-5} : (\text{pbk} : \mathbb{N}) \rightarrow < \text{accept}_5 \text{pbk} >iff([\text{opHash}]) < \text{accept}_4 \text{pbk} >
\]
\[
\text{correct-6} : (\text{pbk} : \mathbb{N}) \rightarrow < \text{wPreCondP2PKH pbk} >iff([\text{opDup}]) < \text{accept}_5 \text{pbk} >
\]

The intermediate conditions can be read off from the operations. We present them in mathematical notation below, using the following conventions and abbreviations: \( m : \text{Msg} \), \( t : \mathbb{N} \) denotes time, \( st, st' : \text{Stack}, x : \mathbb{N} \); for brevity we omit types after \( \exists \) quantifiers. We use here and in the remaining paper 4 for operations where the \( \text{StackState} \) argument has been unfolded into its components.

---

5 The superscript \( ^b \) for basic indicates that we are referring to the language of Bitcoin without opcodes for conditionals – conditionals would require an extension of the state space.
In Agda, these formulas are defined by case distinction on the stack. As examples, the code for the accept condition (acceptState) and the weakest precondition (wPreCondP2PKH) is as follows:

\[
\text{acceptState}^s : \text{StackPredicate} \\
\text{acceptState}^s \text{ time } \text{msg}_1 [] = \bot \\
\text{acceptState}^s \text{ time } \text{msg}_1 (x :: \text{stack}_1) = \text{NotFalse } x \\
\]

\[
\text{wPreCondP2PKH}^s : (pbkh : N) \rightarrow \text{StackPredicate} \\
\text{wPreCondP2PKH}^s \text{ pbkh time } m [] = \bot \\
\text{wPreCondP2PKH}^s \text{ pbkh time } m (x :: []) = \bot \\
\text{wPreCondP2PKH}^s \text{ pbkh time } m (\text{pubKey} :: \text{sig} :: st) = \\
(\text{hashFun } \text{pubKey} \equiv \text{pbkh} ) \land \text{IsSigned } m \text{ sig pbkh} \\
\]

Using our syntax for equational reasoning, we can prove the weakest precondition for the P2PKH script as follows:

\[
\text{theoremP2PKH} : (pbkh : N) \rightarrow < \text{wPreCondP2PKH}^s \text{ pbkh } > \text{iff scriptP2PKH}^s \text{ pbkh } < \text{acceptState} > \\
\text{theoremP2PKH} \text{ pbkh} = \text{wPreCondP2PKH}^s \text{ pbkh } <><>([\text{opDup }])<\text{correct-6 pbkh}> \\
\text{accept}_5 \text{ pbkh } <><>([\text{opHash }])<\text{correct-5 pbkh}> \\
\text{accept}_4 \text{ pbkh } <><>([\text{opPush pbkh }])<\text{correct-4 pbkh}> \\
\text{accept}_3 \text{ pbkh } <><>([\text{opEqual }])<\text{correct-3}> \\
\text{accept}_2 \text{ pbkh } <><>([\text{opVerify }])<\text{correct-2}> \\
\text{accept}_1 \text{ pbkh } <><>([\text{opCheckSig }])<\text{correct-1}>e \text{acceptState} p \\
\]

The locking script will be accepted if, after executing the code starting with the stack returned by the unlocking script, the accept condition acceptState is fulfilled. The verification conditions and proofs were developed by working backwards starting from the last instruction and determining the weakest preconditions (i.e. conditions accept_i) w.r.t. the end piece of the script starting with that instruction and the accept condition as post-condition. We continued in this way until we reached the first instruction and obtained the weakest precondition for the locking script. theoremP2PKH is using single instructions in order to prove the correctness of P2PKH. The proofs correct-1, correct-2, etc are done by following the case distinctions made in the corresponding verification conditions. The harder direction is to prove that they are actually \textit{weakest} preconditions.
6 Proof of Correctness using Symbolic Execution

In this section, we will introduce a second method for obtaining readable representations of weakest preconditions of Bitcoin scripts. This method is based on symbolic execution of the Bitcoin script, and investigating the sequence of case distinctions carried out during the execution. We will consider three examples: The first will be the P2PKH script which we analysed already. We use it to explain the method and provide a second approach to determine and verify the already obtained weakest precondition. The second example will consider the multisig script which is a direct application of the OP_MULTISIG instruction. The third example will see an application of a combination of both methods.

6.1 Example: P2PKH Script

When applying the symbolic evaluation method to the P2PKH script and analysing the sequence of case distinctions carried out, we will see that there will be exactly one path through the tree of case distinctions which results in an accepting condition. The conjunction of the cases that determine this path will form the weakest precondition. (In examples with more than one accepting path we would take the disjunction of the conditions for each path.) We will prove that the weakest precondition is indeed a representation of the weakest precondition by developing an equivalent program $p2pkhFunctionDecoded$ and showing that it fulfils the weakest precondition.

We start by creating symbolic variables (we use Agda’s postulates) for the parameters such as $pbkh$, $msg_1$, $stack_1$, etc. This allows us to evaluate expressions symbolically and to determine the function $p2pkhFunctionDecoded$. Afterwards, we stop using those postulates (they were defined as private) and prove that the result of evaluating the P2PKH script for arbitrary parameters is equivalent to $p2pkhFunctionDecoded$.

When evaluating $[\text{scriptP2PKH}^{b} \; pbkh \; \text{msg}_1 \; \text{time}_1 \; \text{stack}_1]$ we obtain

```
executeStackDup stack₁
\lambda \text{stack₂} \rightarrow \text{executeOpHash} \; \text{stack₂} \\
\lambda \text{stack₃} \rightarrow \text{executeStackEquality} \; (pbkh :: \text{stack₃}) \\
\lambda \text{stack₄} \rightarrow \text{executeStackVerify} \; \text{stack₄} \\
\lambda \text{stack₅} \rightarrow \text{executeStackCheckSig} \; \text{msg}_1 \; \text{stack₅}
```

The do notation (a widely used Haskell notation adapted to Agda) provides an alternative syntax for the same expression making it appear as an imperative program if one reads $←$ as assignments. It demonstrates that we are consecutively executing the instructions, with the possibility of aborting in each step:

```
do
  \text{stack₂} ← \text{executeStackDup} \; \text{stack₁}
  \text{stack₃} ← \text{executeOpHash} \; \text{stack₂}
  \text{stack₄} ← \text{executeStackEquality} \; (pbkh :: \text{stack₃})
  \text{stack₅} ← \text{executeStackVerify} \; \text{stack₄}
  \text{executeStackCheckSig} \; \text{msg}_1 \; \text{stack₅}
```

At this point further reduction is blocked by the first line of the previous expression, because $\text{executeStackDup} \; \text{stack₁}$ makes a case distinction on $\text{stack₁}$. Therefore, we introduce a symbolic case distinction on $\text{stack₁}$:

- $[\text{scriptP2PKH}^{b} \; pbkh \; \text{msg}_1 \; \text{time}_1 \; \text{msg}_1]$ evaluates to nothing.
- $[\text{scriptP2PKH}^{b} \; pbkh \; \text{msg}_1 \; \text{time}_1 \; \text{msg}_1 \; (pbk :: \text{stack}_1)]$ evaluates to what in do notation can be written as
The proof is a simple distinction of cases following the cases defining the weakest precondition that we have determined before.

Evaluation of the latter expression is blocked by the function `executeStackVerify` which makes a case distinction on the argument `cmp` of `abstrFun`.

- `abstrFun [] (suc x1)` evaluates to `nothing`.
- `abstrFun (sig1 :: stack1) (suc x1)` evaluates to `just (boolToNat (isSigned msg1 sig1 pbk) :: stack1)`.

In order to normalise further, `executeStackCheckSig` needs to make a case distinction on `stack2`, so we carry out a symbolic case distinction on that argument:

- `abstrFun [] (suc x1)` evaluates to `nothing`.
- `abstrFun (sig1 :: stack1) (suc x1)` evaluates to `just (boolToNat (isSigned msg1 sig1 pbk) :: stack1)`.

We can now read off the weakest precondition. The only path which ends up in a `just` result is when the stack is non empty of the form `pbk :: stack1`, `compareNaturals pbkh (hashFun pbk)` evaluates to `suc x1`, i.e. it must be >0. Furthermore, in this case `stack1` needs to be itself non empty. For `stack1 = sig1 :: stack2`, the result returned is `just (boolToNat (isSigned msg1 sig1 pbk) :: stack1)`, which fulfils the accept condition if `boolToNat (isSigned msg1 sig1 pbk) > 0`. The latter is the case if `isSigned msg1 sig1 pbk` is true.

Furthermore, `compareNaturals n m` returns 1 if `n, m` are equal otherwise 0, so it is >0 if `n = m`. Therefore the P2PKH locking script succeeds with an output stack fulfilling the acceptance condition, iff the input stack has height at least two, and if it is `pbk :: sig1 :: stack2`, then `pbkh` is equal to `hashFun pbk`, and `isSigned msg1 sig1 pbk` is true. That is exactly the weakest precondition that we have determined before.

In order to prove correctness, we first determine a more Agda style formulation of the result of evaluation of the P2PKH script, which we derive from the previous symbolic evaluation:

```agda
p2pkhFunctionDecoded : (pbkh : N)(msg1 : Msg)(stack1 : Stack) → Maybe Stack
p2pkhFunctionDecoded pbkh msg1 [] = nothing
p2pkhFunctionDecoded pbkh msg1 (pbk :: stack1) = p2pkhFunctionDecodedaux1 pbk msg1 stack1 (compareNaturals pbkh (hashFun pbk))

p2pkhFunctionDecodedaux1 : (pbk : N)(msg1 : Msg)(stack1 : Stack)(cpRes : N) → Maybe Stack
p2pkhFunctionDecodedaux1 pbk msg1 [] cpRes = nothing
p2pkhFunctionDecodedaux1 pbk msg1 (sig1 :: stack1) zero = nothing
p2pkhFunctionDecodedaux1 pbk msg1 (sig1 :: stack1) (suc cpRes) =
  just (boolToNat (isSigned msg1 sig1 pbk) :: stack1)
```

We prove that this function is equivalent to the result of evaluating the P2PKH script. The proof is a simple distinction of cases following the cases defining `p2pkhFunctionDecoded`:
We now show that the extracted weakest precondition is a weakest precondition for the extracted program:\footnote{\emph{<\_\_>_g<\_\_> \textit{is the generalisation of <\_\_>iff<\_\_> \textit{where Bitcoin scripts are replaced by Agda functions \textit{StackState} \rightarrow \textit{StackState}; <\_\_>_g\textit{<\_\_> \textit{is the version, where the \textit{StackState} is unfolded into its components.}}}\\}

\[
\text{lemmaPTKHcoraux} : (\text{pbkh} : \mathbb{N}) \rightarrow < \text{weakestPreConditionP2PKH}^g\text{pbkh} >^g_s \\
(\lambda \text{time} \; \text{msg}, \text{stack} \rightarrow \text{p2pkhFunctionDecoded pbkh msg stack} ) \\
< \text{acceptState}^g > 
\]

Afterwards, this is transferred into a proof of the weakest precondition for the P2PKH script, using the equality proof from before:

\[
\text{theoremPTPKHcor} : (\text{pbkh} : \mathbb{N}) \\
\rightarrow < \text{wPreCondP2PKH}^g \text{pbkh} >\text{iff scriptP2PKH}^g \text{pbkh} < \text{acceptState} > 
\]

### 6.2 Example: MultiSig Script (P2MS)

The \texttt{OP\_MULTISIG} instruction is an instruction which has a complex behaviour: It assumes that the top elements of the stack are as follows:

\[
n :: \text{pbk}_n :: \cdots :: \text{pbk}_2 :: \text{pbk}_1 :: k :: \text{sig}_k :: \cdots :: \text{sig}_2 :: \text{sig}_1 :: \text{dummy}
\]

\texttt{OP\_MULTISIG} checks whether the \textit{k} signatures are signatures corresponding to \textit{k} of the \textit{n} public keys for the msg to be signed, where the matching public keys are in the same order as the signatures. Observe that when pushed from a script, the public keys and signatures appear in reverse order on the stack, as \texttt{pbk}_1 is pushed first onto the stack, etc. The \texttt{dummy} element occurs due to a mistake in the Bitcoin protocol, which has not been corrected as it would require a hard fork.

The operational semantics is given by a function \texttt{executeMultiSig}, which fetches the data from the stack as described before. It fails if there are not enough elements on the stack and otherwise returns \texttt{just (boolToNat (cmpSigs msg sigs pbks :: restStack), where sigs and pbks are the signatures and public keys fetched from the stack in reverse order, and restStack is the remainder of the stack. The function \texttt{cmpSigs} compares whether signatures correspond to public keys and is defined as follows:

\[
\text{cmpSigs} : (\text{msg} : \text{Msg})(\text{sigs pbks} : \text{List} \; \mathbb{N}) \rightarrow \text{Bool} \\
\text{cmpSigs msg [] pbkeys} = \text{true} \\
\text{cmpSigs msg (sig :: sigs) []} = \text{false} \\
\text{cmpSigs msg (sig :: sigs) (pbk :: pbks)} = \text{cmpSigsAux msg sigs pbks sig (isSigned msg sig pbk)}
\]

\[
\text{cmpSigsAux} : (\text{msg} : \text{Msg})(\text{sigs pbks} : \text{List} \; \mathbb{N})(\text{sig} : \mathbb{N})(\text{testRes} : \text{Bool}) \rightarrow \text{Bool} \\
\text{cmpSigsAux msg sigs pbks sig false} = \text{cmpSigs msg (sig :: sigs) pbks} \\
\text{cmpSigsAux msg sigs pbks sig true} = \text{cmpSigs msg sigs pbks}
\]

In the following, we consider the two out of four Pay to MultiSig Script (P2MS). The locking script MultiSig script P2MS applies \texttt{OP\_MULTISIG} to 2 signatures and 4 public keys.
It pushes 2 as the number of required signatures, four public keys, and 4 as the number of required public keys on the stack and executes `OP_MULTISIG`. It is formalised in Agda as follows:

\[
\text{multiSigScript2-4}^b : (pbk_1 \, pbk_2 \, pbk_3 \, pbk_4 : \text{N}) \rightarrow \text{BitcoinScriptBasic}
\]

\[
\text{multiSigScript2-4}^b \, \text{pbk}_1 \, \text{pbk}_2 \, \text{pbk}_3 \, \text{pbk}_4 = (\text{opPush} \, 2) :: (\text{opPush} \, \text{pbk}_3) :: (\text{opPush} \, \text{pbk}_2) :: (\text{opPush} \, \text{pbk}_1) :: (\text{opPush} \, 4) :: [\text{opMultiSig}]
\]

We will use the second approach of determining a readable form of the weakest precondition and proving correctness by symbolic evaluation for `multiSigScript2-4^b` because `opMultiSig` has a very complex precondition that is difficult to handle. It requires that the stack contains the number of public keys, then the public keys themselves, then the number of signatures and the signatures, and a dummy element.

The `multiSigScript2-4^b` script is easier to handle, since, after the data has been inputted, the number of required signatures is known, and the public keys are already provided by the script. Hence it will be easier to determine the weakest precondition for `multiSigScript2-4^b` and prove that it is correct. In Subsect. 6.3 we will apply the step by step approach to determine a readable form of the weakest precondition and prove correctness to a script of which `multiSigScript2-4^b` is one part. This will demonstrate that in some cases it is beneficial to interleave the two processes, and apply the second method to sequences of instructions while applying the first approach to the resulting sequences of instructions instead of single instructions.

We start the symbolic evaluation by computing the normal form of

\[
[\text{multiSigScript2-4}^b \, \text{pbk}_1 \, \text{pbk}_2 \, \text{pbk}_3 \, \text{pbk}_4]^\ast \text{time}_1 \, \text{msg}_1 \, \text{stack}_1
\]

and obtain

\[
\text{executeMultiSig} \, \text{msg}_1 \, (\text{pbk}_1 :: \text{pbk}_2 :: \text{pbk}_3 :: [ \text{pbk}_4 ]) \, \text{time}_1 \, \text{msg}_1 \, \text{stack}_1 \]

Here, `executeMultiSig` is one of the auxiliary functions in the definition of `executeMultiSig`. That expression makes a case distinctions on `stack_1` and returns:

- **nothing** when the stack has height at most 2.
- Otherwise, when the stack is of the form `sig_2 :: sig_1 :: dummy :: stack_1`, it reduces to

  \[
  \text{just} \, (\text{boolToNat} \, (\text{cmpSigsMultiSigAux} \, \text{msg}_1 \, [ \text{sig}_2 ] \, (\text{pbk}_2 :: \text{pbk}_3 :: [ \text{pbk}_4 ]) \, \text{sig}_1, \text{isSigned msg}_1 \, \text{sig}_1 \, \text{pbk}_1)) :: \text{stack}_1
  \]

The program has succeeded because we obtain **just** as a result of the evaluation. We now need to check whether the result fulfils the accept condition. For this the top element of the stack needs to be $>0$, which is the case if

\[
\text{cmpSigsMultiSigAux} \, \text{msg}_1 \, [ \text{sig}_2 ] \, (\text{pbk}_2 :: \text{pbk}_3 :: [ \text{pbk}_4 ]) \, \text{sig}_1 \, \text{isSigned msg}_1 \, \text{sig}_1 \, \text{pbk}_1
\]

returns **true**. Therefore, we perform symbolic case distinctions in the following way:

- In case `isSigned msg_1 \, \text{sig}_1 \, \text{pbk}_1` evaluates to **true**, the reduction continues to

  \[
  \text{cmpSigsMultiSigAux} \, \text{msg}_1 \, [] \, (\text{pbk}_3 :: [ \text{pbk}_4 ]) \, \text{sig}_2 \, \text{isSigned msg}_1 \, \text{sig}_2 \, \text{pbk}_2,
  \]

  which makes a case distinction on `isSigned msg_1 \, \text{sig}_2 \, \text{pbk}_2`.

- If that expression returns again **true**, we obtain **true**.

- If it returns false, we obtain

  \[
  \text{cmpSigsMultiSigAux} \, \text{msg}_1 \, [] \, [ \text{pbk}_4 ] \, \text{sig}_2 \, \text{isSigned msg}_1 \, \text{sig}_2 \, \text{pbk}_3
  \]

  which makes a case distinction on `isSigned msg_1 \, \text{sig}_2 \, \text{pbk}_3`.

- In case of **true**, we obtain **true**.

- Otherwise the case distinctions continue, see the the git repository [32] for the full unfolding.

In total we see that we obtain **true** iff one of the following cases holds:
14 Verification of Bitcoin’s Smart Contracts in Agda

- \((\text{isSigned } \text{msg} \ 1 \ 1 \ 1) \land (\text{isSigned } \text{msg} \ 1 \ 1 \ 2)\)
- \((\text{isSigned } \text{msg} \ 1 \ 1 \ 1) \land \neg (\text{isSigned } \text{msg} \ 1 \ 1 \ 2) \land (\text{isSigned } \text{msg} \ 1 \ 1 \ 3)\)
- \((\text{isSigned } \text{msg} \ 1 \ 1 \ 1) \land \neg (\text{isSigned } \text{msg} \ 1 \ 1 \ 2) \land (\text{isSigned } \text{msg} \ 1 \ 1 \ 3) \land \neg (\text{isSigned } \text{msg} \ 1 \ 2 \ 3) \land (\text{isSigned } \text{msg} \ 1 \ 2 \ 4)\)

or many more cases.

These cases can be simplified to an equivalent disjunction of the following cases:
- \((\text{isSigned } \text{msg} \ 1 \ 1 \ 1) \land (\text{isSigned } \text{msg} \ 1 \ 1 \ 2)\)
- \((\text{isSigned } \text{msg} \ 1 \ 1 \ 1) \land (\text{isSigned } \text{msg} \ 1 \ 1 \ 3)\)
- \((\text{isSigned } \text{msg} \ 1 \ 1 \ 1) \land (\text{isSigned } \text{msg} \ 1 \ 1 \ 4)\)
- \((\text{isSigned } \text{msg} \ 1 \ 1 \ 1) \land (\text{isSigned } \text{msg} \ 1 \ 2 \ 3) \land (\text{isSigned } \text{msg} \ 1 \ 2 \ 4)\)
- \((\text{isSigned } \text{msg} \ 1 \ 1 \ 1) \land (\text{isSigned } \text{msg} \ 1 \ 2 \ 3) \land (\text{isSigned } \text{msg} \ 1 \ 2 \ 4)\)
- \((\text{isSigned } \text{msg} \ 1 \ 1 \ 1) \land (\text{isSigned } \text{msg} \ 1 \ 3 \ 3) \land (\text{isSigned } \text{msg} \ 1 \ 3 \ 4)\)
- \((\text{isSigned } \text{msg} \ 1 \ 1 \ 1) \land (\text{isSigned } \text{msg} \ 1 \ 3 \ 3) \land (\text{isSigned } \text{msg} \ 1 \ 3 \ 4)\)
- \((\text{isSigned } \text{msg} \ 1 \ 1 \ 1) \land (\text{isSigned } \text{msg} \ 1 \ 3 \ 3) \land (\text{isSigned } \text{msg} \ 1 \ 3 \ 4)\)

So we obtain as weakest precondition that the stack must have height at least 3, and if it is of the form \(\text{sig}_2 :: \text{sig}_1 :: \text{dummy} :: \text{stack}_3\) then the above disjunction must hold.

Using the same case distinctions as they occurred in the symbolic evaluation above we can now prove:

\[
\text{theoremCorrectnessMultiSig-2-4} : (pbk_1 \ pbk_2 \ pbk_3 \ pbk_4 : \mathbb{N}) \rightarrow <\text{weakestPreConditionMultiSig-2-4} pbk_1 \ pbk_2 \ pbk_3 \ pbk_4 > \iff \text{multiSigScript2-4} b pbk_1 \ pbk_2 \ pbk_3 \ pbk_4 <\text{stackPred2SPred acceptState} >\]

6.3 Example: Combining the two Methods

In this subsection we show how to verify a script which consists of a simple script checking time and the multisig script from the previous subsection. To determine a readable form of the weakest precondition and proving correctness we will combine both of our techniques: The weakest precondition for the multisig script has been determined by symbolic evaluation in the previous subsection. The weakest precondition for the simple time checking script will be obtained directly, as it is very simple. When we consider the combined scripts we will use the first method of moving backwards step by step. However, instead of using single instructions in each step, we now use several instructions as a single step.

We define the checktime script as follows:

\[
\text{checkTimeScript}^b : (\text{time}_1 : \text{Time}) \rightarrow \text{BitcoinScriptBasic}
\]

\[
\text{checkTimeScript}^b \text{time}_1 = (\text{opPush } \text{time}_1) :: \text{opCHECKLOCKTIMEVERIFY} :: [ \text{opDrop } ]
\]

If we define

\[
\text{timeCheckPreCond} : (\text{time}_1 : \text{Time}) \rightarrow \text{StackPredicate}
\]

\[
\text{timeCheckPreCond} \text{time}_1 \text{time}_2 \text{msg} \text{stack}_1 = \text{time}_1 \leq \text{time}_2
\]

we can define its weakest precondition relative to a post condition \(\phi\) only affecting the stack as in the following theorem:

\[
\text{theoremCorrectnessTimeCheck} : (\phi : \text{StackPredicate}) (\text{time}_1 : \text{Time}) \rightarrow <\text{stackPred2SPred} (\text{timeCheckPreCond} \text{time}_1 \land \text{sp} \phi) > \iff \text{checkTimeScript}^b \text{time}_1 <\text{stackPred2SPred} \phi >
\]
Now we can determine the weakest precondition for the combined script and prove its correctness as follows:

\[
\text{theoremCorrectnessCombinedMultiSigTimeCheck} : (time_1 : \text{Time}) \to \text{TimeCheckCondPreCond} time_1 \land \text{sp weakestPreConditionMultiSig-2-4} pbk1 pbk2 pbk3 pbk4 > \iff \text{checkTimeScript} time_1 \leftrightarrow \text{multiSigScript2-4} pbk1 pbk2 pbk3 pbk4
\]

< acceptState >

\[
\text{theoremCorrectnessCombinedMultiSigTimeCheck} \quad \text{time}_1 \quad \text{pbk1 pbk2 pbk3 pbk4} = \text{stackPred2SPred} \quad \text{timeCheckPreCond} \quad \text{time}_1 \land \text{sp weakestPreConditionMultiSig-2-4} pbk1 pbk2 pbk3 pbk4
\]

\[
\text{checkTimeScript} time_1 \leftrightarrow \text{multiSigScript2-4} pbk1 pbk2 pbk3 pbk4
\]

< acceptState >

\[
\text{theoremCorrectnessCombinedMultiSigTimeCheck} \quad \text{time}_1 \quad \text{pbk1 pbk2 pbk3 pbk4} = \text{stackPred2SPred} \quad \text{timeCheckPreCond} \quad \text{time}_1 \land \text{sp weakestPreConditionMultiSig-2-4} pbk1 pbk2 pbk3 pbk4
\]

\[
\text{checkTimeScript} time_1 \leftrightarrow \text{multiSigScript2-4} pbk1 pbk2 pbk3 pbk4
\]

\[
\text{stackPred2SPred} \quad \text{weakestPreConditionMultiSig-2-4} pbk1 pbk2 pbk3 pbk4
\]

\[
\text{checkTimeScript} time_1 \leftrightarrow \text{multiSigScript2-4} pbk1 pbk2 pbk3 pbk4
\]

\[
\text{stackPred2SPred} \quad \text{acceptState} \land \text{p}
\]

The weakest precondition states that the state time is $\geq time_1$, and that the weakest precondition for the multisig script is fulfilled ($\land \text{sp}$ forms the conjunction of the two conditions). For proving it we used a combination of both methods, the second method was used to determine preconditions for the two parts of the scripts, and the first method, where we used whole scripts instead of basic instructions, was used to determine the combined weakest precondition.

## 7 Conclusion

In this paper, we have implemented and tested two methods for developing human-readable weakest preconditions and proving their correctness. These methods can help smart contract developers to fill the validation gap between user requirements and formal specification. We have argued that weakest preconditions in Hoare logic is the correct notion for specifying the security property of access control. We have applied our approaches to Pay to Public Key Hash (P2PKH), Pay to Multi-Signature (P2MS), and a combination of P2MS with a time lock. The whole approach has been formalised in Agda [32].

In future work, we will treat other opcodes such as conditionals like OP_IF, and integrate them into our methodologies. The verification of scripts involving control statements is more involved and will require a careful refinement of our current methods.

Another route for future research is to develop our approach into a framework for developing smart contracts that are correct by construction.
Verification of Bitcoin’s Smart Contracts in Agda

References

1. Adacore. SPARK 2014, Retrieved 9 November 2021. https://www.adacore.com/about-spark.
2. Mouhamad Almakhour, Layth Sliman, Abdel Ellatif Samhat, and Abdelhamid Mellouk. Verification of smart contracts: A survey. Pervasive and Mobile Computing, 67:1–19, 2020. doi:http://dx.doi.org/10.1016/j.pmcj.2020.101227.
3. Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, pages 66–77, New York, NY, USA, 2018. Association for Computing Machinery. doi:https://doi.org/10.1145/3167084.
4. Andreas M Antonopoulos. Mastering Bitcoin: Programming the open blockchain. O’Reilly Media, Inc., 2017. URL: https://www.oreilly.com/library/view/mastering-bitcoin-2nd/9781491954379/.
5. Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In Financial Cryptography and Data Security, pages 494–509, Cham, 2017. Springer International Publishing. doi:https://doi.org/10.1007/978-3-319-70278-0_31.
6. Massimo Bartoletti and Roberto Zunino. BitML: A Calculus for Bitcoin Smart Contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS ’18, pages 83–100, New York, NY, USA, 2018. Association for Computing Machinery. doi:http://dx.doi.org/10.1145/3243734.3243795.
7. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal Verification of Smart Contracts: Short Paper. In Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS ’16, pages 91–96, New York, NY, USA, 2016. Association for Computing Machinery. doi:https://doi.org/10.1145/2993600.2993611.
8. Stefano Bistarelli, Ivan Mercanti, and Francesco Santini. An Analysis of Non-standard Bitcoin Transactions. In 2018 Crypto Valley Conference on Blockchain Technology (CVCBT), pages 93–96, 2018. doi:http://dx.doi.org/10.1109/CVCBT.2018.00016.
9. Bitcoin Community. Welcome to the Bitcoin Wiki. Availabe from https://en.bitcoin.it/wiki/Bitcoin, 2010.
10. Harris Brakmić. Bitcoin Script, pages 201–224. Apress, Berkeley, CA, 2019. doi:http://dx.doi.org/10.1007/978-1-4842-5522-3_7.
11. Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.org/en/whitepaper. [Online; accessed 10-December-2021].
12. Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. Native Custom Tokens in the Extended UTXO Model. In Leveraging Applications of Formal Methods, Verification and Validation: Applications, pages 89–111, Cham, 2020. Springer International Publishing. doi:https://doi.org/10.1007/978-3-030-61467-6_7.
13. Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnentferner. UTXOma: UTXO with Multi-asset Support. In Leveraging Applications of Formal Methods, Verification and Validation: Applications, pages 112–130, Cham, 2020. Springer International Publishing. doi:https://doi.org/10.1007/978-3-030-61467-6_8.
14. James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. System F in Agda, for Fun and Profit. In Mathematics of Program Construction, pages 255–297, Cham, 2019. Springer International Publishing. doi:https://doi.org/10.1007/978-3-030-33636-3_10.
15. Edmund M. Clarke and Jeanette M. Wing. Formal methods: State of the art and future directions. ACM Comput. Surv., 28(4):626–643, dec 1996. doi:http://dx.doi.org/10.1145/242223.242225.
Verification of Bitcoin’s Smart Contracts in Agda

31 Anton Setzer. Modelling Bitcoin in Agda. CoRR, abs/1804.06398, 2018. URL: http://arxiv.org/abs/1804.06398, arXiv:1804.06398.

32 Anton Setzer, Fahad Alhabardi, and Bogdan Lazer. Verification Of Smart Contracts With Agda. Available from https://github.com/fahad1985lab/Smart--Contracts--Verification--With--Agda, 2021.

33 N. Szabo. Smart Contracts: Building Blocks for Digital Markets. EXTROPY: The Journal of Transhumanist Thought, (16), 18(2), 1996. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/L0Twinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

34 Philip Wadler, Wen Kokke, and Jeremy G. Siek. Programming Language Foundations in Agda. Online textbook, July 2020. URL: https://plfa.github.io/Equality/.

35 Maximilian Wohrer and Uwe Zdun. Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity. In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pages 2–8, 2018. doi:http://dx.doi.org/10.1109/IWBOSE.2018.8327565.