Verification of Bitcoin Script in Agda Using Weakest Preconditions for Access Control

Fahad Alhabardi[1], Anton Setzer, Arnold Beckmann, and Bogdan Lazer
Department of Computer Science
Swansea University

December 7, 2022

Table of Contents

1. Background
    - Smart Contracts
    - EVM vs Script
    - Bitcoin Script Language
2. Contribution
3. The Proof Assistant Agda
4. Bitcoin Script
    - Main example (P2PKH)
5. Operational semantics
6. Hoare logic and weakest pre-conditions
    - Human-readable weakest pre-condition
    - Our library
7. Proof of Correctness of the P2PKH script using Step-by-Step approach
8. Conclusion

Smart Contracts

- What are smart contracts?
  Smart contracts are transactions that are defined through software and executed automatically when conditions in the blockchains are met.
- Smart contracts in cryptocurrency are written in many languages:
  - Script in case of Bitcoin [4, Ch 6].
  - Solidity in case of Ethereum [3, Ch 7].

EVM vs Script

- Ethereum Virtual Machine is based on Bitcoin Script.
- EVM [11]:
    - EVM extends and modifies Bitcoin Script, especially it
        - adds loops (jumps),
        - allows calls to other contracts,
        - adds cost of execution of instructions (gas) to guarantee termination.
- Bitcoin Script [11]:
    - without loops.
    - without possibility to calling other contracts.

Bitcoin Script Language

- The scripting language for Bitcoin is stack-based, and similar to Forth.
- The script in Bitcoin has a set of commands called Operation Codes such as OP_ ADD, OP_ EQUAL etc...
- Several standards scripts are used in Bitcoin such as the pay-to-public-key-hash (P2PKH) script.

Contribution

- Our verification focuses on Pay to Public Key Hash (P2PKH) and Pay to Multisig (P2MS) [1].
- We have introduced an operational semantics of the script commands used in P2PKH and P2MS, which we have formalised in the Agda proof assistant and reason about using Hoare triples [1].
- Use of weakest pre-condition in order to formalise the correctness of smart contracts.
- Two methodologies for obtaining human-readable weakest pre-conditions [1].
  - A step-by-step approach (backwards instruction by instruction).
  - Symbolic execution.
- To support our verification, we develop a library [1].

The proof assistant Agda

- A dependently typed functional programming language that expands the Martin-Löf constructive type theory [9].
- Introduced by Ulf Norell [10].
- Agda's features [7, 8, 2, 6] include but not are limited:
  - Inductive and inductive-recursive data types.
  - Pattern matching
  - Completely support for Unicode.
  - Coverage and termination checkers.
- The Agda standard library defines the inductive type of natural numbers as follows:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

Cont.

- As an example, we define the inductive type of InstructionBasic in Agda is as follows:

```
data InstructionBasic : Set where
  opEqual opAdd opVerify opDup : InstructionBasic
  opMultiSig opCHECKLOCKTIMEVERIFY : InstructionBasic
```

- Another example to define a function in Agda is as follows:

```
executeStackEquality : Stack → Maybe Stack
executeStackEquality [] = nothing
executeStackEquality (n :: []) = nothing
executeStackEquality (n :: m :: e)
  = just ((compareNaturals n m) :: e)
```

Bitcoin Script

- Several opcodes have been introduced and formalised in Agda [1].
  - **OP_ADD** adds the two top elements of the stack together
  - **OP_DUP** duplicates the top element of the stack.
  - **OP_HASH** takes the top item of the stack and replaces it with its hash.
  - **OP_EQUAL** checks whether the top two elements in the stack are equal or not.
  - **OP_VERIFY** invalidates the transaction if the top stack value is false. The top item on the stack will be removed.

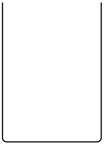
Cont.

- **OP_CHECKSIG** 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.
- **OP_CHECKLOCKTIMEVERIFY** fails if the time on the stack is greater than the current time.
- Bitcoin scripts that use non-local instructions such as **OP_IF**, **OP_ELSE**, and **OP_ENDIF** [1].

Cont.

- Simple example of local instructions:

<2> <3> *OP_ADD* <5> *OP_EQUAL*

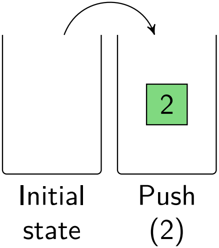


Initial state

Cont.

- Simple example of local instructions:

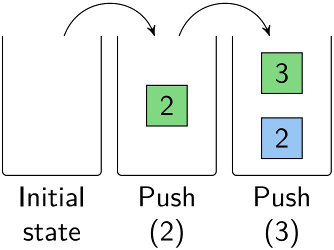
<2> <3> *OP_ADD* <5> *OP_EQUAL*



Cont.

- Simple example of local instructions:

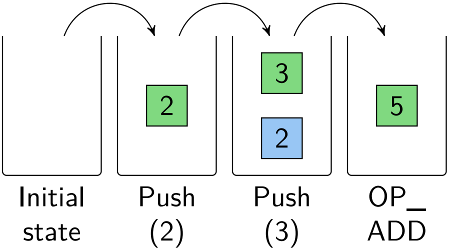
<2> <3> *OP_ADD* <5> *OP_EQUAL*



Cont.

- Simple example of local instructions:

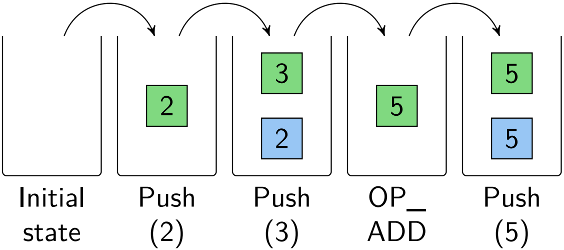
<2> <3> *OP_ADD* <5> *OP_EQUAL*



Cont.

- Simple example of local instructions:

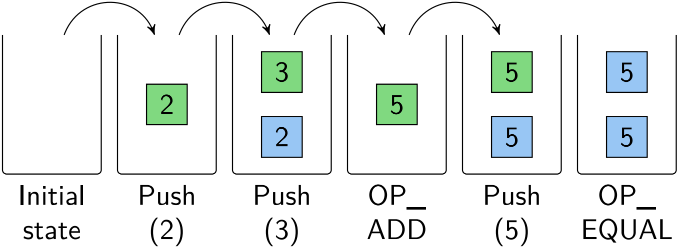
<2> <3> *OP_ADD* <5> *OP_EQUAL*



Cont.

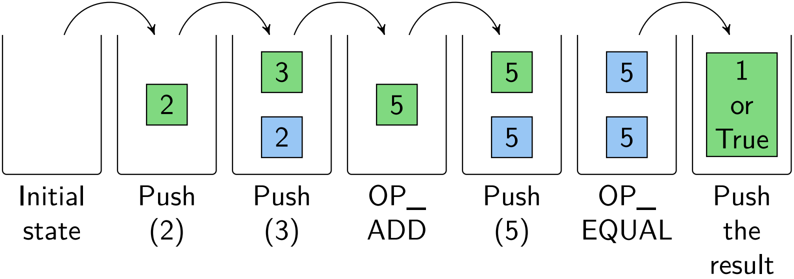
- Simple example of local instructions:

<2> <3> *OP_ADD* <5> *OP_EQUAL*



- Simple example of local instructions:

<2> <3> *OP_ADD* <5> *OP_EQUAL*



Locking and Unlocking Script

- The access to bitcoins is protected by an **locking script**.
- In order to unlock it, one needs to provide a **unlocking script**.
- The unlocking script succeeds if
  - when first executing the **unlocking script**
  - followed by the **locking script**
  - one obtains a state which fulfils the **accept condition** accept
  - where $\mathrm{accept}(s)$ means that the top element of the stack is > 0 i.e. not false.

P2PKH

The P2PKH script consists of a locking script (scriptPubKey) and an unlocking script (scriptSig) [5].
For clarity:

- The OP_Codes for scriptPubKey are as follows:

OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG

- Locking script checks:
  - the top element of the stack is a public key which hashes to pubKeyHash.
  - the second element on the stack is a signature for the message signed by the public key.
- The scriptSig pushes the required signature and public key onto the stack:
  <sig> <pubKey>

Operational semantics

- The operational semantics of opcodes depends on Time × Msg × Stack . We define it in Agda as the record type StackState.
    - Time: there are instructions for checking that a certain amount of time has passed, and time is used for checking against the current time.
        - opCHECKLOCKTIMEVERIFY: allows to lock a resource until a certain amount of time has passed.
    - Msg is the part of the transaction to be signed when a signature is required.
    - Stack is given as a list of natural numbers.
- All opcodes is given as InstructionBasic.
    - Opcodes can fail, for example if there are not enough elements on the stack as required by the operation.

Cont.

- The operational semantics of an instruction p : InstructionBasic
  ⟦ p ⟧s : StackState → Maybe StackState
- The message and time never change, so ⟦ 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:

  ⟦ opEqual ⟧s = liftStackToStackStateTransformer' executeStackEquality

Cont.

- executeStackEquality has two cases:
  - Fails and returns nothing if the stack has height $\leqslant 1$,
  - Otherwise compares the two top numbers on the stack, replacing them by:
    - 1 in case they are equal,
    - 0 otherwise.
- The component Time of StackState will be used to define the semantics of opCHECKLOCKTIMEVERIFY.
  - opCHECKLOCKTIMEVERIFY fails if the current time is less then the top element on the stack.
- Msg will be used to define the semantics of opCheckSig.

Hoare triple and pre-condition

We define for $\Phi, \Psi \subseteq \mathrm{State}$ and p a Bitcoin Script the Hoare triple with pre-condition

$$\langle\ \Phi\ \rangle^{\leftrightarrow}\ p\ \langle\ \Psi\ \rangle :\Leftrightarrow$$
$$(\forall s \in \mathrm{State1}.\Phi(s) \rightarrow \Psi([\![\ p\ ]\!]s))$$

- The above expresses that if we fulfil pre-condition Φ and run program p we obtain a state in which the post-condition Ψ holds.

Hoare triple and weakest pre-condition

We define for $\Phi, \Psi \subseteq \mathrm{State}$ and p a Bitcoin Script the Hoare triple with **weakest** pre-condition

$$
\begin{aligned}
&\langle\ \Phi\ \rangle^{\leftrightarrow}\ p\ \langle\ \Psi\ \rangle :\Leftrightarrow \\
&\qquad (\forall s \in \mathrm{State1}.\Phi(s) \to \Psi([\![\ p\ ]\!]s)) \\
&\qquad \wedge (\forall s \in \mathrm{State1}.\Psi([\![\ p\ ]\!]s) \to \Phi(s))
\end{aligned}
$$

- The above expresses that
  1. (1) If we fulfil pre-condition Φ and run program p we obtain a state in which the post-condition Ψ holds.
  2. (2) And we obtain this state Ψ only if we had fulfilled pre-condition Φ before.

Hoare triple and weakest pre-condition

We define for $\Phi, \Psi \subseteq \mathrm{State}$ and p a Bitcoin Script the Hoare triple with **weakest** pre-condition

$$
\begin{aligned}
&\langle\, \Phi \,\rangle^{\leftrightarrow}\; p \,\langle\, \Psi \,\rangle :\Leftrightarrow \\
&\quad (\forall s \in \mathrm{State1}.\Phi(s) \to \Psi([\![\, p \,]\!] s)) \\
&\quad \wedge (\forall s \in \mathrm{State1}.\Psi([\![\, p \,]\!] s) \to \Phi(s))
\end{aligned}
$$

- If we take $\Psi = \mathrm{accept}$, and p a locking script, the above means:
  - The locking script only reaches an accepting state starting in state s if $\Phi(s)$ is fulfilled.
  - Therefore a successful unlocking script must compute a state s fulfilling Φ.
  - Therefore who unlocks the script has knowledge of the conditions defined in Φ.

Cont.

For the locking script of P2PKH we compute the weakest precondition Φ, i.e. a Φ such that

$$\langle \Phi \rangle^{\leftrightarrow} \text{ scriptSig } \langle \text{accept} \rangle$$

holds, and show that

$\Phi(s)$
$\Longleftrightarrow$ the two top elements of the stack in s consist of a pubkey hashing to the pbkh and a corresponding signature.

- So the only way to unlock scriptSig is by providing the pubkey and signature required.

Human-readable weakest pre-condition

- A person who builds this script needs to look at this condition and check whether it expresses the conditions the person wants.
- Therefore we need a **human readable weakest pre-condition**.
- In order to support that, we use a **Step by Step approach**.

Our library

- Develop a library in Agda and prove it [1].

Assuming programs prog1, prog2, prog3, and proofs

```
proof1 : < precondition >iff prog1 < intermediateCond1 >
```

means after proof1 pre-condition is weakest pre-condition for prog1 w.r.t. post-condition intermediateCond1.

```
proof2 : < intermediateCond1 >iff prog2 < intermediateCond2 >
```

```
proof3 : intermediateCond2 <=>p intermediateCond3
```

<=>p means both conditions are equivalent predicates.

```
proof4 : < intermediateCond3 >iff prog3 < postcondition >
```

Then the proof for the Hoare triple for prog1 ++ (prog2 ++ prog3) is given as follows:

```
theorem : < precondition >iff prog1 ++ (prog2 ++ prog3) < postcondition >
theorem = precondition      <><>⟨ prog1 ⟩⟨ proof1 ⟩
          intermediateCond1 <><>⟨ prog2 ⟩⟨ proof2 ⟩
          intermediateCond2 <=>⟨  proof3 ⟩
          intermediateCond3 <><>⟨ prog3 ⟩⟨ proof4 ⟩e postcondition ∎p
```

Proof of Correctness of the P2PKH script using the Step-by-Step approach

- P2PKH script:

$\text{scriptP2PKH}^b : (pbkh : \mathbb{N}) \rightarrow \text{BitcoinScriptBasic}$

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

- Intermediate conditions accept_1, accept_2, etc...
  - For example:
    - ★ $\text{accept}^s_1\ m\ t\ st \Leftrightarrow \exists\ pbk, sig, st'.st \equiv pbk :: sig :: st' \wedge \text{IsSigned}\ m\ sig\ pbk$
    - ★ $\text{accept}^s_2\ m\ t\ st \Leftrightarrow \exists\ x, pbk, sig, st'.st \equiv x :: pbk :: sig :: st' \wedge x > 0 \wedge \text{IsSigned}\ m\ sig\ pbk$

Cont.

- Proofs correct-1,correct-2, etc... of corresponding

  correct-1 : < accept$_1$ >iff([opCheckSig]) < acceptState >

  correct-2 : < accept$_2$ >iff([opVerify]) < accept$_1$ >

- Weakest pre-condition

$$
\begin{aligned}
&\mathsf{wPreCondP2PKH}^{s} : (pbkh : \mathbb{N}) \rightarrow \mathsf{StackPredicate}\\
&\mathsf{wPreCondP2PKH}^{s}\ pbkh\ time\ m\ [\,] = \bot\\
&\mathsf{wPreCondP2PKH}^{s}\ pbkh\ time\ m\ (x :: [\,]) = \bot\\
&\mathsf{wPreCondP2PKH}^{s}\ pbkh\ time\ m\ (pubKey :: sig :: st) =\\
&\qquad (\mathsf{hashFun}\ pubKey \equiv pbkh) \wedge \mathsf{IsSigned}\ m\ sig\ pubKey
\end{aligned}
$$

- If the stack has hight 0 or 1 then false.
- If the stack has hight 2 then hold
  if and only if hasFun $pubKey \equiv pbkh \wedge$ IsSigned $m\ sig\ pubKey$

Cont.

- Prove the weakest pre-condition for the P2PKH script as follows

```
theoremP2PKH : (pbkh : ℕ) → < wPreCondP2PKH pbkh >iff scriptP2PKHᵇ pbkh < acceptState >
theoremP2PKH pbkh = wPreCondP2PKH pbkh <><>⟨ [ opDup ] ⟩⟨ correct-6 pbkh ⟩
                    accept₅ pbkh <><>⟨ [ opHash ]       ⟩⟨ correct-5 pbkh ⟩
                    accept₄ pbkh <><>⟨ [ opPush pbkh ] ⟩⟨ correct-4 pbkh ⟩
                    accept₃      <><>⟨ [ opEqual ]      ⟩⟨ correct-3 ⟩
                    accept₂      <><>⟨ [ opVerify ]     ⟩⟨ correct-2 ⟩
                    accept₁      <><>⟨ [ opCheckSig ]   ⟩⟨ correct-1 ⟩e acceptState ∎p
```

  - Used single instructions to prove the correctness of P2PKH.
  - Proofs correct1, correct2 etc. . . are done by the following case ditinctions made in the corresponding verification conditions.

Conclusion

- Specified the correctness of smart scripts by weakest pre-conditions.
- Implemented and tested two methods for developing human-readable weakest preconditions and proving their correctness.
- Applied our approaches to P2PKH and P2MS.
- All the above was implemented in Agda.
- Next talk:
  - Treat conditional OP_IF in Bitcoin script.
- Future work:
  - Develop our approach into a framework for developing smart contracts that are correct by construction.

Thank you for listening.

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

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu.
Relative monads formalised.
Journal of Formalized Reasoning, 7(1):1–43, Jan. 2014.
doi: http://dx.doi.org/10.6092/issn.1972-5787/4389.

Andreas Antonopoulos and Gavin Wood.
Mastering Ethereum. Building smart contracts and Dapps.
O'Reilly Media, December 2018.

Andreas M Antonopoulos.
Mastering Bitcoin: Programming the open blockchain.

" Second ed. O'Reilly Media, Inc.", 2017.

Bitcoin Community.
Welcome to the Bitcoin Wiki.
Availabe from `https://en.bitcoin.it/wiki/Bitcoin`, 2010.

Ana Bove, Peter Dybjer, and Ulf Norell.
A brief overview of agda – a functional language with dependent types.
In *Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

Agda community.
Agda community.
Availabe from `https://agda.readthedocs.io/en/v2.6.2/`, Retrived 10 January 2021.

Nils Anders Danielsson and Ulf Norell.
Parsing Mixfix Operators.
In *Implementation and Application of Functional Languages*, pages 80–99, Berlin, Heidelberg, 2011. Springer.

doi: https://doi.org/10.1007/978-3-642-24452-0_5.

Per Martin-Löf.
An intuitionistic theory of types: Predicative part.
In *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier, 1975.
doi:https://doi.org/10.1016/S0049-237X(08)71945-1.

Ulf 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.

Dejan Vujičić, Dijana Jagodić, and Siniša Ranđić.
Blockchain technology, bitcoin, and ethereum: A brief overview.
In *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–6, 2018.

doi:http://dx.doi.org/10.1109/INFOTEH.2018.8345547.